Ανάλυση κατά τη Μεταγλώττιση για την Υποστήριξη Εκτέλεσης Εφαρμοφών σε Ελαττωματικές Αρχιτεκτονικές / Compiler Analysis for Supporting Execution on Faulty Architectures

Konstantinos Parasyris

July 15, 2015

ACKNOWLEDGEMENTS

I would like to thank my advisors Christos D. Antonopoulos, Nikolaos Bellas and Spyros Lalis for their help and guidance throughout this process, their ideas and feedback have been absolutely invaluable.

This work has been supported by the EC within the 7th Framework Program under the FET-Open grant agreement SCoRPiO, grant number 323872.

"Good judgement comes from experience. Experience comes from bad judgement."

Will Rogers.

ΠΕΡΙΛΗΨΗ

Οι συμβατικές σημερινές τεχνολογίες υπολογιστικών συστημάτων εγγυώνται αξιόπιστη λειτουργία. Παρόλα αυτά η αξιοπιστία αυτή απειλείται από την εξέλιξη της τεχνολογίας. Καθώς τα συστήματα συμπυκνώνονται όλο και περισσότερο και το μέγεθος του τρανζίστορ μειώνεται, το σύστημα γίνεται ευάλωτο σε σφάλματα. Μικρές αλλαγές στη θερμοκρασία ή την τάση του συστήματος μπορεί να το οδηγήσουν σε μη αξιόπιστη λειτουργία. Επίσης, για να μειώσουν την κατανάλωση ενέργειας, τα συστήματα αυτά θα θέτουν την παροχή τάσης δυναμικά σε σημεία όπου θα εμφανίζονται σφάλματα. Τεχνολογίες οι οποίες λειτουργούν σε τέτοιες συνθήκες ακολουθώντας μια συντηρητική λύση, παρέχουμε επιπλέον ζώνες προστασίας οι οποίες στην ουσία παρέχουν επιπλέον χρόνο (time-slack) στο κύκλωμα να υπολογίσει την έξοδά καθώς επίσης και επιπλέον τάση Αυτές οι τεχνικές διορθώνουν όλα τα σφάλματα στο επίπεδο του υλικού και διαφυλάσσουν σε αυτές τις συνθήκες τη σωστή λειτουργία του κυκλώματος. Παρόλα αυτά στο μέλλον ίσως να μην έχουμε τα περιθώρια να προσφέρουμε επιπλέον χρόνο και ενέργειά στο κύκλωμα Παραδοσιακοί τρόποι ενίσχυσης της ανθεκτικότητας ενός συστήματος όπως checkpointing μπορούν να διορθώσουν σφάλματά, παρόλα αυτά και αυτές οι τεχνικές εισάγουν καθυστερήσεις

Πολλές εφαρμογές παρουσιάζουν αυξημένη ανθεκτικότητα σε σφάλματα, είτε διότι περιέχουν πλεονάζων υπολογισμούς είτε λόγω της φύσης της εφαρμογής. Στα πλαίσια της μεταπτυχιακής εργασίας εκμεταλλευόμαστε την αυξημένη αυτή ανθεκτικότητα έτσι ώστε να μειώσουμε το υλικό το οποίο είναι υπεύθυνο για τον εντοπισμό και τη διόρθωση σφαλμάτων. Για την ακρίβεια, δεν είναι όλες οι εντολές ισάξιες. Στα πλαίσια της εργασίας θεωρούμε ότι εντολές οι οποίες υπολογίζουν διεύθυνση μνήμης ή διαχειρίζονται τη ροή εκτέλεσης μιας εφαρμογής είναι πιο σημαντικές από τις υπόλοιπες. Όταν σφάλματα εμφανίζονται κατά τη διάρκεια εκτέλεσης τέτοιων εντολών, είναι πολύ πιθανό η εφαρμογή να μην τερματίσει κανονικά. Επομένως, στόχος της μεταπτυχιακής εργασίας είναι να αναγνωρίσουμε και να προστατεύουμε μόνο τις σημαντικές εντολές. Στην περίπτωση που ένα λάθος επηρεάσει μια μη σημαντική εντολή, ελπίζουμε ότι η εφαρμογή ή το λογισμικό σύστημα θα διορθώσει το σφάλμα. Επειδή πλέον προστατεύουμε ένα υποσύνολο των εντολών έχουμε κέρδος σε ενεργεία, αφού θα χρησιμοποιείται λιγότερο υλικό.

Δημιουργήσαμε μια ανάλυση στα πλαίσια ενός μεταγλωττιστή, η οποία αναγνωρίζει σημαντικές εντολές. Όπως προαναφέρθηκε, θεωρούμε σαν σημαντικές εντολές αυτές που έχουν σαν τελούμενα διεύθυνσης μνήμης είτε διαχειρίζονται τον γράφο εκτέλεσης της εφαρμογής. Χρησιμοποιώντας την πληροφορία από αυτές τις αρχικές εντολές εντοπίζουμε αναδρομικά άλλες εντολές οι οποίες επιδρούν πάνω στον υπολογισμό διευθύνσεων ή

πάνω στο γράφο εκτέλεσης. Κάθε εντολή πλέον έχει χαρακτηριστεί ως σημαντική ή μη σημαντική.

Κατά τη διάρκεια της πειραματικής μελέτης χρησιμοποιούμε ένα εργαλείο εισαγωγής σφαλμάτων το GemFI στο επίπεδο της προσομοίωσης. Το εργαλείο έχει επεκταθεί και πλέον εισάγει σφάλματα μόνο σε μη σημαντικές εντολές. Έτσι, προσομοιώνουμε ένα σύστημα το οποίο προστατεύει από σφάλματα μόνο τις σημαντικές εντολές. Τα αποτελέσματα δείχνουν ότι προστατεύοντας μόνο σημαντικές εντολές η εφαρμογή παρουσιάζει επιπλέον ανθεκτικότητα σε σφάλματα. Επίσης, χρησιμοποιώντας διάφορες βελτιστοποιήσεις του μεταγλωττιστή καθώς και του προγραμματιστή μειώνεται σημαντικά ο αριθμός των σημαντικών εντολών. Άρα, στην ουσία, βελτιστοποιώντας τον κωδικά μειώνεται αρκετά η προστασία που του υλικού κι, επομένως, το κόστος της προστασίας.

ABSTRACT

Conventional computer systems deliver error-free operation. The strict prerequisite of reliability is threatened, due to the continuous efforts towards denser structures. These denser structures are vulnerable to voltage and temperature non-idealities. Next generation technologies will Dynamic Voltage Scale (DVS) processors at the point of the first failure. In such conditions errors occur due to timing violations. The hardware unreliability can be handled by traditional fault tolerance approaches, such as replication or check pointing or recent technologies such as Razor flip flops.

Some application domains demonstrate increased error resiliency due to the their characteristics (redundant computation, iterative algorithms etc.). In this MSc thesis we exploit the resiliency of such application by removing some of the hardware error protection mechanisms. To be more precise, not all instructions are equal in terms of fault vulnerability. Instructions that operate on top of memory addresses or control flow information are more vulnerable to faults (when errors manifest during the execution of such instructions failures are usually observed). The hardware protects only such instructions and the remaining instructions are not protected. Therefore less protection takes place at the hardware, hence reducing the overhead in terms of performance, area, power.

A compile time analysis is devised, using the LLVM infrastructure, which categorizes instructions to critical and non-critical ones. The analysis initially categorizes as critical instructions those that explicitly perform pointer arithmetic or compute control flow information, for example load store instructions and branching instructions. Using the information of the initial critical instructions it identifies instructions that also influence the control flow or the pointer arithmetic. The instruction criticality information is lowered to the executable during the linking process. An extended linker handles the information concerning the criticality of instructions.

In the experimental evaluation we use GemFI, a fault injection tool. GemFI is extended to recognize the criticality of each instruction. Faults are injected only to non-critical instructions. By doing so, we emulate a system in which the hardware corrects faults only when the error is manifested during the execution of a critical instruction. We quantify the increase of application resiliency when protecting only critical instructions. In general the exploration of the criticality information increases substantially the applications resiliency. The less the critical instructions are the less the overhead of protecting instructions will be. To that direction we monitor the percentage of

critical instruction for different compiler optimizations and manual optimizations. In general optimizations greatly increase the number of non-critical instructions.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1

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 to 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 voltage of circuits 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 [11]. 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 [28], 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 [26] suggests when trying to DVS (Dynamic Voltage Scale) 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 the case of multimedia, a few bit errors in the output image or the output stream can be negligible. Likewise the iterative solvers can converge to the desired solution, 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 [2, 27, 31], 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 portions of the computation produce incorrect results.

Nevertheless, all applications contain certain instructions which should always be executed correctly, even if they are within 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 [14, 15]. Those mechanisms try to contain hardware faults and present an error-free execution engine to the software.

Although these mechanisms allow operations below nominal V_{dd} values and are able to correct errors, they may impose a certain degree of performance and area overhead.

The main contributions of my MSc thesis are :

i Introduction of a theoretical background on error detection correction mechanisms and provide a simple model of estimating their performance overheads which is decoupled from the technology trends.

ii Not all instructions have been created equally, pointer arithmetic and control flow instructions should always be executed correctly. To this direction a compilation analysis technique is implemented, which identifies and tags such instructions as critical. This work is based on the LLVM [19] compilation infrastructure. During execution critical defined instructions are protected by the hardware by a respective technology, e.g *Razor Flip Flops*, consequently they are not susceptible to faults. The non-critical instructions are susceptible to faults.

iii Executables should somehow represent the critical information of instructions in order to protect them during execution time. We encode the criticality of instructions as metadata. This metadata are encoded in the executable file. Therefore the *binutils* linker (*ld*) is extended to recognize metadata from object files, associated with the criticality of instructions.

iv The next contribution of my thesis is the extension of GemFI [23], a fault injection tool based on Gem5 [4] cycle accurate simulator.

The extensions include support of fault injection on the *X86* instruction set architecture and an incremental checkpointing mechanism that facilitates fast fault injection campaigns. Finally, GemFI is extended to distinguish critical from non-critical instructions at execution time.

v Evaluation of the extended ISA using the meta-data file and GemFI. Faults were injected in a set of benchmarks while simulating an X86 instruction set architecture. The results are compared to a fault injection campaign conducted on the same benchmarks without ISA extensions, therefore all instructions can be considered as possible fault locations.

vi Identified the influence of compiler and user-made optimizations to the number of critical instructions. The more the critial instructions are, the more error-protection takes place during execution time. Therefore our goal was to reduce the number of critical instructions.

The rest of this thesis is organized as follows. Chapter 2 presents required background on the used compiler and simulator. Chapter 3 we model the impact of error correction mechanism on the system performance. Chapter 4 presents the implementation of the compiler analysis pass as well as extensions made on the linker and the simulator. In chapter 5 the experimental evaluation is presented. Chapter 6 presents related work. Finally in chapter 7 I conclude my thesis.

2

BACKGROUND

This chapters reviews some basic aspects used by this Msc. thesis. It introduces the structure of compilers and the theory behind their operation. *LLVM* the compiler which is used in this thesis is described. Finally we present the simulator and the fault injection tool used by this Msc. thesis.

2.1 INTRODUCTION TO COMPILERS

A compiler is a computer application or a set of applications which translates a source code written in a programming language to a computer language, the target language usually consists of a binary form known as object file. A compiler verifies code syntax, generates efficient object code, performs run-time organization, and formats the output according to target machine specifications and the linker options. A compiler consists of:

- The front end. During the front end of a compilation procedure the syntax and the semantics of the input source file are verified. After the completion of the error checking the compiler generates an intermediate representation (IR) of the source code for processing by the next phase (middle-end). The front end performs type checking by collecting type information. Generates errors and warning, if any, in a useful way. Aspects of the front end include lexical analysis, syntax analysis, and semantic analysis.

- The middle end performs optimizations, including removal of useless or unreachable code, discovery and propagation of constant values, relocation of computation to a less frequently executed place (e.g., out of a loop), or specialization of computation based on the context. Generates another IR for the back end.

- The back end generates the assembly code or the binary file, performing register allocation in the process. It improves throughput and increases ALU utilization by appropriate instruction scheduling. Usually algorithms for optimization are in NP complete therefore heuristic techniques are well-developed and usually adopted by this phase.

2.1.1 *The Data-Flow Abstraction*

The execution of a program can be represented as a function of the program states transformations. A program state is the set of all available variable values. Every intermediate code statement transforms an input state to a new output state. The input/output state is correlated with the application state before/after executing this statement.

When analyzing the behavior of a program all possible paths through the program graph should be considered. Depending on the purpose of the analysis at each program state different information is taken under consideration. It is not possible to keep track of all the program states for all possible paths. In data-flow analysis certain details are abstracted out, keeping only the data needed for the purpose of the particular analysis.

In each application of data flow analysis we associate at each program point a *value* that represents an abstraction of all possible program states that can be observed at that point. The choice of abstraction is connected with the type of the analysis. The data flow values before and after each statement are defined by the *IN*, *OUT* sets respectively. The data flow problem is defined as a solution to a set of constraints on the *IN* and *OUT* for all program statements. There are two kinds of constraints: 1. **transfer functions**, which are connected with the semantics of the statement and 2. **control flow constraints**.

2.1.2 *Transfer Functions*

The data flow values before and after a statement are constrained by the semantics of the statement. Transfer functions come in 2 flavors, information might propagate forward along execution paths or it may flow backwards up the execution paths. We show below a forward transfer function F_s which operates on the values before the statement and produces the new data flow values after the execution of the statement.

$$OUT_s = f_f(IN_s) \tag{1}$$

Accordingly for a backward flow analysis the function is defined as :

$$IN_s = f_b(OUT_s) \tag{2}$$

2.1.3 *Control Flow Constraints*

The second set of constraint on data flow values is derived from the flow of control. In a basic block consisted of the ordered elements OUT $s_1, s_2 ... s_n$ the control flow value *OUT* of s_i is the same as the control flow value *IN* of s_{i+1}. However control flow edges between basic blocks create more complex constraints between the last statement of a basic block and the first statement of the proceeding block.

