Compiler Extensions towards Reliable Multicore Processors

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 current trend in commercial processors is producing multi-core architectures which pose both an opportunity and a challenge for future space based processing. The opportunity is how to leverage multi-core processors for high intensity computing applications and thus provide an order of magnitude increase in onboard processing capability with less size, mass, and power. The challenge is to provide the requisite safety and reliability in an extremely challenging radiation environment.

The objective is to advance from multiple single processor systems typically flown to a fault tolerant multi-core system. Software based methods for multi-core processor fault tolerance to single event effects (SEEs) causing interrupts or ‘bit-flips’ are investigated and we propose to utilize additional cores and memory resources together with newly developed software protection techniques. This work also assesses the optimal trade space between reliability and performance. Our work is based on the modern compiler “LLVM” as it is ported to many architectures, where we implement optimization passes that enable automatic addition of protection techniques including N-modular redundancy (NMR) and error detection and correction (EDAC) at assembly/instruction level to languages supported. The optimization passes modify the intermediate representation of the source code meaning it could be applied for any high level language, and any processor architecture supported by the LLVM framework. In our initial experiments, we implement separately triple modular redundancy (TMR) and error detection and correction codes including (Hamming, BCH) at instruction level. We combine these two methods for critical applications, where we first TMR our instructions, and then use EDAC as a further measure, when TMR is not able to correct the errors originating from the SEE.

Our initial experiments show good performance (about 10% overhead) when protecting the memory of code using double error detection single error correction hamming code and TMR (Triple modular redundancy), further work is needed to improve the performance when protecting the memory of code using the BCH code. This work would be highly valuable, both to satellites/space but also in general computing such as in aircraft, automotive, server farms, and medical equipment (or anywhere that needs safety critical performance) as hardware gets smaller and more susceptible.

TABLE OF CONTENTS

TABLE OF CONTENTS.....	1
1. INTRODUCTION.....	1
2. SOFTWARE-BASED RADIATION HARDENING	2

3. MULTICORE ARCHITECTURE & LLVM.....	2
4. PROPOSED LLVM EDAC & TMR PASSES	2
5. IMPLEMENTATION & EXPERIMENTAL SETUP	3
5.1. SOFTWARE TMR EXPERIMENTS	3
5.2. IN-LINE SOFTWARE EDAC EXPERIMENTS	4
6. RESULTS & EVALUATION.....	5
7. CONCLUSION & FUTURE WORK	5
REFERENCES	5
BIOGRAPHY	6

1. INTRODUCTION

Processors for space have stringent requirements such as high performance, low cost, low power dissipation, and immunity from radiation and environmental effects. Processors available for space applications fulfil only some of these requirements. A revolution has taken place in commercial off the shelf (COTS) processor architectures in recent years. Multi-core processing has enabled a two to three order-of-magnitude improvement in performance, fulfilling many of the above requirements except for their susceptibility to radiation and environmental effects.

This research contributes to alleviating this problem by extending compiler functionality to mitigate against space-borne SEEs on processor architectures and allowing for regular coding found on Earth to be applicable in space. The current gap in performance between COTS and radiation hardened by design (RHBD) processor architectures is estimated to be between five and ten years, and at a fraction of the cost. Furthermore, there is an ever increasing need for greater processing capabilities in space applications that continually pushes the space industry to consider using COTS components. Common practices in industry, such as hardware triplication, hardware-based error correction, and cross strapping, allow for reduced risk to SEEs in space. While both Radiation hardened (RH) and RHBD processors are less susceptible to radiation effects, they are expensive and only target a very small market (aerospace/defense). But by exploiting opportunities in commercial multi-core architectures and treating this system as a set of redundant processors, we can leverage the multi-core architecture in software, associated high frequency and throughput performance. This can provide an order of magnitude increase in onboard processing capability with less volume, mass, and power.

2. SOFTWARE-BASED RADIATION HARDENING

In the ARGOS project at Stanford [1], experiments were carried out on two processor boards: one radiation hardened, and the other using COTS parts with different software fault-tolerance techniques. The experiment on the COTS processor implemented various error detection and correction techniques. The two processors are 32-bit, the RH3000 and the IDT3081. In this experiment, software-implemented EDAC provided protection for code segments and significantly enhanced the availability of a system in a “low-radiation” space environment. The software EDAC implemented in the ARGOS experiment could be widely improved using a better performance COTS multicore CPU architectures, where faster, multiple error detection and correction algorithms could be implemented by exploiting the greater resources.

