Modelling Processor Reliability using LLVM Compiler Fault Injection

Y. Nezzari, C. P. Bridges
Surrey Space Centre
University of Surrey
Guildford, Surrey, United Kingdom
+44 (0)1483 689137
{y.nezzari, c.p.bridges}@surrey.ac.uk

Abstract — **The use of commercial of the shelf (COTS) processors is increasingly attractive for the space domain, especially with emerging high demand applications in Earth observation and communications. An order of magnitude improvement in on-board processing capability with less size, mass, and power is possible, however, COTS parts still lag in terms of reliability in the space environment. Costly protection techniques to ensure resilience to single event effects (SEEs) is required. Whilst current software reliability techniques are only capable of detecting errors, and performing partial recovery, our research offers a step change for both error detection and recovery without degradation in fault coverage. This targets modern multicore processors.**

We have previously shown how to create additional passes in the compiler's intermediate representation layer to automatically add differing protection codes at compile-time using the LLVM compiler framework. LLVM is supported by multiple processing architectures, and multiple high level languages – meaning it can be ported to not just space applications, but aerospace, defence, medical, and automotive.

In this paper a new LLVM fault injection tool is presented to validate and measure software protection methods – either statically at compile time or dynamically at runtime for multiple errors such as silent data corruption (SDC), control/flow errors, and crashes. We use our tool to inject faults into unprotected and protected codes and make quantitative comparisons of the errors and associated statistical confidence. Our protection method shows high coverage, up to 100% for some benchmarks, and does not assume that the memory system is protected via typical TMR hardware approaches. This means that we protect all memory instructions that use read and write. Another reason for the high coverage is the inclusion of multiple data and instruction types (i32, i32*, i1, i8, i8*, i64, float & double, float & double pointers). This research has been implemented in two processing architectures; Intel core i5-3470 with 3.2 GHz frequency and a Raspberry Pi 3. On the 1st processing platform the overhead was less than 15% and on the 2nd platform the overhead was less than 17%.

TABLE OF CONTENTS

1. INTRODUCTION

In recent decades, system designers have access to increasingly low-cost and high-performance processors. This is due to dramatic technology scaling, with smaller and faster transistors that operate at low threshold voltages yet are more sensitivity to noise margins. Unluckily, this issue has led to increasing terrestrial soft errors and, in mission-critical domains, lower device reliability [1, 2, 3]. As transistors become increasingly prone to environmental factors, high-reliability components should also be extended to processors used in mainstream our computing.

Soft errors are transient faults caused by the external environment, such as cosmic radiation particles striking an Integrated Circuit (IC) resulting in erroneous program execution. This could disrupt the processor registers or the stored data in the RAM and caches. Soft errors also include intermittent faults caused by overstressed operating conditions (e.g. component wear-out, component overload) [4]. Soft errors do not cause permanent damage. In the space domain, Radiation Hardened (RH) and Radiation Hardened By Design (RHBD) processors are used to mitigate against radiation effects. However these processors only fulfil some of this domain's requirements and fail in the others due to their lag in performance compared to COTS. This gap is estimated to be 5-10 years, in addition to higher power consumption and higher costs. All the previous constraints steer the space domain into considering the use of COTS. For the high-end high-availability market with no budgetary restrictions, system designers usually depend on redundancy [5, 6] to tackle soft errors. An example is the use of hard Error Correction Codes (ECC) that could be costly in terms of both performance [7] and power [8] which can be

978-1-5386-2014-4/18/$31.00 ©2018 Crown

impractical for desktop and embedded system domains. The use of COTS can be highly beneficial because of their high performance, low cost and low power dissipation. But their lack of reliability in radiation environments causes bit flips in memory (RAM and caches) and CPU registers which can be mitigated by using software protection techniques.

In this paper, pure-software protection techniques have been implemented that increase the reliability of the COTS multicore processing architectures. Their reliability improvements have been tested by the mean of software fault injection. Pure-software protection techniques can be complementary alternatives for when hard protection techniques cannot be afforded [9].

This research presents a new software protection technique based on extending the LLVM compiler framework, in order to automatically add protection code to the Intermediate Representation (IR) of the code intended to protect. The added protection code includes redundant instructions and a voter function to decide at run-time which instruction is the correct one. Using instructions redundancy will allow for the mitigation of more than one bit errors (could tolerate Single event upsets "SEUs" and Multiple bit upsets "MBUs" in some cases). Our work contributes to the state of the art with the following points;

- Low overhead utilising independently interleaved instructions in the CPU pipeline to add redundancy and an acceptable overhead,
- High coverage, this will be shown with three benchmark applications, where the coverage reached up to 100%,
- Our work includes the protection of multiple data and instruction types (i32, i32*, i1, i8, i8*, i64, float & double, float pointers & double pointers) [10],
- Both the CPU and the memory Read/Write instructions are protected.

The rest of the paper is organised as follows: in the second section, the solutions that have been applied to overcome the SEEs obstacle will be introduced, focusing on the soft-based protection techniques. In the third section, our problem approach is introduced in detecting instructions dependency and predicting the appropriate protection technique. In the fourth section, the fault injection experiment will be explained and how fault coverage will be deduced using our software protection techniques. In the fifth section, the injection experiment will be evaluated, and we include the measured error coverage improvements from using our protection approach. We finally conclude which of the soft-protection techniques are appropriate for our applications.