2.2 INTRODUCTION TO LLVM

LLVM implements a compiler framework, aiming to provide lifelong program analysis and transformations for arbitrary programs. The entire procedure is transparent to the programmers. The success of the project is contributed to:

- An independent machine code representation, that serves as a reference point for analysis, transformation and code distribution.
- A design based on this representation adaptable to provide non-traditional compilation capabilities, for example user developed optimization phases can be plugged into the LLVM tool.

LLVM's intermediate representation is based on SSA (Static Statement Assignment) form coupled with RISC like code. However higher level information is provided by different structures, for example control flow graphs, explicit data representation and use-def chains. All these structures increase the effectiveness of analysis and transformation techniques.

2.2.1 *LLVM Instruction Set*

The LLVM instruction set avoids machine specific constraints of modern processors, e.g pipelines, physical registers, however it captures the key operations of such machines. LLVM provides an infinite set of virtual values which can hold any C++ primitive type (Boolean, integer, floating point and pointer). These registers are in Static single assignment (SSA) form. LLVM features a load store representation, so data are moved from and to the memory explicitly.

The entire instruction set consists of 31 opcodes. Using the C++ overloading abilities most opcodes are overloaded, so the add instruction can operate on any integer, floating operand without the introduction of a new opcode. Almost all instructions use a three/two-operand form (they take one or two operands and produce a single result). The instruction set implements an explicit ϕ instruction, which corresponds directly to the standard ϕ function of SSA form. SSA form provides a compact def-use graph that simplifies many data-flow optimization's and enables fast, flow insensitive algorithms to achieve many of the benefits of flow sensitive optimization's without expensive data flow analysis.

LLVM also makes the Control Flow Graph (CFG) of every function explicit in the representation. A function is a set of basic blocks, and each basic block is a sequence of LLVM instructions, ending in exactly one terminator instruction. Moreover the LLVM type system includes source language independent types with predefined constant

```
%p = getelementptr %xty* %X, long %i, ubyte 3
store   int 1, int* %p
```

Listing 2.1: The getelement ptr instruction

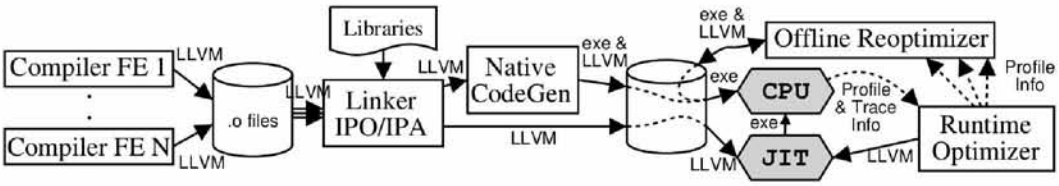


Figure 1.: LLVM system architecture diagram [19].

sizes, (void, bool, signed/unsigned (1byte to 8 byte). There are 4 extra data types, pointers, array, structures and functions. Higher level data structures can be represented using a combination of the aforementioned types.

LLVM abstracts pointer arithmetic using one instruction, called *getelementptr* , this instruction preserves type information and is machine independent. Given a type pointer to an object of some aggregate type, this instruction calculates the address of a sub-element of the object in a type-preserving manner (effectively a combined '.' and '[ ]' operator for LLVM). For example the C statement:

$$X[i].a = 1;$$

can be translated into the LLVM instruction set in the 2 commands presented in Listing 2.1 where we assume a is field number 3 within the structure $X[i]$, and the structure is of type $\%xty$.

2.2.2 *High-Level Design of the LLVM Compiler Framework*

In Fig 1 we depict the high level flow chart of the LLVM architecture. The front end translates the C input code to the LLVM intermediate representation and emits the corresponding code which is combined together by the LLVM linker. The linker performs a variety of optimization's applying extra effort to inter-procedural ones. The resulting LLVM code is then translated to native code for a given target at link time or install time. Another option is to translate the code at runtime with a just in time translator. The native code generator inserts light-weight profiling hooks to gather information about frequently executed code regions and these can be optimized at runtime. The profiled data can be collected by the end user and used in an offline optimizer to perform aggressive profile driven optimization's for the specific target machine.

We should mention that during the link time different optimization phases can be plugged in, using the LLVM pass manager A.

Moreover during compilation time LLVM procedure LLVM instructions are represented with different *incarnations* when it goes through LLVM's multiple compilation stages. These representations are described in B. Finally in C we describe the object file generation performed by LLVM.

2.3 GEM5 DESCRIPTION

Gem5 is a popular open-source system simulator. It provides a modular platform for computer system-level architecture research, encompassing system-level architecture as well as processor micro-architecture.

Object oriented design enhances the flexibility of Gem5. The ability to construct configurations from independent objects facilitates multicore and multi-system design. Moreover, Gem5 provides four different CPU models, each of them representing a different point in the speed vs simulation accuracy trade-off. *Atomic Simple* is a single *IPC* CPU model. *Timing Simple* is similar but also simulates the timing of memory references. *InOrder* is a pipelined in order CPU. Finally, *O3* is a pipelined out-of-order CPU model. Gem5 also supports two memory system models: *classic* and *ruby*. The classic is fast and easily configurable, while the ruby model provides a flexible infrastructure capable of accurately simulating a wide variety of cache coherence memory systems.

Gem5 operates in two modes: *System Call Emulation (SE)* and *Full System (FS)*. In SE mode applications execute on simulated "bare metal". Whenever the program executes a system call, Gem5 traps and emulates the call usually by passing it to the host OS. Currently there is no thread scheduler in SE mode. Therefore, threads are statically mapped to a core, hindering its use with multi-threaded applications. FS mode offers an environment for running an operating system (OS) on top of the simulator. There is support for interrupts, exceptions and I/O devices. Applications are executed under the control of the OS.

Gem5 supports a number of ISAs, including Alpha, MIPS, ARM, Power, SPARC and x86. The simulator's modularity allows these different ISAs to be easily implemented on top of the generic CPU models and the memory system. Alpha is the most maturely supported ISA, with ARM and x86 following.

2.4 GEMFI DESIGN AND IMPLEMENTATION

GemFI is developed using C++ and Python. It fully supports the Alpha instruction set architecture. Supporting more instruction sets is rather straightforward GemFI supports full system simulation mode as well as the execution of multi-threaded applications.

GemFI provides an API consisted of two intrinsic functions.

- **void fi_activate_inst(int id)** is translated to a pseudo-assembly instruction. Its successive occurrences toggle (active/inactive) the manifestation of faults for the specific process/thread. The executing thread is assigned a numerical *id* which can be used as an identifier of the thread in fault injection configuration.

- **void fi_read_init_all()** checkpoints the simulation. Upon restoring from the checkpoint, it resets all the internal information of GemFI, allowing the same checkpoint to be used as a starting point for multiple experiments with potentially different fault injection configurations.

Faults are described in the input file provided by the user at GemFI command line. The file is parsed at startup and each fault is inserted to one of five internal queues. Each queue corresponds to a different pipeline stage.

On each simulation tick, GemFI checks if fault injection has been enabled for the running thread. In such a case, it prefetches the corresponding *ThreadEnabledFault* objects. Then and for each instruction served at a pipeline stage, GemFI updates the thread's data and scans the corresponding queue for faults targeting the executing thread at the specific simulation point. Queue entries are sorted according to the timing of each fault. If such a fault is found, the value of the targeted location is corrupted according to fault's behavior.

3

PERFORMANCE MODELING

This chapter presents a simplified performance overhead model when executing on unreliable hardware. We model an abstracted error detection and correction mechanism which takes place at hardware using Razor Flip Flop technologies [14, 7].

3.1 SIMPLE APPROACH

Our model takes into account the instruction mix of the executing code in terms of critical/non-critical instructions. We assume that detection of an error does not induce any performance overhead, which is consistent with publications regarding razor flip flops [5, 14, 15]. However the correction part does induce performance overhead, which is highly related with the used razor technology. Our model is abstracted from a specific technology trend, and assumes that error correction induces a penalty of EC (stands for Error Correction)[1] cycles.

Execution time can be modeled using Equation 3 in an error-free platform. An extention to the aforementioned equation, is Equation 4 which does consider possible errors. $ER(V)$ denotes the error rate, is a function of the supply voltage, and returns the average number of errors expected in every clock cycle. The term $ER(V) * Cycles$ estimates the number of errors inserted during execution. Finally this term is multiplied by the penalty for correcting each error.

$$T = Cycles * Freq \tag{3}$$

$$T_{all} = (1 + ER(V) * EC) * Cycles * Freq \tag{4}$$

In [5] the error rate (ER) is estimated to be one error every 10 million cycles when operating on the point of first failure (PoFF)[2] However past that point the error rate increases exponentially by an order of $10mV$ supply voltage decrease. When operating on the PoFF the energy gains vary from 35% to 45% [5].

1 Different razor technologies induce different error correction penalties, therefore we model it as a variable which is defined by the respective technology

2 The clock frequency/voltage is set in such a way, that the clock cycle is equal to the critical path with no extra margins (guardbands)

Equation 5 expresses the number of total executed instructions. We define as p the propability of a fault to manifest during the execution of an instruction I. We assume that all instructions have the same probability of failure. However when we categorize the instructions into critical and non critical the error correction overhead will take place only when the erroneous instruction is a critical one. The probability that an instruction is critical (P_{crit}) is expressed by equation 6.

$$Exec_{inst} = Exec_{critical} + Exec_{non\_critical} \tag{5}$$

$$P(I, I \in Critical) = \frac{Exec_{critical}}{Exec_{inst}} = P_{crit} \tag{6}$$

So when the application code is split into critical and non critical instructions the execution time under the presence of faults can be estimated using Equation 7. When comparing 7 with 4 it is obvious that the less the critical instructions are, the less overhead is induced by the hardware correction mechanisms, since errors are going to be corrected only if the corrupt a critical instruction.

$$T_{crit} = (Cycles + ER(V) * Cycles * EC * P(I, I \in Critical)) * Freq \tag{7}$$

Performance depends on the percentage of critical instructions and the error rate. The more the critical instructions, the more the error correction takes place. In a worst case scenario all instructions would be critical and the error rate would be equal to 1. In such a case the performance penalty would be equal to $Cycles * EC * Freq$. However,

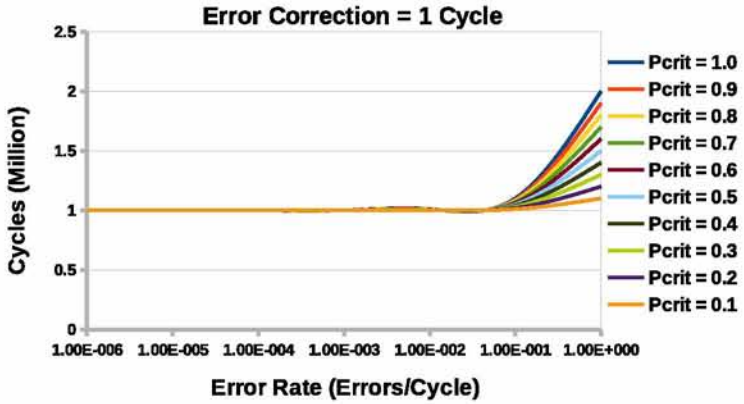


Figure 2.: Modeled execution time in Cycles for different error rates and different percentages of critical instructions in the instruction mix. PCrit denotes the probability of an instruction to be critical.

in a realistic scenario not all instructions are going to be critical. Figure 2 presents the increase of the execution time for different error rates. The hardware mechanism seems to withstand error rates up to 10^{-1}. From that point the execution time rapidly increases. When assuming different instruction mixes the correction overhead lineary drops by a factor of $1 - Pcrit$. Note that these are not real data, this is just a projection of our model.

3.2 INSTRUCTION LEVEL VUNERABILITY AWARE APPROACH

When the supply voltage is scaled below the nominal value, a digital system may suffer from timing violations. Instructions which activate paths with timing close to the critical path tend to fail frequently. In [26] the instructions are catergorizied into classes. We assume all instructions can be classified in n classes. Each class of instructions has a failure propability PF_i. During an execution the number of failures can be computed by Equation 8.

$$Errors = \sum_{i}^{n} PF_i * \frac{\#Inst_{i \in Class_i}}{Total_{inst}} \tag{8}$$

Using the notion of critical instruction, Equation 8 can be rewritten as Equation 9 to calculate the number of errors the hardware should correct. Equation 9 corresponds to the probability of an instruction that are classified in a class and are critical during application execution. The term $\#CriticalInst_{i \in Class_i}$ corresponds to the number of instructions which belong to $Class_i$ and are identified as critical for the application execution.

$$P_{Correction} = \sum_{i}^{n} p_i * \frac{\#CriticalInst_{i \in Class_i}}{Total_{inst}} \tag{9}$$

$$T_{crit} = (1 + ER(V) * *EC * P_{Correction}) * Freq * Cycles \tag{10}$$

Equation 10 summarizes the expected execution time of an application for a classified instruction set architecture with different probabilities of failure for each class.

4

IMPLEMENTATION

The source code of all applications consists of 3 types of instructions: a) Those that operate on control flow information. For example instructions which correspond to *for* loops or *if* statements. b) Those that perform pointer arithmetic and calculate addresses to load values from and store addresses to. c) Those that compute data values required for the final output of the application. Basically such instructions are any instruction not contained on the previous two categories.

In Listing 4.2 and 4.1 we present a simple vector addittion in C and the *MIPS* assembly implementation of the vector addition respectively. Line *6* of the assembly version corresponds an instruction operating on top of data, this instruction does not affect explicitly or implicitly any memory address or the control flow of the application. Lines *1*, *3*, *11*correspond to instructions operating on the control flow and all the remaining instructions correspond to pointer arithmetic. 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 an HW/OS trap) more frequently than faults impacting data-instructions. The same applies for instructions controlling control flow.

```
1  add $s1 $o $o
2  for
3  beq $so, $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 4.1: Vector add used a simple example.