Shoestring [2], Fault Tolerance Software Checking [3], and DAFT [4] which are implemented using the LLVM compiler. While Shoestring is a single core solution and DAFT is multicore CPU protection solution, memory in these techniques is still vulnerable to SEEs. Error Detection by Duplicated Instructions (EDDI) [5], the CASTED, decoupled Compiler-based instruction-level fault-tolerance (DRIFT) [6], Composite Data Type Protection Algorithm (CDTP) [6] are implemented using the GCC compiler, by duplicating instructions and inserting compare instructions where needed. EDDI is capable of covering both the CPU and memory from SEEs, but only ported to the MIPS architecture. CDTP only protects memory, with significant overhead. CASTED and DRIFT only protect the CPU. Software Implemented Fault Tolerance (SWIFT) [7] and SRMT are implemented using OpenIMPACT [8] and Intel production compiler and ICC 9.0 compilers respectively. SWIFT is single core solution, SRMT is multicore but has large overhead. Both previous techniques consider memory protected in hardware. Other techniques for protecting only the CPU are Process-Level Replication (PLR), Thread Level Replication (TLR), Heartbeats [9], and Efficient Online Multiprocessor Replay via Speculation and External Determinism (Respec) [10], PLR and TLR replicate code three times, and use a majority voter for error detection and correction. Respec duplicate code twice and rollback to a state free of errors in case of error detection. Heartbeat uses a scoreboard in memory updated by the CPUs. All of these techniques add large overhead and the protection code has to be added manually.

It has been observed that a minimal overhead was produced when error detection and correction codes were implemented at compiler level. Amongst compiler implementations, LLVM has shown to be popular with many projects. When providing a multicore solution (error detection and correction), it is inevitable to have an overhead, caused by spawning, communicating, and joining threads. Current solutions implemented in a single core do not show this.

3. MULTICORE ARCHITECTURE & LLVM

The technological advances in COTS has led to a dramatic growth in the clock frequency and in the quantity of logic (number of transistors) that a chip can host. These new features are exploited in order to further boost performance using architectural techniques. Figure-1 shows the significant gap between terrestrial and space processors, where terrestrial processor reach up to 3.6 GHz in frequency and up to 238,000 MIPS. Space processors lag this advancement, reaching only 1 GHz in frequency and 4000 MIPS.

Figure 1. Performance of Earth vs Space Processors

By making terrestrial processors radiation tolerant using efficient software techniques, the gap could be closed and the space industry could benefit from new technological advances.

4. PROPOSED LLVM EDAC & TMR PASSES

This work proposes implementing LLVM passes to automatically detect and correct errors caused by the SEEs. This work will be supported by multiple high level languages, and multiple processing architectures (supported by the LLVM framework). The LLVM Compiler [11] is the backbone of this work and is broken into three parts:

- The Frontend takes a high level language and transforms it into LLVM Intermediate Representation (IR). This stage uses the Lexer which transforms a sequence of characters into a sequence of tokens and classifies them. The Abstract Syntax Tree (AST), representing the source code of a programming language in a tree form, and the Parser examine the lexed code syntactically according to the rules of the language's grammar.
- The Optimizer takes the LLVM IR and returns an optimized LLVM IR – e.g. for area, speed, etc.
- The Backend takes the optimized LLVM IR and returns machine or executable code suitable for a given architecture.

In our implementation, we are going to modify the optimizer stage of the compiler allowing us to target multiple high level languages and multiple processing architectures. Figure-3 shows examples of high level languages and processing architectures supported by LLVM.

Figure 3. LLVM Optimizer with examples of supported high level languages & processing architectures.

The automatic implementation of error detection and correction code is achieved by adding redundancy and comparison instructions. We begin by running an analysis pass that will determine if and where protection can be added. This analysis pass will be able to analyze the instructions, functions and memory to determine data type, return values, and useful statistics regarding the memory usage, including new allocations and read/write operations.

Both memory and CPUs will be protected, starting with the memory, and later the pass will be extended to protect the multicore architecture. The extension to multicore will be achieved by automatically calling parallelism libraries, starting with the pthread library [12], to achieve redundancy in available CPUs. Techniques to reduce the spawning, communication and joining of threads will be implemented in order to minimize the overhead.

The memory will be protected using combinations of NMR or EDAC, NMR includes dual modular redundancy and triple modular redundancy (TMR) which depends on the resources available. The EDAC will be implemented using single and multiple error correction codes depending on the error rate. In addition to minimizing the overhead of multithreading, this work considers automatic parallelization and improving data locality for better use of the caches.