2. FAULT TOLERANCE

Fault-tolerance can either mask or detect and recover errors [4]. A Fault-tolerant system's external state should not be affected by faults, however, its components can fail. Errors must be detected and corrected before their propagation to the external state. The detection can be achieved by concurrent error detection techniques. The recovery can be done by schemes like re-execution, rollback recovery, roll forward recovery and check pointing [4].

2.1 FAULT TOLERANCE BY REDUNDANCY

Redundancy is used for error detection and recovery, this can be achieved by hardware; extra processors and memories, or software; entire programs or parts of it.

The N-modular redundancy (NMR) is a well-known technique used for error detection and/or recovery, where N marks the number of duplicated similar processing components, with similar data. Dual modular redundancy (DMR) is the minimal NMR technique with two identical elements, the original and a redundant one. DMR can only detect errors, which can be achieved by comparing the outcome of its two elements [11]. Triple modular redundancy (TMR) adds one replica to DMR, allowing it to detect and recover faults using a voter, instead of a comparator, allowing it to decide using majority. The extra redundant element makes TMR more reliable than the DMR, because of its ability to identify the wrong element and continue execution using the correct one, this is called fault masking. TMR's reliability comes with an extra cost in terms of performance compared to DMR. Redundancy can also be classified as;

- Spatial, meaning the addition of extra hardware parts to the system, allowing the same functions on distinct components
- Temporal, this is the NMR scheme implemented using the same hardware, implying that identical operation performed N times sequentially on the same hardware in different intervals of time
- Information, by either replicating the data and CPU registers, or by adding extra check bits to the original data using error detection codes (EDC) or error-correction codes (ECC). In ECC, when writing data to memory, it will be encoded and stored alongside the check bits. When reading from memory the data will be decoded using the check bits in order to detect and correct any error that has occurred

2.2 SOFTWARE-BASED REDUNDANCY FAULT TOLERANCE APPROACHES

A. Process-Level Replication (PLR):

PLR is implemented at kernel level. This technique runs the same application on three separate tiles (cores). The

outcomes of the three tiles are compared with TMR. Opening files was done using the shared memory that is equipped with bookkeeping. The results for this technique [12] using image compression of size 904K show a total overhead of 2,187,742,009 cycles including both the compression and PLR overhead.

The implementation is achieved in three phases [12], the two main phases, init and interposition only cause minor overheads however the last phase, fini, could add large overheads to the system. In addition, this technique does not consider the case where the shared memory is compromised by the SEEs. This study has no error coverage information (the ability to detect and correct errors has not been included).

B. Thread-level Replication (TLR):

The TLR is a software based fault tolerance technique [12] originating from the N-version programming [13] with thread replication. The user function is replicated on three threads running on separate tiles. The execution is done in parallel. TLR compares the outputs from the three threads using TMR. The redundancy of Maestro architecture [14] is used in this technique, where threads replace processes.

Three copies of the original function will be created using pthread library on different tiles. The outcomes of the threads will be compared using TMR. TLR has two functions: The main implemented by the threadReplicate function and the insertErrors for evaluation.

The overhead caused by TLR is application dependent [12]. On an empty function the total overhead was 12,021,270 Cycles. TLR's shared memory is still unprotected.

C. Instructions Level Replication:

DRIFT (Decoupled Compiler-based Instruction-level Fault-Tolerance) is a compiler error detection technique based on replicating instructions of the program and introducing checks. This scheme is aiming to minimize the error detection overhead and enhancing the system's performance without effecting fault coverage. DRIFT attains this by decoupling the execution of the code (original and replicated), and introducing checks [15]. This technique generates an overage overhead of 29%, using Mediabench II video [16] and SPEC CPU2000 [17]. The error detection only covers the CPU, the memory is always vulnerable to SEEs.

In the Composite Data Type Protection Algorithm (CDTP) each variable is protected by extra error coding scheme. Every read and write operation carried on the protected object is replaced by a set of operations responsible for checking its correctness, in case of an error, the correct value of the variable is obtained from the redundant information stored together with original data [18].
Automatic enrolment of the CDTP algorithm was implemented in the cc1 compiler as an independent stage of the compilation process. Protection techniques are applied at the beginning of the source code optimization, directly after the transformation of the protected program to GIMPLE internal representation [18]. The overhead depends on the benchmark tested and the algorithm used for protection. Using Hamming algorithm generates an average overhead of 86%, extended Golay algorithm 146%, full iterated coding scheme 116%, and selective iterated coding scheme 117%. This technique only protects memory of the processor architecture, the CPU is still vulnerable to SEEs.

The SWIFT-R Transformation technique creates two redundant replicas of the original instruction and then uses TMR for error detection and recovery. Having three copies means that if a fault corrupts any one version's computation, the two other versions will still have the correct computation.

However this scheme doesn't protect memory instructions, including the store, load, alloca, GetElementPointer, and the branch instructions. The random nature of soft errors does not exclude any instruction type, meaning all instructions are susceptible to soft errors originating from SEEs [19]. This technique adds up to 198% overhead of the execution time.