```
for ( k = 0; k < SIZE ; k++)
    C[k] = A[k] + B[k];
```

Listing 4.2: Vector add used a simple example.

Should someone compare the importance of instructions in relevance to application resiliency, instructions operating on top of data should be the least important. Instructions operating between data might mask a fault, or in any case they rarely result to program fauilures. Therefore, protecting such instructions in the hardware may result to unnecessary waste of resources since errors might never manifest at the end result. 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 chapter present a description of the *LLVM* compiler pass that detects critical instructions in an application. Those instructions should be error-free, or the program will, most probably suffer from crashes. We also present linker extensions to merge information of the crital and non-critical instructions Finally we present the implementation of extensions on GemFI.

In Figure 3 we present the flow chart of tools used to implement and validate the critical instruction identification analysis. Initially the source and header files are passed to *Clang*, the front end of *LLVM*. *LLVM* processes the output of *Clang* and performs optimiza-

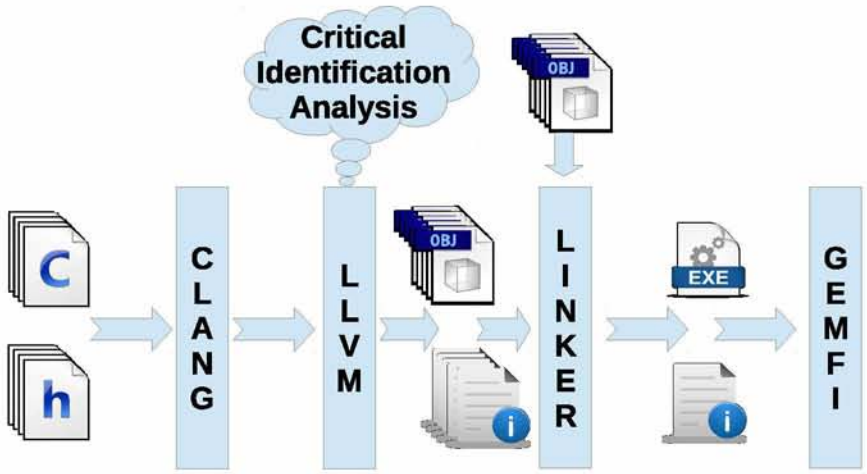


Figure 3.: The interaction of tools used to implement critical instruction identification and the evaluation of the effect of targeted instruction protection to application resilience.

tions. During this phase our analysis takes place. *LLVM* outputs two types of files, the object files and the meta data files which contain information about the criticality of the instructions. Both files are fed to an extended linker, which creates an executable and a metadata file. Finally, these two files are used by the *Gem5* simulator through an extended version of GemFI to complete the fault injection campaign.

4.1 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 :

ClassI: During the execution of the instruction an address calcualtion is performed. For example the *ld* instruction of the *MIPS* architecture.

ClassII: The instruction has implicit or explicit impact on the control flow of the application. For example a branch instruction has explicit impact on the control flow whereas a compare instruction has implicit impact.

These instructions operate by definition on top of critical information, e.g adresses or data flow. Therefore, in our analysis we use the operands used by such instructions to identify other instruction which operate also on top of critical information. In other words, the analysis propages information from the obvious critical instructions to all the instruction of the application.

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 **ClassI**, only the operands participating in the address calculation are pushed to the *GEN* vector. If the instruction is in **ClassII**, 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, however, there is no definition of x between

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

s and the beginning of the basic block. Equations 11, 12 present the transfer function.

$$GEN(I_0) = \emptyset$$

$$IN_n(I) = GEN(I_n) = (GEN(I_{n-1}) - defs(I_n)) \cap f(I_n) \tag{11}$$

$$f(I_n) = \begin{cases} uses(I_n), & \text{if } I_n \in ObviousCritical \\ uses(I_n), & \text{if } defs(I_n) \in GEN(I_{n-1}) \\ \emptyset, & \text{Otherwise} \end{cases} \tag{12}$$

After the procedure traverses the entire block, it propagates the information to all the predecessors of this block using the union operator (Equation 13). We apply this operator to the analyzed code iteratively until there are no changes in the *GEN* set.

$$\forall B_i \in Predecessor_B | OUT(B_i) = OUT(B_i) \cup IN(B) \tag{13}$$

The analysis iterates continuesly on the basic blocks of the function until there is no change between consecutive iterations.

4.1.1 *Example*

Figure 4 on the left shows the assembly of a vector addition application and on the right shows the values of the *GEN* vector as they propagate for each analyzed instruction. Starting from the last block of the code (*node 12*) the *GEN* is an empty set. The set is propagated to the predecessor blocks (*node 2*). Afterwards the algorithm processes the next block which in this case ends with node 11. The instruction is an obvious critical instructions therefore the instruction is tagged as critical, however since the instruction has no *Register* operands the *GEN* set remains empty.

Instructions 10-7 are not obvious critical ones and the GEN set is empty, therefore, there is no addition of operands in the *GEN* set and none of these instructions are identified as critical. Instruction 6 is a store word, therefore, it is contained in the obvious critical instructions and the *USE* are pushed into the *GEN* set. Instruction 5 does not define any operand contained in the *GEN* set hence the set remains the same.

Instructions 4,3 both load values from the memory and are considered as obvious critical ones. The *USE* operands of this instructions are pushed into the *GEN* set. Instruction 2 affects the control flow of the application so our analysis sets as critical the instruction and pushes all the uses of the instruction into the *GEN* set. At this point the *GEN* set is propagated to the *GEN* set of blocks 1 and 3.

Finally the procedure processes the last basic block. The basic block contains only instruction 1, which is considered as critical due to the definition of register $s1 which is contained in the *GEN* set. Note that register $1 is removed from the *GEN* set, but register $0 is not included in the *GEN* set because in the *MIPS* instruction set architecture register $0 is always equal to 0.

Figure 5 presents the second iteration of our algorithm. Starting again from the bottom of the *CFG*, node 12 has inherited the GEN set of their predecessors. Instruction 11 has already been tagged as critical and has no operands therefore the *GEN* set remains the same. Instructions 7-10 define values which are contained inside the *GEN* set so the instructions are tagged as critical. Initially when processing these instructions their definitions are removed from the *GEN* set. Afterwards when processing their *uses* , the same registers are pushed back into the *GEN* set[2]. Instruction 6 is already tagged as critical and it does not define any value inside the *GEN* vector. Instruction 5 does not define any value contained in the set, therefore the instruction

2 When instructions define and use the same registers. We process them in an hieararchical order, firstly process definitions set and afterwards we process the uses

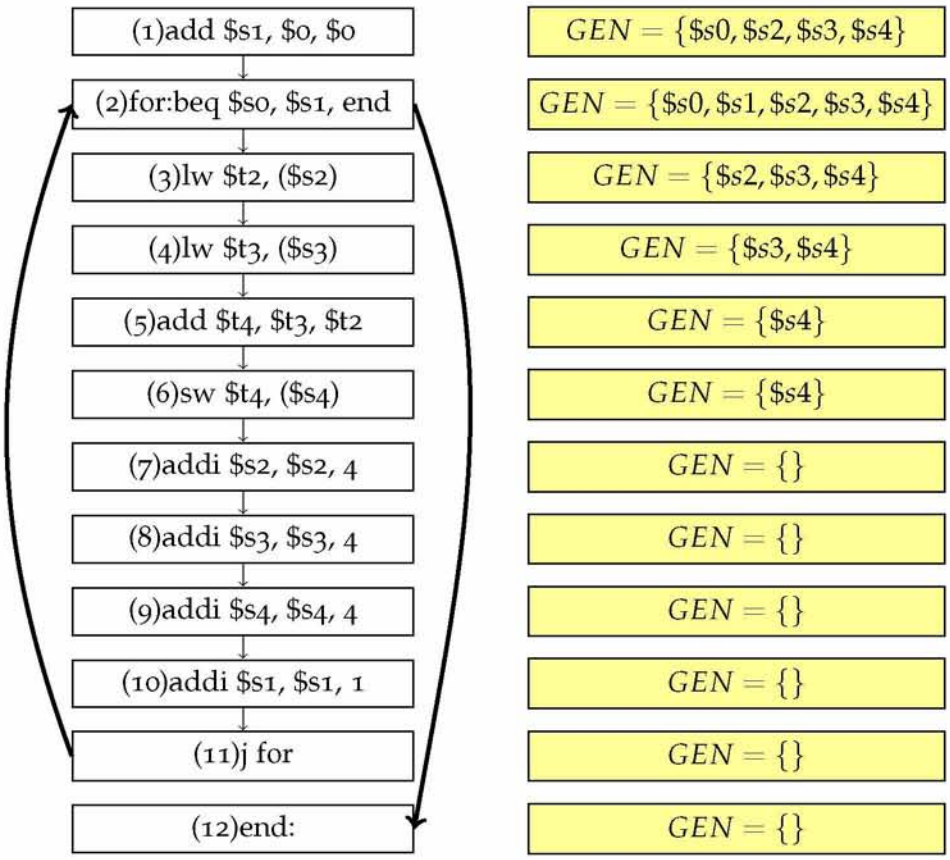


Figure 4.: On the left there is the CFG of MIPS assembly of a vector add, on the right the GEN set is produced as we move from the last instruction up to the first.

remains the same. For the remaining instructions the procedure is the same as in the previous iteration.

In figure 6 we present the third and last iteration of our algorithm. The *GEN* set remains unmodified, therefore our algorithm terminates.

4.1.2 *Implementation*

The analysis handles each function separately, therefore we register our analysis as a *Machine-Function* pass, a pass that operates on top of the internal LLVM machine dependent instruction representation. All instructions should be analyzed and grouped into critical and non-critical ones. To avoid loss of information due to other optimization which may modify the instruction stream, we register our analysis as a *pre-emmit* pass hence the analysis is performed just before emitting the instructions to their binary representation (*MCInst*). To identify obvious critical instructions we use member functions of the *MachineInstr* class. To be more precise, we use the following build-in functionsi: *isBranch(), isCall(), isReturn(), isCompare(), mayLoad(), may-*

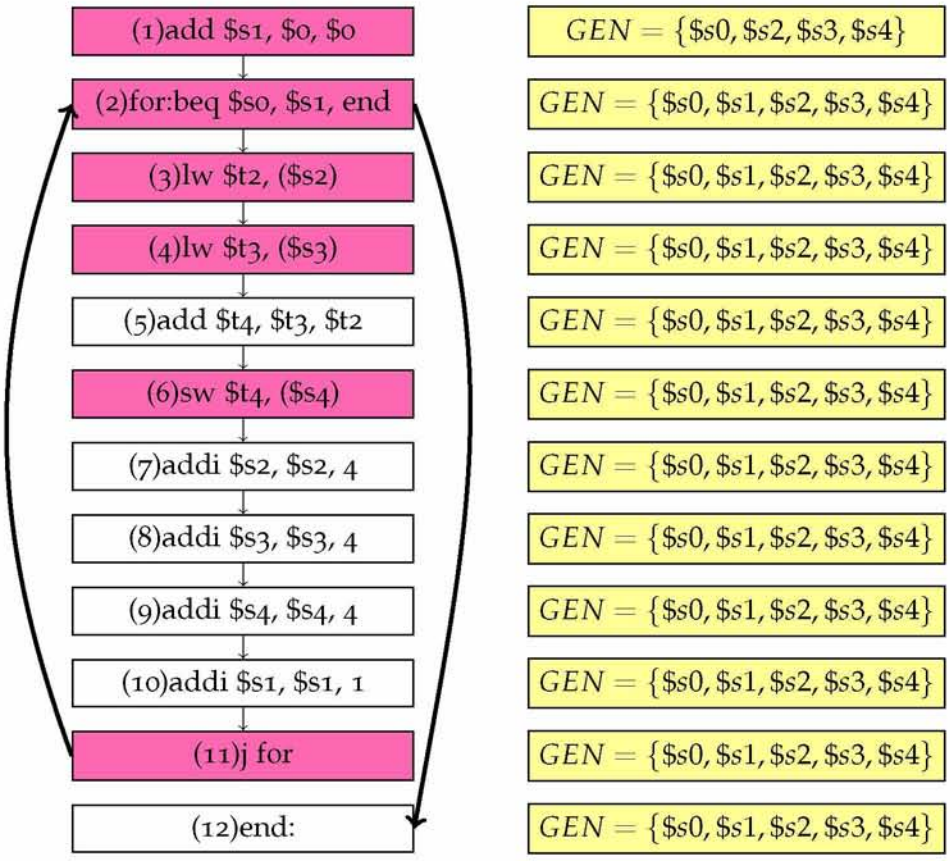


Figure 5.: The red rectangle's present instructions which were tagged as critical. On the right side of the figure the second iteration of the algorithm takes place.

Store() . Moreover, for the *x86* instruction set we manually check the *opcode* of the instruction against the *load effective address opcodes (lea)*. *lea* instruction computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the addressing modes of the processor; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

In the *x86* instruction set, branches or function calls do not have any register operands. Therefore such instructions are critical however, they do not further interact with our analysis. In the case of returning from a function call (isReturn()) again the instruction is considered as critical, however the optional use of this instruction is not recorded in the *GEN* vector, because we want to protect the PC address calculation of the call instruction not the returning value. In the case of compare instructions, all use-register operands are recorded in the *GEN* vector. For *load, store* instructions we consider only the operands which participate in the calculation of the source/destina-

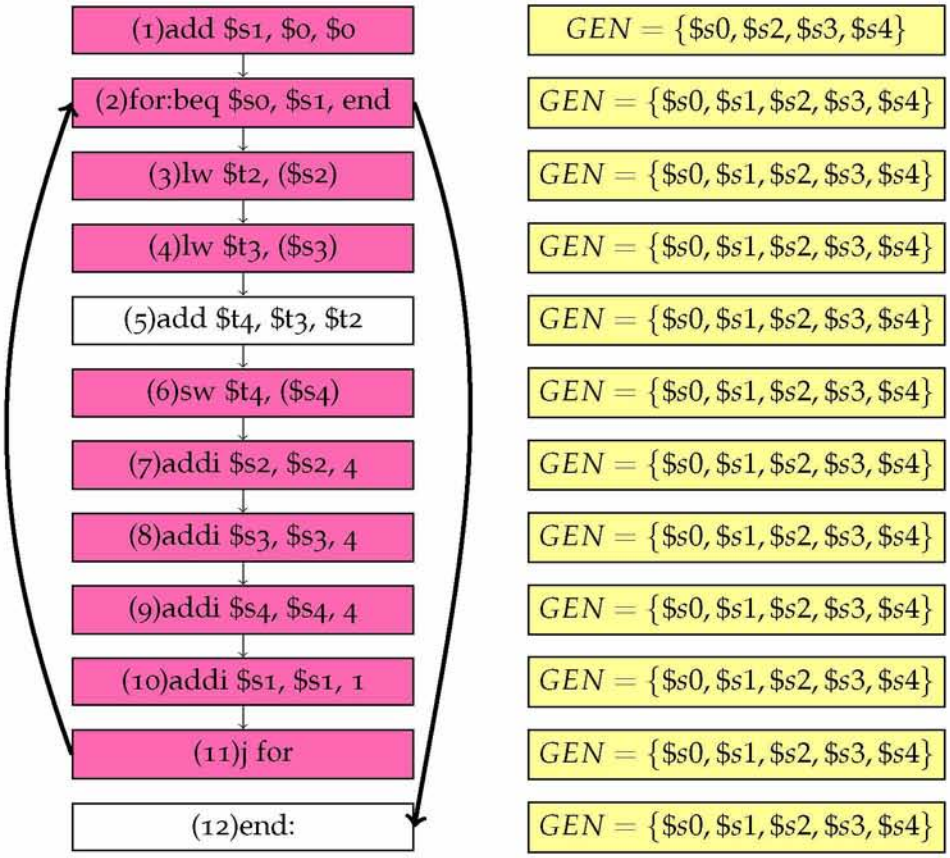


Figure 6.: The last iteration of our algorithm