Further Techniques include Shoestring [20], Fault Tolerance Software Checking [21], Error Detection by Duplicated Instructions (EDDI) [9] and Software Implemented Fault Tolerance (SWIFT) [22]. These techniques are implemented by modifying compilers; namely LLVM, GCC, and OpenIMPACT [23] respectively. The modifications include duplicating instructions and inserting compare instructions where needed. While all the mentioned techniques in this Section are only for CPU protection, EDDI is capable of covering both the CPU and memory from the SEE, but it only covers the MIPS architecture and it has no recovery scheme.
DAFT [24] and SRMT [25] techniques use LLVM, Intel production compiler and ICC 9.0 respectively to automatically generate two redundant threads for an application, and compare their outcome for error detection. DAFT shows a considerable improvement from SRMT. Both of these techniques are for CPU protection, the memory system is still vulnerable to the SEEs soft errors.

CASTED [26] system is implemented by modifying the back-end passes in GCC-4.5.0 [27] compiler framework. This has been implemented with two passes, one for error detection and the other for error correction. Like most of the techniques in the literature, this only covers the CPU, memory has no coverage.

3. Automatic Compiler Error Detection & Recovery (ACEDR)

The LLVM compiler is chosen as the baseline source of this research, where we created passes in order to automatically add protection code to the intermediate representation of any code of choice. The user of our software protection

method does not have to write a single line of protection code, all he has to do is to compile his unprotected code via our passes, and the code will be protected. The passes include an analysis and transformation pass. The analysis pass will go through the code line by line in order to determine all types of instructions, and provides statistical information about them. The transformation pass will use the information provided by the analysis to call the appropriate protection technique [28]. In our previous work [28] we studied the overhead of applying different protection techniques. We started by studying the implementation of Hamming code for our automatic compiler error detection and recovery, with the ability to detect two bits errors and recover one bit. We also implemented the BCH code, with ability to detect and recover multiple bits. We have eliminated the use of both (Hamming and BCH) in our automatic compiler error detection and recovery, because of their large overhead. In addition to that these codes are more suitable to protect memory instructions, where there is access to memory with read and write operations. For CPU instructions like arithmetic and logic operations, it is more suitable to use redundancy method NMR to detect and/or recover errors.

Automatic compiler implemented TMR [28] is showing great potential, because of its low overhead and ability to detect and recover any single bit error, and in some cases it was able to detect and recover more than one bit error, provided that the multiple errors happen in the same word, or two different words in two different TMR-ed words. If two or more errors take place in two different words of the same TMR-ed words, it would be possible to detect the errors, but impossible to recover, since the TMR does not know which one is the right word.

3.1 ADDING ACEDR INSTRUCTIONS IN IR

The creation of redundant instructions is achieved using LLVM Compiler passes, allowing automatic addition of protection code to the code intended to be protected in its IR format. Our previous work [28] has been extended to include multiple data types: i32, i32*, i1, i8, i8*, i64, float & double, float & double pointers. We predict, this allows for higher coverage compared to the state of the art.

ACEDR-TMR will add two redundant instructions to the original one, and then calls a voter function in order to decide the correct outcome, amongst the three instructions, the replicated and the original. In this work we found that both memory instructions (alloca, load, store and GetElementpointer) and CPU instructions (Arithmetic and logic Operators…etc) require protection, because our work does not assume that the memory system is protected with any type of hard-ECC protection techniques, allowing as to extend our implementation to more processing architectures.

A. Memory Instructions Protection:

Memory and CPU instruction types were separated from each other, enabling the protection of both types. The memory instructions are treated differently because there is a dependency when writing and reading from a memory location. Our software protection techniques detects this dependency and adds the appropriate protection code accordingly. The implementation was done by changing the intermediate representation of the code intended to protect, instead of having a single memory location, two redundant locations were added. In the following code `%i` is the original **`alloca`** instruction and `%pwtc21` and `%pwtcx32` are the redundant ones we created Fig 1.

```
%i = alloca i32, align 4 // Original Code

%pwtc21 = alloca i32 // ACEDR Code
%pwtcx32 = alloca i32
%i = alloca i32, align 4
```

Figure 1. Protecting the "alloca" memory instruction

Redundant writes to the memory locations created previously will be added**, `store i32 10, i32* %i`** is the original store, **`store i32 10, i32* %pwtc21`** and **`store i32 10, i32* %pwtcx32`** are the newly created stores Fig 2.

```
store i32 10, i32* %i, align 4
// Original Code

store i32 10, i32* %pwtc21 // ACEDR Code
store i32 10, i32* %pwtcx32
store i32 10, i32* %i, align 4
```

Figure 2. Protecting the "store" memory instruction

Every time a load or (read) instruction is detected (the original read is **`%2 = load i32* %i, align 4`**) from a memory location, redundant reads are added (the redundant reads are **`%0 = load i32* %pwtc21`** and **`%1 = load i32* %pwtcx32`**) and the outcome is compared using a voter **`%func = call i32 @vote(i32 %2, i32 %1, i32 %0)`**, resulting that the correct memory location only will be the one with the final read Fig 3.

```
%2 = load i32* %i, align 4
// Original Code

%0 = load i32* %pwtc21 // ACEDR Code
%1 = load i32* %pwtcx32
%2 = load i32* %i, align 4
%func = call i32 @vote(i32 %2, i32 %1, i32
%0)
```