```
bool protectionAnalysis( Function &F){

//Initialize all vectors to an empty set.
//BB stands for BasicBlock
    for (BB = F.end();  B !=F.start() ; BB++)
        BB.GEN.init (false);

//Traverse the CFG in reverse order and apply the tranverse
    function.
    while ( noChanges )
        noChanges = false;
        for (BB = F.end();  B !=F.start() ; BB++){
            for ( I = BB.end() ; I != BB.Start() ; I++ ){
                if ( IsObviousProtected(I) ){
                    I.Protected = true;
                    noChanges = true;
                    propagateProtection(I);
                }
                else if ( I.defs() in BB.GEN ) {
                    noChanges=True;
                    I.Protected = true;
                    propagateProtection(I);
                }
        }
// Propagate the IN to the out of the predecessor blocks.
        for ( P = BB.predecessors() ; P !=NULL ; P++)
            P.setGen(BB.GEN);
        }
    }
}

void propagateProtection(Instruction I){
    for ( Operands in I )
        if (Operand.isReg () &&  Operand.isDef() )
            BB.GEN[Operand.getReg()] = false;

    for ( Operands in I )
        if ( Operand.isReg () && Operand.isUse()  )
            BB.GEN[Operand.getReg()] = true;
}
```

Listing 4.3: A C++ pseudo code demonstrating the main driver of our algorithm.

tion address. Finally, all use/def operands are considered when processing instructions which are not contained in the obvious critical set but need to be critical due to implicit dependencies of critical instructions upon them. For example instructions *7*, *8*, *9* Figure 6 are not contained into the obvious critical set, but are tagged as critical due to the implicit dependency of instructions *3*, *4*, *6* upon them.

In Listing 4.3 we provide a *pseudo*-C++ like implementation of our analysis.

```
1 movslq  -32(%rbp), %rax# CRITICAL:1
2 movq    -8(%rbp), %rcx# CRITICAL:1
3 movl    (%rcx,%rax,4), %edx# CRITICAL:1
4 movslq  -32(%rbp), %rax# CRITICAL:1
5 movq    -16(%rbp), %rcx# CRITICAL:1
6 addl    (%rcx,%rax,4), %edx# CRITICAL:1
7 movslq  -32(%rbp), %rax# CRITICAL:1
8 movq    -24(%rbp), %rcx# CRITICAL:1
9 movl    %edx, (%rcx,%rax,4)#CRITICAL:1
```

Listing 4.4: x86 Assembly corresponding to the inner block of a vector add

4.1.3 *Object/Assembly File Creation*

LLVM after optimizing the source code, the binary creation takes place. The binary creation in LLVM is supported by the *MC* interface. To start the binary creation the internal represantion of instructions is changed from the *MachineInstr* class to the *MCInst* class. As the transition from *MachineInstr* to *MCInst* takes place, the criticality of each instruction is tranfered to the *MCInst* represantation.

From this point on the compiler either emits assembly files or creates an object file. The assembly file displays the criticality of the instructions in the form of comments. At the end of each instruction the compiler prints the *string #CRITICAL:X*, as presented in Listing 4.4. Critical instructions have *X* equal to 1 whereas non-critical instructions have the value of 0. This information is produced mainly for debugging purposes and cannot be transformed back to any binary representationi, since it would require modification of the assembler parser to recognize such information and encode it.

The x86 instruction set architecture follows the *Complex instruction set computing (CISC)*, in which a single instruction, in the decoding stage, may be translated to multiple simpler micro-operations. The x86 instructions use variable-length encoding: an instruction can be anywhere from 1 to 15 bytes in length whereas in RISC architectures such as ARM the instruction size is either 16 bits or 32 depending on the CPU mode. In Figure 7 we present the instruction format as presented in the Intel manual volume 2A.

We do not extend the x86 instruction set to encapsulate the criticality information, since this would require extensive modifications to the compiler and the simulator, and would effectively result to a new x86-like ISA. Upon object file creation we encapsulate critical information into a separate file called metadata (MD) file. The file contains the critical identification for each compiled instruction. Again critical instructions are encoded with the value of *1* whereas non-critical instructions have the value of 0. During the object file creation, for every byte emitted to the object file we emit a byte to the MD file. If

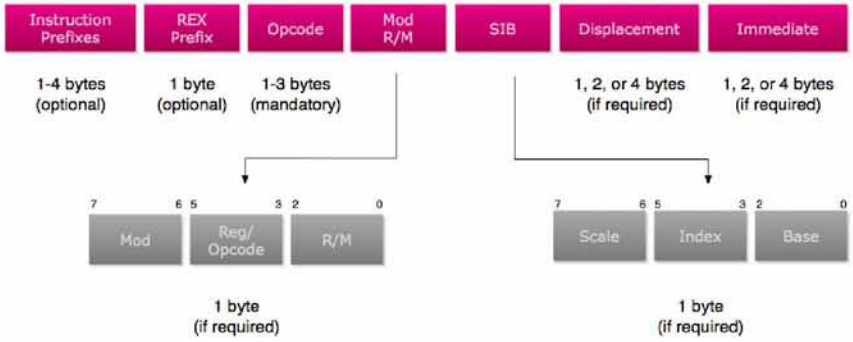


Figure 7.: Instruction format of an x86 instruction [13]

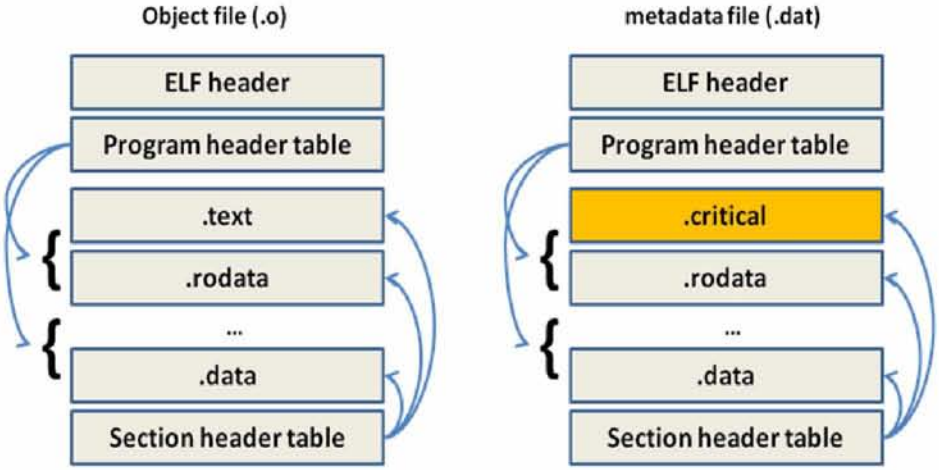


Figure 8.: Format of the metadata object file

the emitted byte corresponds to an instruction we emit the criticality information. If the emitted byte does not correspond to an instruction (e.g data segment) we just emit the same value to the MD file. By doing so the two files are identical except of the .text segment, where in each byte we store information on the criticality of the instruction encoded in the respective byte of the object file. Figure 8 presents the format of the metadata file in conjuction with the format of the object file.

4.2 LINKING

As the last step of the binary creation phase all the object files accompanied with the *MD* files should be linked into two separate files, the executable file and a second one containing the criticality of each instruction. LLVM does not provide a linker, therefore we extended the *ld* linker from the *binutils (GNU ld (GNU Binutils) 2.24.51)*. The *bfd* library is extended to support the linking of the metadata files. Again the final MD file should contain an exact match of the executable

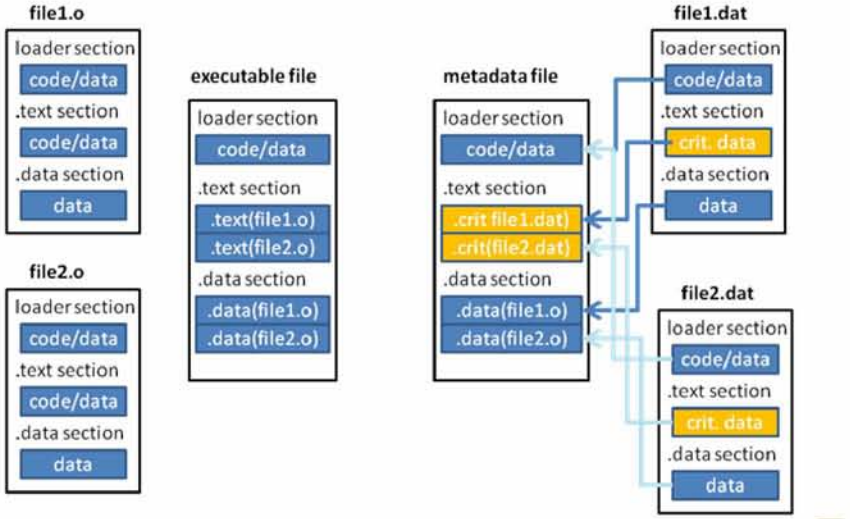


Figure 9.: Procedure of the linker to create a binary and a metadata file.

bytes to the critical bytes. Figure 9 present the procedure followed by the extended linker. In essence the linker combines each section of the object files to a grouped new section. We follow the exact same procedure and combine in the same way the metadata files.

The *struct bfd*, which is is the structure representing an object or executable file was extended to store criticality information. Moreover, the structure representing a section was extended. The section consist of two vectors: the first contains the information of the original section, whereas the second vector contains the information about the criticality of instructions. In the case there is no MD file, for example when linking with an external library all the metadata are set to *1*. By doing so we protect all instructions which are not compiled by our framework.

When the writing of the final executable takes place, each object file is processed sequentially, writing the data of each section in predefined file locations. We take care to write the final metadata of that section on a secondary file. All the data contained in the metadata file point at the exact same locations (offsets from the beginning of the file). At the end of the linking two files with the exact same size are created. The second one contains information about all instructions inside the executable file.

4.3 GEMFI EXTENSION

4.3.1 *Fault injection in x86 Architectures*

The executable coupled with the *MD* files are executed on the Gem5 simulator using the GemFI fault injection framework. GemFI was

extended to support fault injection on the x86 instruction set architecture. To identify the executing thread at the hardware level we use the base of the *fs* register. This register points to the starting address of the structure that represents a software thread within the oprating system. When the OS context switches applications, the *fs* register is automatically set to point to that location, which is unique for all live software threads.

The second step was to decide which structures should be corrupted at the execution stage. When executing load store micro-ops, faults corrupt the address calculation of the corresponding load/store. On control flow instructions faults corrupt the calculation of the next fetched address. The remaining instructions corrupt the result of the instruction. For the remaining pipeline stages no further enhancements were applied on the original GemFI implementation.

4.3.2 *GemFI Performance Enhancement*

GemFI checks, on each clock cycle whether the executing thread has enabled fault injection. This checking is performed by searching a thread identifier inside a map [3]. To avoid checking the map on each cycle we monitor context switches on the hardware. Writes on the thread identifier register are monitored[4]. If the thread to be executed has enabled fault injection, the running core sets a pointer to the corresponding *ThreadEnabledFault*[5] object. By doing so we check the map only when this registers are written.

4.3.3 *Dual ISA extensions*

To limit fault injection only to non-critical instructions, from the command line, the path to the MD file is specified. At the fetch stage, the instruction bytes are read together with corresponding *MD* bytes. In order to read the correct bytes from the MD file we use a function which takes as an argument the current PC address and returns an offset from the beginning of the file. (Equation 14). The *FirstBinaryAddress* is provided by the injected application using an extended version of the GemFI function *fi_read_init_all()*. The new *fi_read_init_all(unsigned int Start, unsigned int Stop)* takes two arguments, the starting virtual address which the loader has mapped the executable and the ending address. This values can be found in any C application when reading the addresses of the variables *__executable_start*, and *__etext*.

$$Offset(PC) = PC - FirstBinaryAddress \tag{14}$$

3 A standard implementation of a hash-table in C++
4 *fs base* for the x86 architecture and the *PCB* register for the alpha
5 GemFI internal represantation of a software thread that has enabled fault injection.

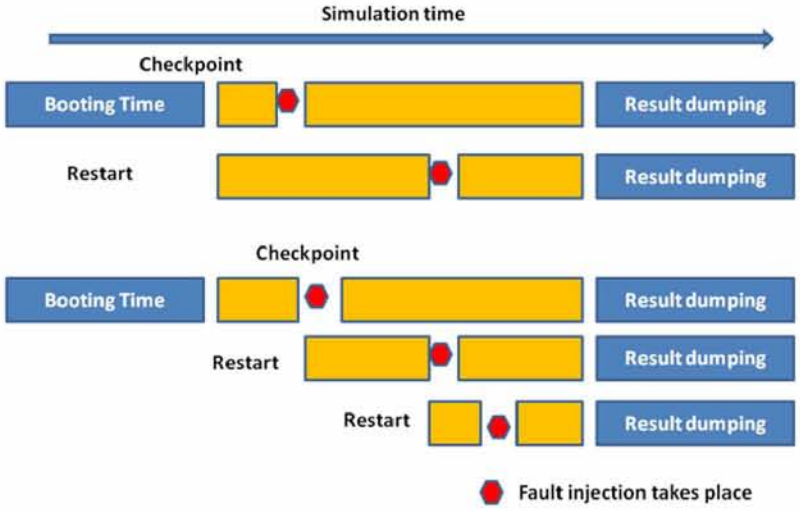


Figure 10.: Procedure of the linker to create a binary and a metadata file.

At the decoding stage, the criticality information is appended to the decoded instructions. Therefore, GemFI can decide whether to inject faults or not using this information.

4.3.4 *Fault Injection Campaings Via checkpointing*

GemFI uses an external checkpointing tool, called DMTCP [1], to create snapshots of the simulated system. We extended the GemFI checkpointing mechanism to support incremental checkpointing. In the top part of the Figure 10 we present the traditional mechanism of GemFI. The user instructs the tool when to take a checkpoint. Using that checkpoint the user can fast forward the simulation to that point and inject another fault. On the lower part of the Figure 10 the increamental checkpointing mechansism is described. Just before injecting a fault GemFI creates a snapshot of the correct simulation state. Afterwards the simulation continues and in the end the results are gathered. The next fault injection campaing can restore from the checkpoint. By doing so for each experiment the user fast forwards the simulation to the point of the prior fault injection. This option is provided by the command line option *–checkpoint-on-fault*.

5

EXPERIMENTAL EVALUATION

In this chapter we validate the compilation, linking and simulation infrastructure and we evaluate the potential cost and benefits for reliability by using fini-grained protection at the instruction level. We use three benchmarks, *Sobel, DCT, Blackscholes.*

5.1 METHODOLOGY

Our goal is to evaluate first the extent of protected instructions in various codes, and the evaluate the effect of code optimizations on the number of protected instructions. The analysis is limited to identify critical instructions to source code only, functionality offered by external libraries is not analyzed by LLVM. Therefore, there is no criticality information on the instuctions contained in such libraries. To that direction we limit the extend of dependencies to external libraries to reduce the number of non-analyzed instruction. We assume that protecting to many instructions will be very expenive in terms of area and power. To this direction we implement different versions of the benchmarks. In each version we increase the extend of manual optimizations and thus programmer 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.

All binaries are executed on the simulator. We count the number of critical and non critical instructions. At the end of the simulation these values are reported. Note that our goal is to correlate compiler optimization, programmer's optimizations with the percentage of non-critical instructions.

Finally the best version in terms of the number of non critical instructions is subjected to a single fault injection simulation campaign. We inject faults into the pipeline stages *Fetch, Decode, IEW, MEM*. The number of executions of each application for every campaign varied from 2300 to 2504 and has been calculated using the method presented in [20], setting 99% as a target confidence level and 1% as the error margin.

The outputs of the campaigns are processed at the end of the simulation and then we create two outcomes. One which does not employ razor like correction for critical instructions and one with no protection at all. When razor-like technology is employed and a fault is injected in a critical instruction we consider the outcome as bitwise correct since the razor technology would have corrected it.

The experiments are categorized in the following categories:

Program Failure: The application failed to terminate normally, for example decoding a corrupted opcode could result to an *Illegal Instruction* violation.

Program Corruption: The application succeeds to terminate, however the result is not acceptable by the end user.

Correct: The application produces a result which is acceptable by the end user however it is not exactly the same as an error-less execution.

Bitwise exact: The execution resulted at the exact same output as an error-less execution.

Protected: The fault corrupted a critical instruction therefore it was corrected by the razor-like technology. The end result is the same as an error-less execution.

During simulation, if the fault corrupts a critical instruction the fault is injected nevertheless and a message is reported on the output of the simulator. At the end of the simulation campaing we process all the results. We create two outcomes: one which there is no support for protecting critical instruction and one with such support. In the second casei, with hardware protection mechanisms supported, when a fault is injected on a critical instruction we suppose that the hardware would correct it and therefore we categorize that experiment as Protected and the end result would be Bitwise exact.

In Table 1 we summarize the configuration settings of the simulator. For the fault injection campaigns the simulation is performed in cycle accurate operational mode. After the injection of the fault we simulate for 10000 cycles and then we switch to a fast but less accurate simulation mode. By waiting to switch to another CPU we ensure that the faults have manifested in the architectural components.

5.2 CASE STUDY I: SOBEL

5.2.1 *Algorithm Description*

In this section we will study the sobel filter. *Sobel* is a 2D filter used for edge detection in images. The sobel filter is based on applying a convolution filter in horizontal and vertical direction. The filter uses

Processor Parameters			
Bandwidth	8 Fetch	8 Issue	8 Commit
Queue Size	32-LoadQ	32-StoreQ	64-InstructionQ
Rename Reg	256-Float	256-Int	
Functional unit	6-IntALU	2-IntMultDiv	4-FPALU
	2-FP MultDiv	2-Read Write Ports	
Branch Predictor Parameters			
Tournament			
8192-Global Predictor		13 bit Global History	
2048-Local Predictor		2-bit Local History	
Branch Target Buffers		4096 Entries	
Size of Tag /RAS	16 tag size	16	
Cache Parameters			
IL1 config	32kb	64 Byte Block	2-way 2 cycle lat
DL1 config	64kb	64 Byte Block	2-way 2 cycle lat

Table 1.: Simulated X86 processor configuration for the experimental evaluation

two $3x3$ matrices which are convoluted with the original image. The matrices are presented in Equation 15 and the filter in Equation 16.

$$Sobel_{horiz} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 2 & 1 \end{bmatrix} Sobel_{vert} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (15)$$

$$Sobel = \sqrt{(Sobel_{vert} * A)^2 + (Sobel_{horiz} * A)^2} \quad (16)$$

Figure 11.: Vertical and Horizontal Operator applied in each pixel during the sobel filter.

5.2.2 *Relation between optimizations and critical instructions*

In Listing 5.1 we present a naive implementation of the sobel filter. Note that for the *sqrt* function we use a custom one. In each simulation we count the number of critical and non-critical instructions.

In both compiler optimized executions, presented in Figure 12, almost 100% of the total number of executed instructions should be protected. This is because in each iteration there is a clamping function (line 22-25). The *if* statement in line 22 checks the computed value of the current iteration. In our analysis values used by branch instructions tag as critical all instructions which explicitly or implicitly influence the outcome of the branch instruction. Since the entire