Figure 3. Protecting the "load" memory instruction

B. CPU Instructions Protection:

The memory instructions have already been protected, meaning they could be used by the CPU registers safely. However, the CPU registers are not immune from the SEEs yet, meaning they require protection. Using redundancy each CPU instruction is replicated, and the outcome will be obtained after calling the voting function, as shown in the next code Fig 4;

```
%add1 = add nsw i32 %11, 1
// Original Code

%add2 = add nsw i32 %11, 1 // ACEDR Code
%add3 = add nsw i32 %11, 1
%add1 = add nsw i32 %11, 1
call i32 @vote(i32 % add1, i32 % add2, i32
% add3)
```

Figure 4 Protecting the "add" CPU instruction

4. Error Injection

We provide an original LLVM fault injection tool to validate and measure our software protection methods – either statically at compile time or dynamically at runtime. Our injector can induce multiple error types such as silent data corruption (SDC), control/flow errors, hangs and crashes. We use our tool to inject faults into unprotected and protected codes, and make quantitative comparisons of the error and associated confidence on the presented interval.

Fig 5 shows how the error injection works, on the left side, the protected code will be injected and traced, meaning the outcome of its instructions will be logged in files, to compare them with the Golden files. The Golden files contain the correct outcome of each instruction of the code after tracing it (no injection is done with the Golden files outputs). The tracing is done on the code in its intermediate representation that can be obtained using the LLVM compiler. A python script adds instructions to show the outcome of every instruction of the traced code.

On the right side of Fig 5, the injection and tracing of the protected code is done. First the code will be protected using our developed software protection techniques, by running the protection pass on the code in its intermediate representation. In the second part, the code will be randomly injected in one of its instructions, causing it to fail with a certain failure rate, depending on the nature of the injected code. At last the code protected and injected will be traced in order to show the outcome of each of its instructions, then log them into files for comparison with the golden outputs obtained previously. The comparison will show the number and the types of errors obtained. The types of errors include: SDC, control/flow errors, crashes and hangs. The last comparison is between the error rate of the unprotected injected code, and the protected injected code. This step will determine how much coverage our protection has added.

In order to inject the intermediate representation of code, instructions have been divided to their different types, and every type is injected differently, and this is achieved by calling different function types specific to every instruction type, including (i32, i32*, i1, i8, i8*, i64, float & double, float & double pointers).

Limitations of our fault injector are that:

- We cannot inject the void type,
- Branches are void. In order to inject these, the decision instruction (ex "cmp") must be injected, and
- Return instructions are void. In order to inject these, the "load" instruction before must be injected.

5. Results & Evaluation

We have chosen to evaluate our method's error detection and recovery ability, our work targets 6 well-known benchmarks: Fib, Qsort, SolveCubic, Rad2Deg, Deg2Rad, UQsort from MediaBench [29] that are implemented into three: Fib, Qsort and MATH. We have chosen this variety of benchmarks in order to have different expected instruction types and coverages, and to evaluate the different error rates resulting from injecting the different types.

All instructions for each benchmark were injected, divided by their type. At the start, we injected the all the instructions of the unprotected code and checked how many injections resulted in errors. This 1st injection experiment was conducted without adding any protection techniques. The results of injection are shown in Fig 6-8.

5.1 Injecting Unprotected Code

In this experiment, we predict high error rates at more than or equal to 50 % since no protection is added to the binaries.

A. Injecting "load" instructions:

The "load" instruction is used to read from memory. In the 1st benchmark (FIBO) 71.43% total error rate has been recorded, the highest rate was the SDC with 50%, followed by 21.43% of control. No crashes have been recorded.

In the second benchmark (QSRT) we have recorded all types of errors, with a total rate of 47.4%, crashes are the major error causes with 24.07%, followed by the 17.6% SDC and 5.56% Control.

The 3rd benchmark (MATH) has shown the highest error rate when injecting the "load" instruction, with 88.62 % total error rate. The main cause of errors was the SDC with 51.74%, followed by 36.88% control, however no crashes were recorded.