```
1  int convolution2D(int posy, int posx, const unsigned char *
       input, char operator[][3]) {
2    int i, j, res;
3    res = 0;
4    for (j = -1; j <= 1; j++) {
5      for (i = -1; i <= 1; i++) {
6        res += input[(posy + i)][posx + j] *
7        operator[i+1][j+1];
8      }
9    }
10   return(res);
11 }
12
13 double sobel(unsigned char *input, unsigned char *output,
       unsigned char *golden){
14   unsigned int temp;
15   for (j=1; j<SIZE-1; j+=1) {
16     for (i=1; i<SIZE-1; i+=1 ) {
17       temp = convolution2D(i, j, input, horiz_operator);
18       p = temp*temp;
19       temp = convolution2D(i, j, input, vert_operator);
20       p+=temp*temp;
21       res = (int)sqrt(p);
22       if (res > 255)
23         output[i][j] = 255;
24       else
25         output[i][j] = (unsigned char)res;
26     }
27   }
28 }
```

Listing 5.1: Source code of the sobel filter

iteration computes a value which is used in a branch instruction almost everything is protected. When executing with -*O*0 20% of the total instructions are identified as non-critical. In the non-optimized version the application continuesly load-store values to and from the stack. Although the branching instruction exists, just before performing the instruction the compiler loads from the stack the value of *res*. Our analysis does not track critical-ness within memory locations. Therefore the critical dependency is not tracked to the remaining instructions.

As a second step we implement the clamping function using binary operators, so that we remove entirely the branching instruction from line 22. Afterwards we simulate again the application twice, once with optimizations enabled and once with no-optimizations. When optimizations are enabled we observe that the number of non critical instructions dramatically increases, reaching up to 61% of the total number of instructions, for both optimized binaries. This is because the aforementioned control flow dependency is broken using a binary clamping function.

On the other hand the execution with no compiler optimizations has almost the same percentage of non-critical instructions. Since the compiler does not optimize the code, it uses *MEM-REG/REG-MEM*[1] instructions. All these instructions are guarded by the analysis. Moreover the compiler does not unroll the *convolution2D* function which is responsible for many control flow instructions, such instruction are always protected.

As a next optimization step we declare the $Sobel_{horiz}$ and $Sobel_{vert}$ as constant. Again we follow the same methodology. Both execution of the binaries produced with compiler optimizations (*-O3, -O3 –fast-math* decrease the number of critical instructions to less than 19%. This is because the compiler replaces all instructions which have taps of the sobel filter as an operand with operations using immediates. Again the number of critical instructions does not change for the non-optimized case since the compiler does not exploit constant arrays.

As a final step we use the clamping function that uses masking instead of branching instructions, perform loop unrolling of the convolution filter by hand and we lower memory references of $Sobel_{horiz}$, $Sobel_{vert}$ into constants. In other words, we enforce the non-optimized code to use instructions operating on immediates. We also remove possible branching inside the body of the main loop. The compiler optimized binares demonstrate a slight decrease in the number of non critical instructions. The produced number of assembly instructions increases by almost $10x$ making it highly impractical for a human to analyze the performed optimizations manually. On the other hand the *O0* version has a slight increase (2%) in non-critical instructions. This is because the instructions performing the iterations of the convolution are removed, since the code does not traverse a $3x3$ matrix but operates on an unrolled version with immediates. Moreover some memory-related operations are completely removed due to multiplications with 0.

5.2.3 *Fault Injection Validation*

Figure 13 presents the results after a fault injection campaign which was conducted on the best version of the sobel filter, in terms of the number of non-critical instructions. Except from faults introduced into the *Fetch* pipeline stage, the extended instruction set is always able to terminate normally. In the case of the fetch stage, 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.

1 Instructions which have as an operand a memory location.

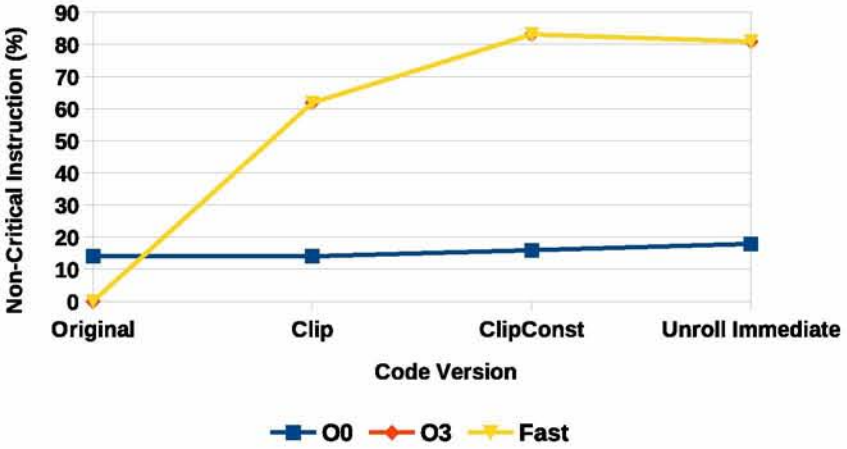


Figure 12.: Percentage of critical instructions for different versions of sobel.

Noticeably the decoding stage does not suffer from any failure when critical instruction 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 the new destination does contain a memory location, the location will be overwritten by the faulty instruction. Therefore an upcoming critical 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.

5.3 CASE STUDY: DCT

5.3.1 *Algorithm Description*

DCT is a module of video compression kernels, which transforms a block of image pixels to a block of frequency coefficients. Precisely we use the *DCT-II* which is more suitable for lossy comparisons since it compacts a lot of information in the first coefficients.

Equation 17 presents the *DCT-II* formula. In image/video compression the equation is applied twice, once horizontally and once vertically. During experimentation we perform the *2D DCT*, however we optimize the function which calculates the 1-DCT (it will be called twice).

$$X_k = \sqrt{2/N} s(k) \sum_{n=0}^{N-1} x_n cos\left[\frac{\pi}{N}(n+.5)k\right] \tag{17}$$

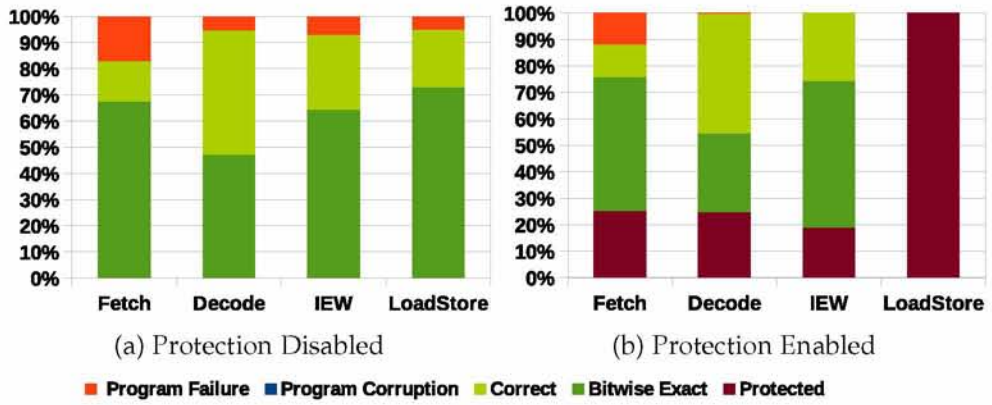


Figure 13.: Application behavior when fault injecting different architectural components, 13a- results when no critical instruction recognition is performed. 13b Results after performing instruction protection.

- X is the DCT output.
- x is the input.
- k is the index of the output coefficient being calculated, from 0 to $N-1$.
- N is the number of elements being transformed.
- s is a scaling function, $s(y) = 1$ except $s(0) = \sqrt{0.5}$

The DCT can be viewed as a matrix multiplication (Equation 18). The inputs and the outputs correspond to row-vectors. For simplicity we define the coefficients using Equations 19. The coefficients matrix is the NxN presented in 20

$$X = x \times M \tag{18}$$

$$c_0 = \sqrt{.5} \times \sqrt{2/N} \quad c_j = cos\,(\pi j/16) \times \sqrt{2/N} \tag{19}$$

$$M = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 \\ c_0 & c_3 & c_6 & c_9 & c_{12} & c_{15} & c_{18} & c_{21} \\ c_0 & c_5 & c_{10} & c_{15} & c_{20} & c_{25} & c_{30} & c_{35} \\ c_0 & c_7 & c_{14} & c_{21} & c_{28} & c_{35} & c_{42} & c_{49} \\ c_0 & c_9 & c_{18} & c_{27} & c_{36} & c_{45} & c_{54} & c_{63} \\ c_0 & c_{11} & c_{22} & c_{33} & c_{44} & c_{55} & c_{66} & c_{77} \\ c_0 & c_{13} & c_{26} & c_{39} & c_{52} & c_{65} & c_{78} & c_{91} \\ c_0 & c_{15} & c_{30} & c_{45} & c_{60} & c_{75} & c_{90} & c_{105} \end{bmatrix} \tag{20}$$

```
void dct_ii(const double x[8], double X[8]) {
 int n,k;
 for (k = 0; k < N; ++k) {
  double sum = 0.;
  double s = (k == 0) ? sqrt(.5) : 1.;
  for (n = 0; n < N; ++n) {
   sum += s * x[n] * COS[k][n] ;
  }
  X[k] = sum * 0.5;
}
```

Listing 5.2: Source code of a naive implementation of DCT-II

```
void dct_ii(const double x[8], double X[8]) {
 X[0] = a*c0 + b*c0 + c*c0 + d*c0 + e*c0 + f*c0 + g*c0 + h*c0;
 X[1] = a*c1 + b*c3 + c*c5 + d*c7 - e*c7 - f*c5 - g*c3 - h*c1;
 X[2] = a*c2 + b*c6 - c*c6 - d*c2 - e*c2 - f*c6 + g*c6 + h*c2;
 X[3] = a*c3 - b*c7 - c*c1 - d*c5 + e*c5 + f*c1 + g*c7 - h*c3;
 X[4] = a*c4 - b*c4 - c*c4 + d*c4 + e*c4 - f*c4 - g*c4 + h*c4;
 X[5] = a*c5 - b*c1 + c*c7 + d*c3 - e*c3 - f*c7 + g*c1 - h*c5;
 X[6] = a*c6 - b*c2 + c*c2 - d*c6 - e*c6 + f*c2 - g*c2 + h*c6;
 X[7] = a*c7 - b*c5 + c*c3 - d*c1 + e*c1 - f*c3 + g*c5 - h*c7;
}
```

Listing 5.3: Source code of an unrolled version of DCT-II

$$M = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 \\ c_0 & c_3 & c_6 & -c_7 & -c_4 & -c_1 & -c_2 & -c_5 \\ c_0 & c_5 & -c_6 & -c_1 & -c_4 & c_7 & c_2 & c_3 \\ c_0 & c_7 & -c_2 & -c_5 & c_4 & c_3 & -c_6 & -c_1 \\ c_0 & -c_7 & -c_2 & c_5 & c_4 & -c_3 & -c_6 & c_1 \\ c_0 & -c_5 & -c_6 & c_1 & -c_4 & -c_7 & c_2 & -c_3 \\ c_0 & -c_3 & c_6 & c_7 & -c_4 & c_1 & -c_2 & c_5 \\ c_0 & -c_1 & c_2 & -c_3 & c_4 & -c_5 & c_6 & -c_7 \end{bmatrix} \tag{21}$$

In 21 we exploit the circle symmetry to transform all angles to the first quadrant. Only eight unique coefficients are needed for an eight-point DCT.

5.3.2 *Relation between optimizations and critical instructions*

We use 4 implementations of the DCT-II algorithm which were obtained from [22].

Naive: In Listing 5.2 we present a naive implementation of the DCT-II function. The function essentially performs a matrix multiplication with a constant matrix. When compiler optimizations are enabled the inner *for* loop is unrolled, resulting to linear code with multiplications and additions.

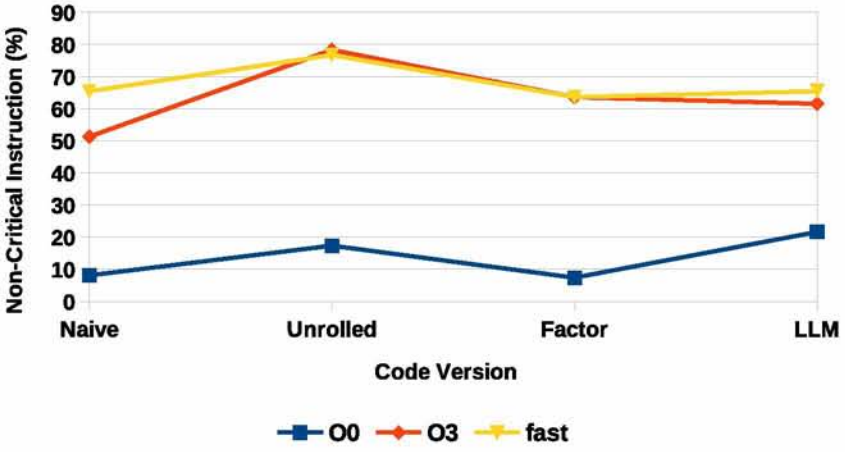


Figure 14.: Percentage of critical instructions for different versions of dct.

Unroll: In Listing 5.3 we present an implementation of the DCT-II where the matrix multiplication is fully unrolled by the programmer. Moreover, the programmer uses only 8 constants to perform the matrix multiplications. This version basically exploits the wisdom, presented in Equation 21

Factorization: In this implementation the user employs factorization to decrease the number of multiplications. Moreover, rotation-like operations can be used to reduce the number of multiplications even further. For example:

$$y_0 = ax_0 + bx_1 = (b-a)x_1 + a(x_0 + x_1) \quad (22)$$

$$y_1 = -bx_0 + ax_1 = -(a+b)x_0 + a(x_0 + x_1) \quad (23)$$

This method basically increases the liveness of register which use values frequently.

LLM This is an implementation of DCT-II called LLM [21], named after its authors. The *dct* is performed with only 11 multiplications and 49 additions.

In Figure 14 we depict the percentage of non-critical instructions when using different compilation flags for the different versions of the dct algorithm. In the same graph the relative speed up is presented on the right *y-axis*. The speed up is computed in with respect to the *Naive* version.

The naive implementation both compiler optimized versions, *O3*, *–fast-math*, perform more than 50% and 65% respectively non critical instructions. Both optimizations perform loop unrolling of the inner loop. The *–fast-math* version generates more but faster arithmetic

operations in comparison with the *O3* version. This is because the fast-math allows to relax rounding errors.

When the programmer enforces only 8 coefficients in the COS Matrix *(c0-c7)*, the register allocation coupled with smart instruction scheduling allows the execution to perform many arithmetic operations. To be more precise, the *dct_ii* function uses only 70 memory transfers to perform 327 pure-arithmetic operations. Therefore both the *-O3* and *-fast-math* versions demonstrate almost 80% of non critical instructions. On the non-optimized binary there is a small increase in the number of non-critical instructions. However, since the compiler does not perform smart register allocation and instruction scheduling the overall result is almost the same.

When the programmer factorizes computations the number of non-critical instructions decrease. Basically the number of load/store operations remains almost the same (68 for the compiler optimized version) but the number of arithmetic operations decreases (304). However, it produces 62% of non critical instructions.

Finally the *LLM* version tries to reduce the number of arithmetic operations which are non-critical. This results to a decreased number of executed non-critical instructions. On the other hand, the non-optimized execution demonstrates a slight increase in the number of non-critical instructions. This is due to the code *LLM* code structure, which basically loads a set of variables performs operations with that variables and in the end stores back the result. All the operation are pure arithmetic ones, therefore the ratio between those two decreases, resulting to an increased percentage of non-critical instructions.

5.3.3 *Fault Injection Validation*

In Figure 15 we illustrate the results after the fault injection campaign performed on *DCT-LLM* compiled with *O3 –fast-math*. Similarly to Sobel, we can clearly distinguish that during fetch stage the extra protection offered by the criticality information does not offer any extra resiliency. During the decoding stage there is a significant increase in terms of resilience, however the protection of instructions evidently does not guarantee lack of program failures. During execution stage the protection mechanism is correctly applied to only critical instruction leading to avoidance of *Program failures*. Finally all instructions moving data to/from memory are protected, therefore all faults injected in the memory stage are corrected by the hardware. In the memory stage when no protection is applied not a single experiment resulted to program failure. However, the criticality information instructs the hardware to protect all instructions in that state. Therefore for this benchamrk this approach is very conservative since during the memory stage not a single experiment lead to program failure.

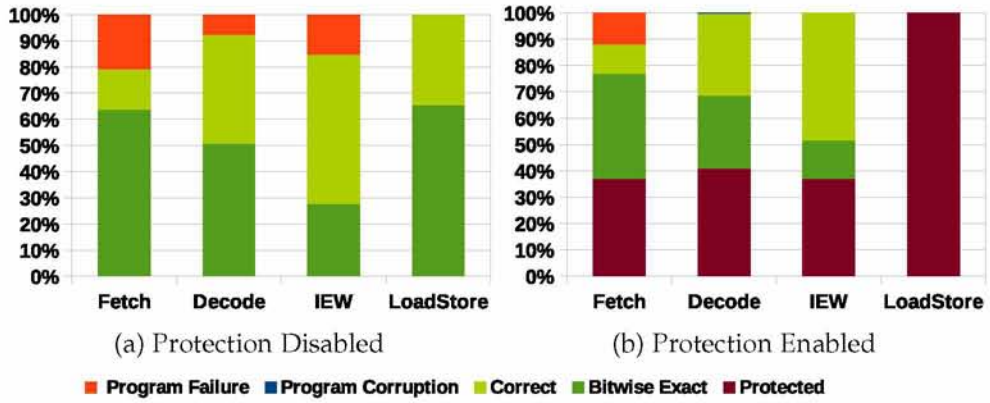


Figure 15.: Application behavior when fault injecting different architectural components: 15a - results when no critical instruction recognition is performed. 15b - Results after performing critical recognition and identification and protecting only critical instructions.

5.4 CASE STUDY: BLACKSCHOLES

5.4.1 *Algorithm Description*

The *Blackscholes* is a mathematical model, of the financial market. The model estimates the price of a financial option[2]. The blackscholes-formula estimates the price of an option over time [6].

The blackscholes equation for the call option as published in [6] is presented in Equation 24. The put option can be estimated by Equation 25

$$\begin{aligned} C(S,t) &= N(d_1)S - N(d_2)Ke^{-r(T-t)} \\ d_1 &= \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right] \\ d_2 &= \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{K}\right) + \left(r - \frac{\sigma^2}{2}\right)(T-t)\right] \\ &= d_1 - \sigma\sqrt{T-t} \end{aligned} \tag{24}$$

$$\begin{aligned} P(S,t) &= Ke^{-r(T-t)} - S + C(S,t) \\ &= N(-d_2)Ke^{-r(T-t)} - N(-d_1)S \end{aligned} \tag{25}$$

The $N(x)$ denotes the cumulative distribution function of the standard normal distribution. The $T-t$ is the the time to maturity. The S is the *spot price* of the current asset, K is the strike price, r is the risk

2 In finance, an option is a contract that allow the buyer (the owner) to buy or sell an asset at a specified price before a specific date

free rate and σ is the volatility of returns for the current product. The cumulative distribution function can be calculated by Equation 26

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{z^2}{2}} \tag{26}$$

5.4.2 *Relation between optimizations and critical instructions*

We use the implementation of BlackScholes from the *PARSEC 2.0* benchmark suite [3] The algorithm is very sensitive to small variations, therefore when compiled with the *–fast-math* option the produced results are incorrect.

In Listing 5.4 we present the implementation of the blackscholes formula. Listing 5.5 demonstrates an implementation of the *CNDF* function. We use the implementation of the *log* function from [36] which provides adjustable accuracy. In our experiments we used accuracy of 11 digits. The *sqrt* function is implemented using logical shifting additions/subtractions and multiplications. Finally for the *exp* function, we use the default *IEEE* double precision *exp* function. The function is implemented in the compiled source file, and no linking with the *libm* is performed. By doing so, we do not have any external dependencies to other functions.

To improve the non-critical instruction rate we create two more versions of the blackscholes implementation. The first version optimizes the control flow of the CNDF function. Both *if* statements in the CNDF function (lines 11,19) are removed by using unions, multiplications and additions. The second version removes the if statements from the *BlkSchlsEqEuroNoDiv* function using the same logic. The latest version produces linear code for the *BlkSchlsEqEuroNoDiv* function.

Using these three version we perform simulations to measure the dynamic instruction count. We do not use the –fast-math optimization flag since it produces incorrect results. The results are depicted in Figure 16. Both optimizations do not impact the number of critical instructions significantly. This is due to the implementation of the *exp* function. The function performs a series of bound checks to calculate the return value. As in *Sobel*, the values used for branching instruction result to identifying a significant percentage of instructions as critical due to ther impact to the branching instruction.

The different versions in fact operate worse than the original version. This is because, masking the branches to linear code requires more operations than a single branch. These operations are tagged as critical due to their impact in branching instructions which are inside the *exp* function. Since more instruction are identified as critical the ratio of non-critical instructions to critical ones decreases.