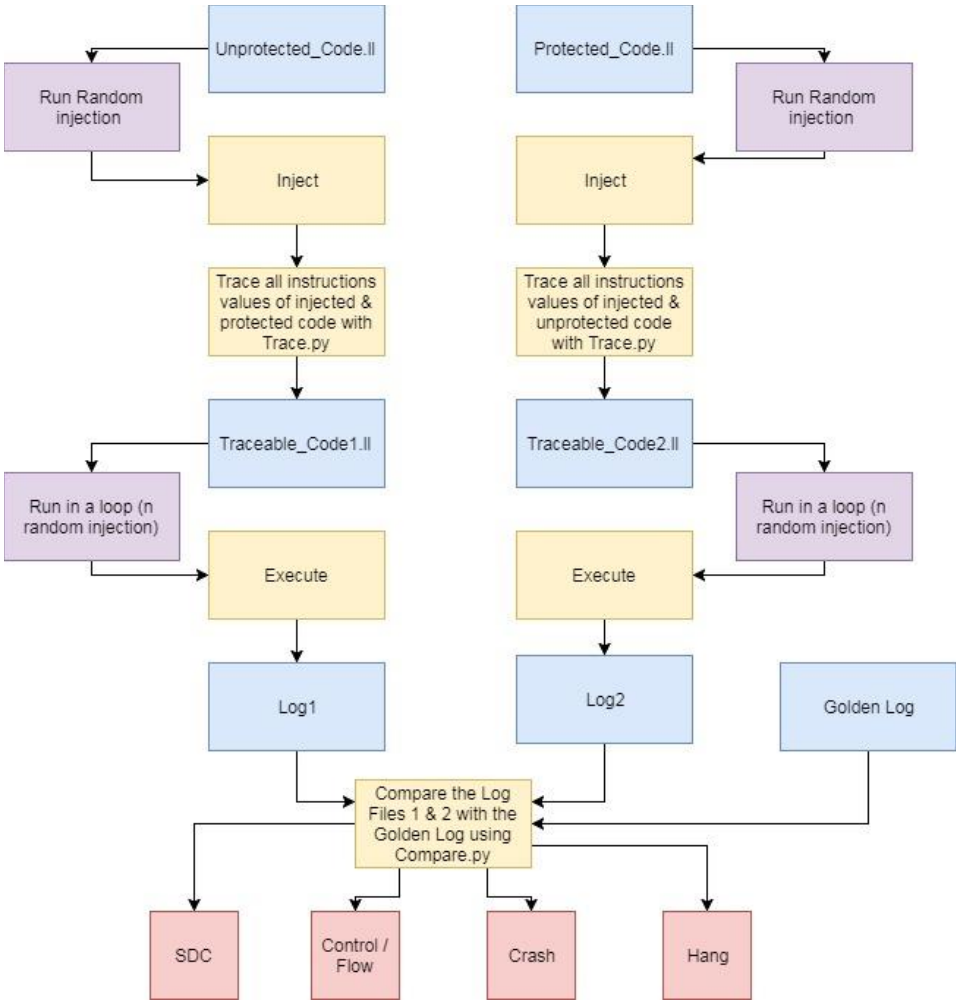

Figure 5. Flowchart of the Error Injection Process

B. Injecting "store" instructions:

The "store" instruction is used to write to memory. Injecting the 1st benchmark (FIBO) has shown a total error rate of 75%. This includes 50% SDC, followed by 25% of control. No crash encountered.

In the 2nd benchmark (QSRT) we recorded all types of errors, with a total rate of 66.66%. The highest error rate in this benchmark was the SDC with 38.1%, followed by 14.29% for both the control and the crashes.

The 3rd benchmark (MATH) has shown the highest error rate in the store injection experiment, reaching up to 90.32%, without crashes, we recorded 53.39% control and 36.94% SDC respectively.

C. Injecting "GEP" instructions:

Injecting the GEP instructions only concerns the 2nd and 3rd benchmarks since the 1st one has no GEP instructions. The GEP or "getelementptr" instruction is used to get the address of a subelement of an aggregate data structure. It performs address calculation only and does not access memory.

In both injection experiments of GEP in the 2nd (QSRT) and 3rd (MATH) benchmarks, we only observed SDC with 32.52 % and 52.38 % rates respectively.

D. Injecting Binary operators "BinOp" instructions:

Binary operators are used to do most of the computations in a program. They require two operands of the same type, execute an operation on them, and produce a single value.

The number of BinOp instructions in the 1st benchmark are statistically insignificant so we decide to exclude injecting the BinOps of this benchmark.

In the 2nd benchmark, we had a 50% total error rate, including all types of error, where crashes have the highest rate with 25%, SDC and control represent equally 12.5% of the total error rate.

In the 3rd benchmark (MATH), the error rate was the highest recorded in all the injection experiments, reaching

92.73%, this is because the MATH benchmark contains a large number of BinOp instructions. SDC has the highest rate, with 61.82%, followed by the control errors occupying 30.91%. We haven't recorded any crashes in this injection experiment.

E. Injecting "sext" instructions:

The "sext" instruction takes a value to cast, and a type to cast it to. This type of instructions is included in the 2nd and 3rd benchmarks (Qsort & Math) only.

The total error rate was 50% in the second benchmark (Qsort), divided between 39% of crashes and 11% of SDC. In this injection experiment we haven't recorded any control errors.

Injecting the 3rd benchmark has shown total failure, meaning a total error rate of 100%; every "sext" instruction that has been injected in this benchmark produced a faulty outcome. The highest fault type was crashes with 78.57%, followed by 18.57% of SDC and 2.86% control.

F. Injecting "icmp" Instructions:

The "icmp" instruction returns a Boolean value or a vector of Boolean values based on comparison of its two integer, integer vector, pointer, or pointer vector operands.

For both the 1st and 3rd benchmarks the number of "icmp" instructions was insignificant so we decided to only include the 2nd benchmark in this injection experiment.