```
float BlkSchlsEqEuroNoDiv(
  float sptprice, float strike,
  float rate, float volatility,
  float time, char *otype, float timet )
{
 logValues = log (sptprice/strike);
 Xpower = volatility*volatility;
 Xpower = Xpower/2;
 XD1 = rate+ Xpower;
 XD1 = XD1*time;
 XD1 = XD1+logValues;

 sqrtTime = sqrt(time);
 XDen = volatility * sqrtTime;
 XD1 = XD1 * time;
 XD1 = XD1/XDen;

 Nd1 = CDNF (XD1);
 XD2 = XD1 -XDen;
 Nd1 = CDNF(XD2);

 FutureValueX = strike * ( exp( -(rate)*(time) ) );
 if (otype == "CALL") {
    OptionPrice = (sptprice * Nd1) - (FutureValueX * Nd2);
 } else {
    Negd1 = (1.0 - Nd1);
    Negd2 = (1.0 - Nd2);
    OptionPrice = (FutureValueX * Negd2) -
        (sptprice * Negd1);
 }
}
```

Listing 5.4: C-Like pseudo-code of the blackscholes formula

To remove the branches we used different approximations for the exponential function, either using the technique introduced in [32] or approximating exponential with a taylor series. All of the approximations where rejected because the execution resulted to erroneous results.

To remove the dependencies added by the *exp* function we use an unorhtodox solution. The body of the *exp* function is replaced with the single statement $return(0)$, which is obviously incorrect. However, we do not care about the correctness of the execution. We just need to remove all the dependencies added by the *exp* function. Afterwards in Fig 17 we compare the percentages of the non-critical instructions for the *BlkSchlsEqEuroNoDiv* function which has inlined the *CNDF* function for the different versions of the code. These percentages are obtained statically; we do not simulate the binaries.

As it can be clearly viewed by the graph, when the side-effects of the exponent function are removed both optimizations result to higher percentages of non-critical instructions. The remaining percentage of critical instructions are attributed to the implementation

```
static double CNDF(double d)
{
    const double        A1 = 0.31938153;
    const double        A2 = -0.356563782;
    const double        A3 = 1.781477937;
    const double        A4 = -1.821255978;
    const double        A5 = 1.330274429;
    const double RSQRT2PI =
    0.39894228040143267793994605993438;
    int sign = 1;
    double abs = d;
    if ( d < 0.0){
        abs = -d
        sign = -1
    }

    double K = 1.0 / (1.0 + 0.2316419 * abs);
    double cnd = RSQRT2PI * exp(- 0.5 * d * d) *
                 (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K *
    A5)))));
    if (sig > 0)
        cnd = 1.0 - cnd;
    return cnd;
}
```

Listing 5.5: The CNDF function

of the logarithm operation. Which uses a small hash table to calculate the logarithm and performs reads from the stack for the input variables.

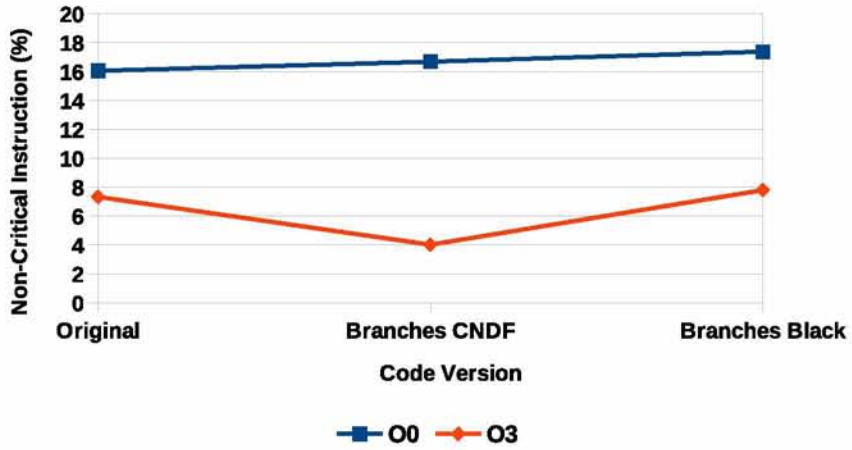
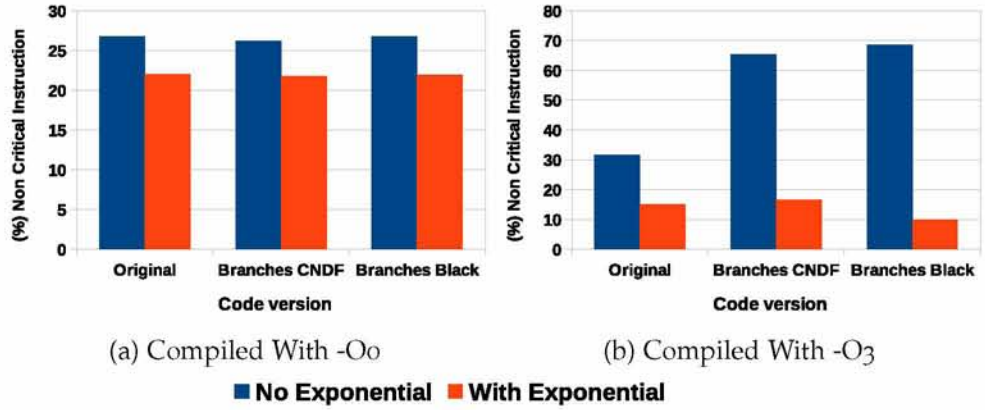
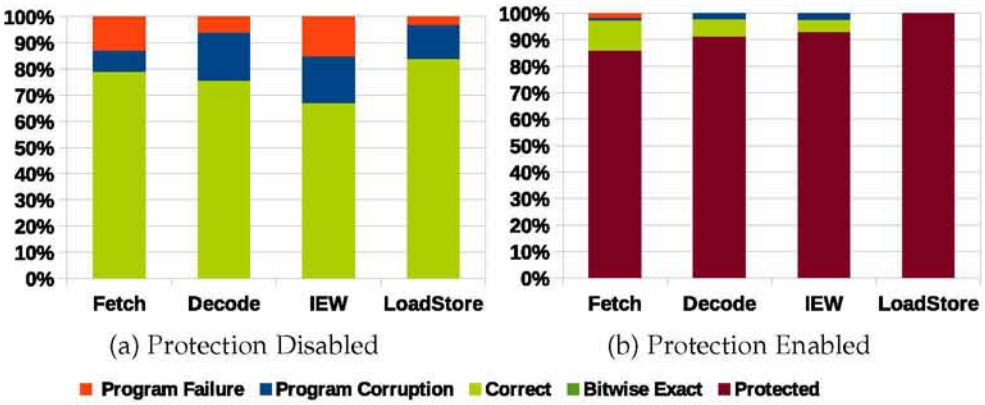


Figure 16.: Percentage of critical instructions for different versions of blackscholes.



(a) Compiled With -Oo (b) Compiled With -O3

Figure 17.: The percentage of statically non critical instructions when the exponential function is enabled/disabled. .



(a) Protection Disabled (b) Protection Enabled

Figure 18.: Application behavior when fault injecting different architectural components:18a- results when no critical instruction recognition is performed. 18b - Results after performing critical recognition and identification and protecting only critical instructions.

5.4.3 *Fault Injection Validation*

In Figure 18 we present the results of a the fault injection campaign performed on the blackscholes version. Blackscholes is the only application which demonstrates such a high degree of non-resiliency. Many experiments result to corruptions (unacceptable quality of output) (on average 15% of the total number of experiments). The blackscholes formula uses functions such as *exponent* and *logarithm*. Both these functions have regions of their definition space where they are very steep. Small deviations in the input results to large deviations in the output. It should be mentioned that all faults corrupted the output since there is not a single experiment which produce bitwise exact result.

5.5 INSTRUCTION SET CHARACTERIZATION

Complex instruction set computing (*CISC*) are those which a single instruction can execute several low level operations. For example a single *CISC* instruction may load from the memory, perform an arithmetic operation and store back the result to the memory. The main differentiating characteristic with RISC instruction sets is that *RISC* uses a uniform instruction length for almost all instructions and employs strictly separate load/store instructions. In the x86 instruction set most instructions have one or more operands that they operate on. The majority of instructions have as operands registers as well as memory locations.

From the critical analysis point of view the x86 instruction set architecture enforces the analysis to identify more instructions as critical. Any instruction with operands memory values is considered by the analysis as obvious critical. This is because, these instructions are translatedi, 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. Since the compiler analyzes *CISC* instructions and not the micro-operations that the *CISC* instructions are translated to. 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. Since the *RISC* instruction set is constructed by instructions which correspond to micro-operations.

6

RELATED WORK

We classify related work in fault tolerance into program analysis techniques, micro-architectural fault tolerance mechanisms, and transistor-level fault detection and correction mechanisms. Finally, we discuss collaborative approaches which utilize hardware, compiler and software-level techniques to provide reliable execution.

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.

6.1 PROGRAM ANALYSIS TECHNIQUES

In [17] 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. A static analysis is used by [35][33] to identify instructions affecting control flow. The first, [35], studies the application resiliency when using some sort of protection on such instructions whereas the second, [33], one replicates instructions to guarantee correct execution.

Although replication of instructions is considered as a fault tolerance method [26], 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 inherentely error tolerant in errors are analyzed in detail in [9]. Based on their observations the authors address common manufacturing defects. In [24][25] 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 [12] 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.

6.2 MICRO-ARCHITECTURAL FAULT TOLERANCE

AR-SMT[30] was one of the first works proposing redundant execution using Simultaneous Multithreading (SMT). The authors use SMT they avoided the extra overhead of executing application using a lock stepped processor as done traditionally. SRT [29] also uses SMT to provide redundant computation however, the authors also proposed the concept of spheres of replication. Such a sphere defines the set of components which are fault tolerant due to redundant execution whereas components outside it should be protected by other mechanisms, e.g ECC. In Slipstream [34] performance of the the redundant threads is improved by reducing the protected instructions of a thread. The redundant thread speculatively executes a subset of the total executed instructions.

6.3 LOW LEVEL FAULT TOLERANCE

Latest error detection mechanisms are based on razor flip-flops[15]. Razor flip-flops enable dynamic voltage scaling below nominal values to a processor, while ensuring its correct operation. The idea behind the integration of this Error Detection Sequential (EDS) is to tune the supply voltage of the chip while monitoring and correcting the detected failures triggered by the increased delay at lower voltages. As soon as a failure is detected, a correction mechanism takes place.

In [15] a simple method is based on clock gating, where, in case of a detected error, the entire pipeline is stalled by gating the clock during the next clock cycle. The stall period is used to recompute the correct value. If an error is due to happen in every clock cycle the CPU performance would drop to the half. Two correction mechanisms are proposed in [8]. In the first, after an error is detected the clock cycle is tuned to half the frequency and the errant instruction is re-executed. The second mechanism flushes the pipeline to avoid corruption of memory and the erroneous instruction is replicated and issued multiple times. If the instruction is replicated enough times the register will provide the correct value.

Finally the Bubble Razor[18] inherits features of razor techniques facilitating real-time detection and correction. It uses a novel bubble propagation algorithm applicable to any architecture. A timing

error is detected when data arriving at a latch varies after the latch open. This is detected using an error detection latch. Upon detecting a timing violation, the circuit automatically recovers by stalling the subsequent latch, giving it an additional clock cycle to process the data. Half of the additional clock cycle is used to compensate for the unexpectedly large delay from the previous latch and the other half accounts for the delay from the current latch to the subsequent one. Thus timing violations are corrected as long as the real delay of each half clock-cycle step never exceeds one clock cycle of time. Unlike other Razor schemes, one significant weakness of Bubble Razor is that it does not consider the impact of metastability in the error detecting logic.

6.4 COLLABORATIVE APPROACHES

In [10] a set of jobs is outlined. The system stack should adopt such jobs to detect and correct errors and variations. The tasks span from the circuit level to application level in order to create a robust system with minimal hardware improvements. In [16] ISA extensions are proposed for approximate computing. The underlying architecture is based on a dual voltage operation. On high voltage the system conducts precise operations whereas on low voltage there are margins of error.

7

CONCLUSION

In this MSc thesis we introduce a compiler analysis technique on the *x86* instruction set which identifies critical instructions. Such instructions are those which perform pointer arithmetic or controlo flow. The remaining instructions are non-critical. In case of operating on a subthreshold voltage, critical instrucions will be protected by Razor flip flops. Using an extended version of GemFI, a fault injection tool based on the Gem5 cycle accurate simulator we simulated a non-reliable execution enviroment. In such an enviroment 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.

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 MSc thesis 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.

A key direction for future work is to evaluate the overhead of protecting instructions in the hardware in terms of power and performance. Correcting all instructions might be costly whereas there might be room for relaxing the subset of protected instructions. For example control flow might be left unprotected by the compiler pass. This would reduce the cost of the hardware error detection - correction mechanism, however, the correction should take place at the software level by utilizing traditional error detection mechanisms such as, OS traps and checkpoining.

Appendices

A

LLVM ANALYSIS PASSES

The LLVM Pass Framework is one of the most useful parts of the LLVM system. Developers can implement their own optimizations and call them from the compiler using the *Manager* or they can use the *opt* tool to apply a pass on the intermediate LLVM representation. Essentially a pass corresponds to the various transformations and optimizations performed by the compiler.

The LLVM pass methodology is based on C++ inheritance and on virtual functions. The base class is called *Pass* and implements virtual methods for different functions that can be used by derived classes. Any user can implement his own pass by creating a class that derives from the following classes:

ImmutablePass:The ImmutablePass should not be used to implement an optimization pass. The class should not change the state of the compiled code but can provide information about the compiling process to the end user. This pass can provide information about the current target machine, or other static information that can affect upcoming transformations.

ModulePass :This pass corresponds to the most general super class a user can use. The pass applies optimizations to the entire application and refers to it as a single unit. The *manager* is able to execute a module pass if only the pass overrides the *runOnModule* method.

CallGraphSCCPass:The *CallGraphSCCPass* is used by passes that need to perform a backward directed optimization. Deriving from *CallGraphSCCPass* provides some mechanisms for building and traversing the *CallGraph*.

FunctionPass:The function pass is applied on the intermediate representation of a single function at a time. The transformations performed by this pass must be local and modify only the code of the specific function. *FunctionPasses* do not require to be executed on a particular order allowing the pass manager to schedule them efficiently. *FunctionPasses* may overload three virtual methods. All of these methods should return true if they modified the program, or false if they didn't.

LoopPass: A *LoopPass* is executed on each loop in the function, independently of all of the other loops in the function. LoopPass processes nested loops from the inner loop to the outer loop. Such passes can efficiently implement various polyhedral transformations. *LoopPasses* may overload three virtual methods. All these methods should return true if they modified the program, or false if they didn't.

RegionPass : The *Region* pass is similar to the loop pass, however it is performed on a single entry single exit region in a function. Nested regions are analyzed in the same way as nested loops. Programmers may overload three virtual methods of Region Pass to implement your own region pass. All these methods should return true if they modified the program, or false if they did not.

BasicBlockPass :The basic block passes offer the finest granularity and are applied on a single basic block at a time. They have many limitations: for example they are not allowed to modify the representative code of any other block except the one which is optimized at the moment.

B

LIFE OF AN LLVM INSTRUCTION

In this chapter we provide description of the various incarnations an *instruction* takes when it goes through LLVM's multiple compilation stages. Starting from a syntactic construct in the source language and up to the point it is encoded as binary machine code in an output object file.