The total error rate in the 2nd benchmark after injecting "icmp" instructions was 44.44%, this includes the SCD and control with an equal error rate of 22.22%.

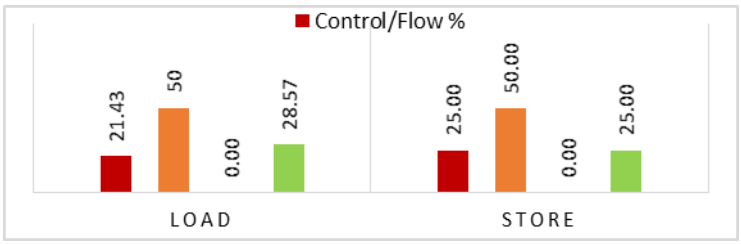

Figure 6. Injection of the different types of instructions of the unprotected code (fib)

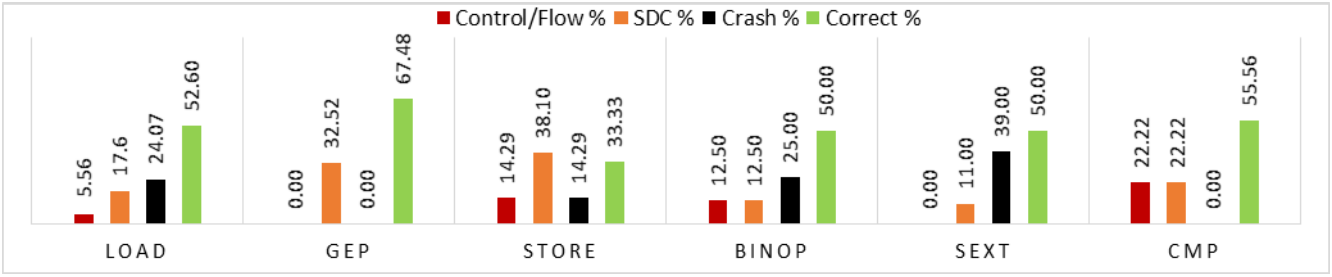

Figure 7. Injection of the different types of instructions of the unprotected code (qsrt)

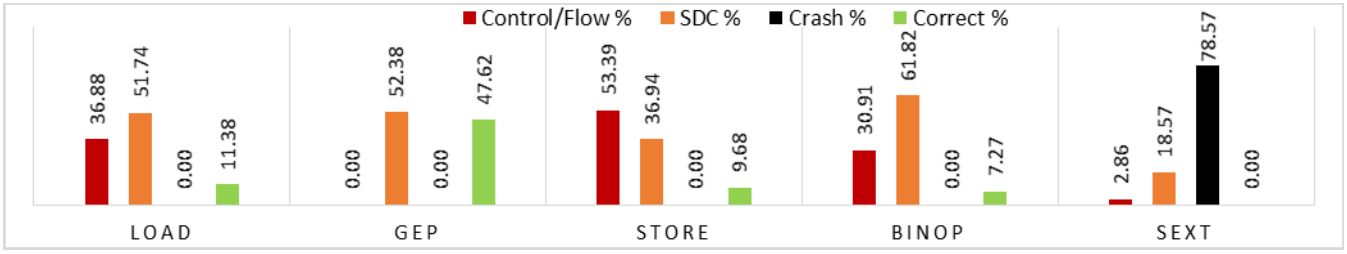

Figure 8. Injection of all instructions of unprotected math (solvecubic, rad2deg, deg2rad, uqsort)

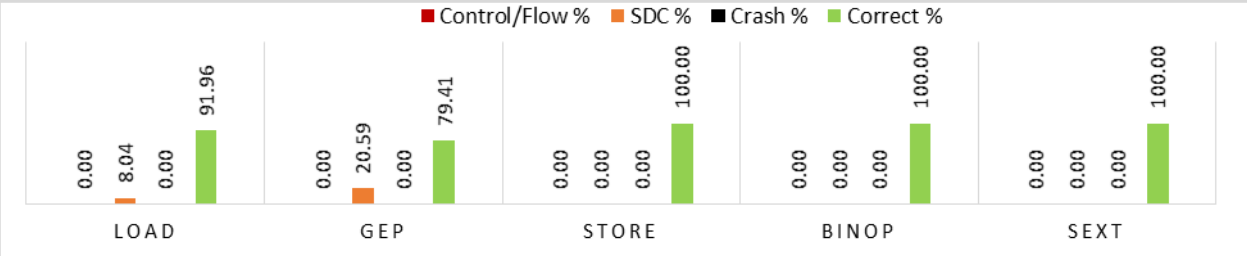

Figure 9. Injection of all instructions of the protected math (solve cubic, rad2deg, deg2rad, uqsort)

5.2 Injecting Protected Code

Injecting the protected code and comparing it to the injected unprotected code of Section 5.1 will quantify what error coverage is provided from our protection technique to the benchmarks. Even whilst injecting in every single instruction (full coverage), the new developed protection code we observed 0% error for the Fibo and Qsort benchmarks against the simulated SEEs from our fault injector. We achieved this result as the protection code replicates all instruction types (CPU instructions + memory instruction) and all datatype formats; i32, i32*, i1, i8, i8*, i64, float and double, float and double pointers. Our passes call a voter function to decide the correct outcome that will replace the faulty register value at runtime.

In the 3rd benchmark (Math) including (SolveCubic, Rad2Deg, Deg2Rad, UQsort), we had 100% correct outcome of injecting the Store memory instruction and all CPU instructions ("BinOp" & "sext"), however we have recorded some errors when injecting "load" and "GEP" memory instructions as shown in Fig 9. In this 3rd benchmark, we recorded 20.59% error rate, representing SDC when we injected the "GEP" instructions, and an error rate of 8.04% when injecting the Load instruction. The total error rate (counting all instruction types) of this benchmark has been dropped from 87.10% to 4.45%.

We have built our fault pass assuming single faults to quantify SEEs, however we predict our protection code will mitigate multiple bit errors provided they are in the same word. We also can protect multiple but separate variables provided they are TMR'ed using our method. Directly using our protection technique prevented crashes and control errors compared no protection.

5.3 ACEDR Time Overhead

In this section, we study the performance of the previous benchmarks under protection codes without injecting errors to investigate the overhead or time delay added in applying our protection code to the different benchmarks. We started by recording the execution time it takes to execute the unprotected code, after that we protect the code with our LLVM compiler passes, and then measure the execution time of the protected code. We used the Linux tool perf, to measure the delays and the number of processor cycles [30].

For the desktop architecture, the Intel core i5-3470 with 3.2 GHz frequency, the time overhead was as follows;

- For the Fibo benchmark, the overhead was 6.44%,
- For the Qsort benchmark, the overhead was 13.89%,
- For the Math benchmark (including SolveCubic, Rad2Deg, Deg2Rad, UQsort) the overhead was 8.55%.

For the 2nd processing embedded platform Raspberry Pi 3, we obtained the following overheads;

- For the first Fibo benchmark the overhead was 12.24%,
- For the second Qsort benchmark the overhead was 16.66%,
- For the third Math benchmark including SolveCubic, Rad2Deg, Deg2Rad, UQsort) the overhead was 15.28%.

The high performance (low time overhead) was thanks to the pipelining of the independent redundant instructions into multiple cores which can execute migrating applications in parallel. We also find that the desktop performed better than the embedded Raspberry Pi 3 architecture. This could be due to the differences in compiler optimisation performance, differences in the instruction set architecture (ISA), or also further underlying hardware and software issues; e.g. the operating system used. Future work needs to investigate these additional variables and how they affect final performance and overhead. Despite this, we demonstrate the portability of the LLVM code, and compilation of protected binary applications for differing processors.

6. Conclusion

Bit-flips originating from SEEs are becoming a prominent problem in the processor architectures. It is crucial for designers in both mainstream and embedded or critical processing systems to ensure the reliability of their systems. Systems with redundant hardware that make use of hard

ECC and TMR will elevate the design complexity of terrestrial applications, often eliminating it as an option. Soft error detection and recovery methods are viable alternatives with high coverage and low overhead and allowing for the best trade space between reliability and performance. Engineers now have more flexible ways of protecting their processing architectures.

New automatic compiler error detection and recovery (ACEDR) techniques have been implemented at compiler level in this research. We have implemented and verified a new error injection tool with experiments on different benchmarks in order to test the coverage of our software protection techniques. We have injected all instructions of the chosen benchmarks, where we divided the instructions to two main categories; Memory instructions, allowing read/write operations on the memory of the processing architecture, and CPU instructions including logic & arithmetic operations. To quantify our results, we injected both the protected and the unprotected code and compared the results. We show that CPU registers and their data or instructions can be fully protected against the bit-flips caused by the fault injection experiment simulating SEEs. For both the first and second benchmarks, we demonstrated that all instructions can be fully protected with almost 100% error injection coverage. When injecting errors in code, we find applications have greatly improved instruction execution outcomes: from 26.92% to 100% for Fibo, 53.84% to 100% for Qsrt and 12.59% to 95.55% for Math.

In the 3rd benchmark (Math) including (SolveCubic, Rad2Deg, Deg2Rad, UQsort), the CPU instructions were fully protected, however, for the memory instructions; GEP and Load have recorded 20.59% and 8.04% errors respectively. Even though the 3rd benchmark was not fully covered, all crashes and control/flow errors have been eliminated using our ACEDR soft-protection techniques in all experiments performed. The high error injection and protection coverage is due to the replication of all data and instruction types, including i32, i32*, i1, i8, i8*, i64, float & double, float & double pointers. The ACEDR protection mitigates different error types (crashes, SDC, control/flow), with low time overhead in multi-core across multiple platforms, enabling the user of our protection technique to many architectures and trade between reliability and performance. We find the delays measured were not highly significant because of the pipelining of independent redundant instructions, allowing the use of abundant resources of our CPU architectures, without causing a bottleneck.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the Algerian Space Agency, UK Space Agency and Surrey Space Centre at the University of Surrey for funding this study.