```
1 int foo(int aa, int bb, int cc) {
2   int sum = aa + bb;
3   return sum / cc;
4 }
```

Listing B.1: This code is used as a reference code to study the various incarnations of an LLVM instruction

This exploration will start from the start of the compilation process therefore we will use as an example the code listed in B.1

Clang serves as the front-end of LLVM, it converts C, C++ and ObjC source into LLVM IR. Clang's main complexity comes from the ability to correctly parse and semantically analyze C++; the flow for a simple C-level operation is actually quite straightforward.

Clang's parser builds an Abstract Syntax Tree (AST) out of the input. The AST is the main "currency" in which various parts of Clang deal. For our division operation, a BinaryOperator node is created in the AST, carrying the *BO_div* operator kind. Clang's code generator emits a *div* LLVM IR instruction from the node, since this is a division of signed integral types. In LLVM IR, *sdiv* is a binary operator, which is a subclass of *Instruction* with the opcode *SDiv*. Like any other LLVM instruction, it can be processed by the LLVM analysis and transformation passes. Listing B.2 presents the IR representation of function *foo*.

After the IR incarnation LLVM moves to the code generation. At that point the LLVM tun time processes each function independently as a *Machine* function pass. The task of this pass is to "lower" the relatively high-level, target-independent LLVM IR into low-level, target-dependent *machine instructions (MachineInstr class)* . During the transformation of the Instructions to MachineInstr an LLVM pass traverses through the selection dag node structure.

Selection DAG nodes are created by the *SelectionDAGBuilder* class when requested by the *SelectionDAGISel*, which is the main base class

```
define i32 @foo(i32 %aa, i32 %bb, i32 %cc) #0 {
entry:
  %aa.addr = alloca i32, align 4
  %bb.addr = alloca i32, align 4
  %cc.addr = alloca i32, align 4
  %sum = alloca i32, align 4
  store i32 %aa, i32* %aa.addr, align 4
  store i32 %bb, i32* %bb.addr, align 4
  store i32 %cc, i32* %cc.addr, align 4
  %0 = load i32* %aa.addr, align 4
  %1 = load i32* %bb.addr, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* %sum, align 4
  %2 = load i32* %sum, align 4
  %3 = load i32* %cc.addr, align 4
  %div = sdiv i32 %2, %3
  ret i32 %div
}
```

Listing B.2: The function foo presented in the LLVM IR

for instruction selection. *SelectionDAGIsel* goes over all the IR instructions and calls the *SelectionDAGBuilder::visit* dispatcher on them. The method handling a *SDiv* instruction is *SelectionDAGBuilder::visitSDiv*. It requests a new SDNode from the DAG with the opcode ISD::SDIV, which becomes a node in the DAG. The initial DAG is still only partially target dependent. In LLVM nomenclature it's called *illegal* – the types it contains may not be directly supported by the target; the same is true for the operations it contains.

Before the SelectionDAG machinery actually emits machine instructions from DAG nodes, nodes undergo a few other transformations. The most important are the type and operation legalization steps. Such steps use target-specific hooks to convert all operations and types into ones that the target actually supports.

The division instruction (idiv for signed operands) of *x86* computes both the quotient and the remainder of the operation, and stores them in two separate registers. Since LLVM's instruction selection distinguishes between such operations (called ISD::SDIVREM) and division that only computes the quotient (ISD::SDIV), our DAG node will be "legalized" during the DAG legalization phase.

An important interface used by the code generator to convey target-specific information to the generally target-independent algorithms is *TargetLowering*. Targets implement the interface to describe how LLVM IR instructions should be lowered to legal SelectionDAG operations. The x86 implementation of this interface is *X86TargetLowering* . In the constructor operations are marked to be expanded. The legalization takes care of the correct translation and legalization. In our case, the ISD::SDIV is one of them should be expanded.

The next step in the code generation process is instruction selection. LLVM provides a generic table-based instruction selection mechanism that is auto-generated with the help of TableGen. Many target back-ends, however, choose to write custom code in their *SelectionDAGISel::Select* implementations to handle some instructions manually. Other instructions are then send to the auto-generated selector by calling *SelectCode*.

The *x86* back-end handles ISD::SDIVREM manually in order to take care of some special cases and optimizations. The DAG node created at this step is a *MachineSDNode*, a subclass of *SDNode* which holds the information required to construct an actual machine instruction, but still in DAG node form. At this point the actual X86 instruction opcode is selected – X86::IDIV32r in our case.

The code we have at this point is still represented as a DAG. But CPUs don't execute DAGs, they execute a linear sequence of instructions. The goal of the scheduling step is to linearize the DAG by assigning an order to its operations (nodes). The simplest approach would be to just sort the DAG topologically, but LLVM's code generator employs clever heuristics (such as register pressure reduction) to try and produce a schedule that would result in faster code. When creating a target description the developer can assign some hints to guide the scheduling procedure.

Finally, the scheduler emits a list of instructions into a *MachineBasicBlock*, using *InstrEmitter::EmitMachineNode* to translate from *SDNode*. The instructions here take the *MachineInstr* form (MI form), at this point the DAG is destroyed.

Apart from some well-defined exceptions, the code generated from the instruction selector is in SSA form. In particular, it assumes it has an infinite set of "virtual" registers to act on. This, of course, isn't true. Therefore, the next step of the code generator is to invoke a "register allocator". Virtual registers are replaced by physical registers, from the target's register bank.

Some instructions in some architectures require fixed registers. A good example is our division instruction in x86, which requires the inputs to be in the EDX and EAX registers. The instruction selector knows about these restrictions, such assignments are done by *X86DAGToDAGISel::Select*. The register allocator takes care of all the non-fixed registers. There are a few more optimization (and pseudo-instruction expansion) steps that happen on machine instructions at this point.

At this point our original C function translated to MI form – a *MachineFunction* filled with instruction objects (*MachineInstr*). This is the point at which the code generator has finished its job . In LLVM, there are two ways to emit the binaries. One is the (legacy) JIT which emits executable, ready-to-run code directly into memory. The other

is *Machine Code (MC)* , which is an ambitious object-file-and-assembly framework.

When LLVM is used as a static compiler (as part of clang, for instance), MIs are passed down to the MC layer which handles the object-file emission (it can also emit textual assembly files). *LLVMTargetMachine::addPassesToEmitFile* is responsible for defining the sequence of actions required to emit an object file. The actual translation is done in the *EmitInstruction* of the *AsmPrinter* interface. For *x86*, this method is implemented by *X86AsmPrinter::EmitInstruction*, which delegates the work to the *X86MCInstLower* class.

The object file (or assembly code) emission is done by implementing the *MCStreamer* interface. Object files are emitted by *MCObjectStreamer*, which is further sub-classed according to the actual object file format. For example, ELF emission is implemented in *MCELFStreamer*.

C

LLVM OBJECT FILE GENERATION

The LLVM *machine code (MC)* is a subproject of the LLVM to resolve numerous binary creation obstacles. The binary creation layer can be divided into two main categories, objects which operate on instructions and the components which operate on other entities, for example labels, data etc. The instructions are represented by the *MCInst* C++ class with operands such registers, immediate and the other entities are encapsulated in series of different classes such as the *MCymbols, MCSection and MCExpr*.

The MC classes are at the very end of the LLVM system and depend on only on the support libraries. The main reason for that is to create an independent binary and assembly creation, since an assembler does not need an register allocator. On the following section a brief description of the major components of the MC project.

Instruction Printer The instruction printer is a very simple target-specific components that implements a simple API, given a single *MCInstr* it formats and emits a textual representation of the instruction to a *raw_owstream*. Different targets can implement multiple *MCInstrPrinters*, for example the *x86* back end includes an AT&T and an Intel syntax instruction printer. Information about section-directives are completely hidden from the Instruction printer, so that they are independent from the object file format.

Instruction Encoder The instruction encoder is another target-specific component which transforms an MCInst into a series of bytes and a list of relocation's, implementing the MCCodeEmitter API. The API is quite general, allowing any bytes generated to be written to a raw_ostream. Because the X86 instruction encoding is very complex the back-end implements this interface with custom C++ code that is driven from data encoded in the .td files. This is the only realistic way to handle all the prefix bytes, REX bytes etc, and is derived from the old JIT encoder for x86.

Assembly Parser The assembly parser handles all the directives and other gunk that is in an .s file that is not an instruction (which may be generic or may be object-file specific). This is the thing

that knows what .word, .global etc are, and it uses the instruction parser to handle instructions. The input to the Assembly parser is a *MemoryBuffer* object (which contains the input file) and the assembly parser invokes actions of an *MCStreamer* interface for each thing it does.

Assembler Backend The assembler back end is included as an implementation of the MCStreamer API, along with the MCAsmStreamer text assembly code emitter) which implements all the binary creation. For example, the assembler has to do "relaxation" which is the process that handles things like branch shortening, situations where the size of one instruction depends on how far apart these two labels are. It lays out fragments into sections, resolves instructions with symbolic operands down to immediate and passes this information off to object-file specific code that writes out for example an ELF or Machine object file (.o).

Compiler Integration The final piece of the assembler is integrating all the MC objects into the compiler. In practice this meant making the compiler talk directly to the MCStreamer API to emit directives and instructions instead of emitting a text file.

BIBLIOGRAPHY

[1] J. Ansel, K. Arya, and G. Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[2] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[3] 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.

[4] 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.

[5] D. Blaauw, S. Kalaiselvan, K. Lai, W. Ma, S. Pant, C. Tokunaga, S. Das, and D. M. Bull. Razor II: in situ error detection and correction for PVT and SER tolerance. In *2008 IEEE International Solid-State Circuits Conference, ISSCC 2008, Digest of Technical Papers, San Francisco, CA, USA, February 3-7, 2008*, pages 400–401, 2008.

[6] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973.

[7] K. A. Bowman, J. W. Tschanz, N. S. Kim, J. C. Lee, C. B. Wilkerson, S. Lu, T. Karnik, and V. K. De. Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance. *Solid-State Circuits, IEEE Journal of*, 44(1):49–63, 2009.

[8] K. A. Bowman, J. W. Tschanz, S.-L. Lu, P. A. Aseron, M. M. Khellah, A. Raychowdhury, B. M. Geuskens, C. Tokunaga, C. B. Wilkerson, T. Karnik, et al. A 45 nm resilient microprocessor core for dynamic variation tolerance. *Solid-State Circuits, IEEE Journal of*, 46(1):194–208, 2011.

[9] 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.

[10] N. P. Carter, H. Naeimi, and D. S. Gardner. Design techniques for cross-layer resilience. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1023–1028, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[11] B. Colwell. We may need a new box. *Computer*, 37(3):40–41, 2004.

[12] 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.

[13] P. Enberg. A short introduction to the x86 instruction set encoding.

[14] 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, 2003.

[15] 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.

[16] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. *SIGPLAN Not.*, 47(4):301–312, Mar. 2012.

[17] 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.

[18] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D. Harris, D. Blaauw, and D. Sylvester. Bubble razor: An architecture-independent approach to timing-error detection and correction. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 488–490. IEEE, 2012.

[19] 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, 2004.

[20] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: quantified error and confidence. In *Proc. of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, 2009.

[21] C. Loeffler, A. Ligtenberg, and G. S. Moschytz. Practical fast 1-d dct algorithms with 11 multiplications. In *Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on*, pages 988–991. IEEE, 1989.

[22] E. Mikulic. Discrete cosine transform.

[23] 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 *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 622–629. IEEE, 2014.

[24] 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, 2007. IEEE Press.

[25] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer. Dynamic derivation of application-specific error detectors and their implementation in hardware. In *Dependable Computing Conference, 2006. EDCC'06. Sixth European*, pages 97–108, Piscataway, NJ, USA, 2006. IEEE Press.

[26] 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, 2012. IEEE Press.

[27] 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.

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

[29] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. *SIGARCH Comput. Archit. News*, 28(2):25–36, May 2000.

[30] E. Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 84–91. IEEE, IEEE, 1999.

[31] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[32] N. N. Schraudolph. A fast, compact approximation of the exponential function. *Neural Computation*, 11(4):853–862, 1999.

[33] 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.

[34] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: Improving both performance and fault tolerance. *SIGARCH Comput. Archit. News*, 28(5):257–268, Nov. 2000.

[35] 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.

[36] O. Vinyals, G. Friedland, and N. Mirghafori. Revisiting a basic function on current cpus: a fast logarithm implementation with adjustable accuracy. *International Computer Science Institute*, 2007.