REFERENCES

[1] Baumann, R. C. (2001). "Soft errors in advanced semiconductor devices-part I: the three radiation sources." IEEE Transactions on device and materials reliability **1**(1): 17-

[2] O'Gorman, T. J., et al. (1996). "Field testing for cosmic ray soft errors in semiconductor memories." IBM Journal of Research and Development **40**(1): 41-50.

[3] Shivakumar, P., et al. (2002). Modeling the effect of technology trends on the soft error rate of combinational logic. Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, IEEE.

[4] Avizienis, A., et al. (2004). "Basic concepts and taxonomy of dependable and secure computing." IEEE transactions on dependable and secure computing **1**(1): 11-

[5] Horst, R. W., et al. (1990). Multiple instruction issue in the NonStop Cyclone processor. ACM SIGARCH Computer Architecture News, ACM.

[6] Slegel, T. J., et al. (1999). "IBM's S/390 G5 microprocessor design." IEEE micro **19**(2): 12-23.

[7] Tremblay, M. and Y. Tamir (1989). Support for fault tolerance in VLSI processors. Circuits and Systems, 1989., IEEE International Symposium on, IEEE.

[8] Phelan, R. (2003). "Addressing soft errors in ARM core-based SoC." ARM White Paper.

[9] Shirvani, P. P., et al. (2000). "Software-implemented EDAC protection against SEUs." IEEE transactions on reliability **49**(3): 273-284.

[10] Lattner, C. and V. Adve (2006). LLVM language reference manual

[11]Aggarwal, N., et al. (2007). Configurable isolation: building high availability systems with commodity multi-core processors. ACM SIGARCH Computer Architecture News, ACM.

[12] Walters, J. P., et al. (2011). Software-based fault tolerance for the Maestro many-core processor. Aerospace Conference, 2011 IEEE, IEEE.

[13] Avizienis, A. (1985). "The N-version approach to fault-tolerant software." IEEE Transactions on software engineering(12): 1491-1501.

[14] Vajda, A. (2011). Multi-core and many-core processor architectures. Programming Many-Core Chips, Springer**:** 9-43.

[15] Mitropoulou, K., et al. (2013). DRIFT: Decoupled compiler-based instruction-level fault-tolerance. International Workshop on Languages and Compilers for Parallel Computing, Springer.

[16] Fritts, J. E., et al. (2009). "MediaBench II video: Expediting the next generation of video systems research." Microprocessors and Microsystems **33**(4): 301-318.

[17] Henning, J. L. (2000). "SPEC CPU2000: Measuring CPU performance in the new millennium." Computer **33**(7): 28-35.

[18] Piotrowski, A. (2010). "Automatic installation of software-based fault tolerance algorithms in programs generated by GCC compiler." International Journal of Microelectronics and Computer Science **1**(3): 263-268.

[19] Reis, G. A., et al. (2007). "Automatic instruction-level software-only recovery." IEEE micro **27**(1).
[20] Feng, S., et al. (2010). Shoestring: probabilistic soft error reliability on the cheap. ACM SIGARCH Computer Architecture News, ACM
[21] Yu, J. and M. J. Garzaran (2007). Compiler optimizations for fault tolerance software checking. Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, IEEE Computer Society.
[22] Reis, G. A., et al. (2005). SWIFT: Software implemented fault tolerance. Proceedings of the international symposium on Code generation and optimization, IEEE Computer Society.
[23] Effort, U. O. The OpenIMPACT IA-64 Compiler.
[24] Zhang, Y., et al. (2012). "DAFT: decoupled acyclic fault tolerance." International Journal of Parallel Programming **40**(1): 118-140.
[25] Wang, C., et al. (2007). Compiler-managed software-based redundant multi-threading for transient fault detection. Proceedings of the International Symposium on Code Generation and Optimization, IEEE Computer Society.
[26] Mitropoulou, K. (2015). "Performance optimizations for compiler-based error detection."
[27] Stallman, R. M. (1989). "Using and porting the gnu compiler collection." Free Software Foundation **51**: 02110-01301.
[28] Nezzari, Y. and C. Bridges (2017). Compiler extensions towards reliable multicore processors. Aerospace Conference, 2017 IEEE, IEEE.
[29] Consortium, M. (2015). "MediaBench II benchmark." URL http://euler. slu. edu/~ fritts/mediabench/. Referenced: 04-15.
[30] de Melo, A. C. (2010). The new linux'perf'tools. Slides from Linux Kongress.

BIOGRAPHY

Yasser Nezzari *State Engineering Degree in Control Engineering in 2014 from the Institute of Electrical and Electronic Engineering at the Univesrity of Boumerdes in Algeria. He is now a 2nd year PhD student in Surrey Space Centre researching software compilers, radiation tolerance using software, and adaptive software.*

Dr Christopher P. Bridges *(BEng, 2005; PhD 2009) leads the On-Board Data Handling (OBDH) research group within Surrey Space Centre (SSC). He researches software defined radios, real-time embedded systems, agent computing, Java processing, multi-core processing in FPGAs, and astrodynamics computing methods in many spaceflight payloads. In 2013, he designed, built and still operates the UK's first CubeSat (STRaND-1) with SSTL and now contributes towards computing hardware and software with SSTL, on ESA's ESEO mission and also the NASA-JPL/CalTech AAReST mission.*