5. IMPLEMENTATION & EXPERIMENTAL SETUP

In these experiments, analysis and optimization passes have been implemented in order to protect the memory of the processor architecture. The analysis pass checks the code, finds the memory instructions and provides some statistics. Optimization pass adds protection code to the LLVM IR of the code, the pass is able to detect the type of memory instruction and summons the appropriate technique to protect it accordingly. In the first experiments TMR and EDAC have

been used, after the high level code intended to protect is transformed to LLVM IR.

To start, our new LLVM pass implementation is introduced by combining different techniques for increased reliability using TMR for memory, and differing EDAC codes for the processor. A code in the LLVM IR is divided to the following layers [13]:

- A module is a top level LLVM class, every other layer is included in it. It represents the highest level structure. An iterator over a Module returns all its Functions.
- Modules include Functions a class representing a single procedure containing chunks of executable code. An iterator over a Function returns all of its BasicBlocks.
- BasicBlocks are housed by Functions, they represent single entry single exit section of the code. The BasicBlock houses a list of instructions, the last one is a terminator instruction. An iterator over a BasicBlock returns all its instructions.
- An Instruction is a single code statement. Each instruction has an opcode and a parent (BasicBlock).

5.1. Software TMR Experiments

In order to implement instructions TMR, two replicates instructions are required in addition to the original one. Replicating instructions could be achieved either by cloning the original ones using clone() [14], or by building new instructions using the IRBuilder [15]. Here, the aim is to implement the algorithm shown in Figure 4, where the numbers in the flowchart represent the instructions that will be TMR-ed. clone() [14] returns a replicate of the instruction, similar to the original except that the clone instruction has no BasicBlock parent (not inserted into a BasicBlock), and it has no class name. In order to successfully clone an instruction, it should be assigned a name and a parent BasicBlock, meaning it should be included in the control flow graph (CFG). Ignoring this part is the reason why the new instructions are deleted by the optimizer. IRBuilder provides a uniform API [16] for creating instructions and integrating them into a BasicBlock: either at the end of a BasicBlock, or at a specific location.

The instruction TMR is implemented following the steps:

1. The pass iterates through all the code layers and detects *memory allocation instructions*, and allocates new addresses of the code. These new addresses are used to replicate instructions three times. In two allocations, the redundant values are stored and in the 3rd alloca or allocaTMR, the protected value is stored.
2. The 2nd step is to iterate again through the code layers and detect the *store instructions*, used to store a data in the preceding memory addresses. Two new stores for every detection are created in the replicated address allocation in step 1, Figure 5.
3. Iterate through all the code layers and detect the *load*

instructions, and are used to return data of a given address in memory. If the address of the detected load matches the original allocation instruction of step 1, it will be replaced with `allocaTMR` – which is the protected address.

Figure 4. Software TMR Algorithm

Figure 5. New allocations of the original store instruction

The TMR algorithm shown in Figure 4 uses the compare instruction to make decisions, and then it jumps to execute the correct block accordingly. The jumps are a conditional or unconditional branch instructions, both `cmp` and `branch` are

created using the *IRBuilder*. In the decision making the outcome of the compare instructions are combined using *and* instructions.

5.2. In-line Software EDAC Experiments

Two methods for automatically implementing any EDAC code were implemented in order to protect the memory of a high level programming language at runtime. For proof of concept, protection of integers and floating point using selected EDAC algorithms have been implemented; the Hamming code for single error correction double error detection and BCH codes with the ability of multiple error detection and correction. There were two choices for implementing inline EDAC:

- The 1st method turns the EDAC code into LLVM IR and then links it with the program that is intended to be protected along with the benchmark, extra instructions, new allocations for the EDAC check bits, and function calls to the EDAC encoding and decoding functions; see Figure 6.

Figure 6. EDAC using Linker

- The 2nd method is directly in the pass, without using the linker, meaning the EDAC encoding and decoding function are automatically generated using our compiler pass; see Figure 7.

Figure 7. EDAC without Linker

The following steps explain the pass of the 2nd method of implementation:

- The pass will run through the intermediate representation of the code and detect all the memory allocations.
- For every allocation in step 1 a new allocation to store the check bits are added.
- If a store is detected to the allocation of step 1, it will be encoded, and then the check bits are stored inside the new allocations that were created in step 2.
- Each time a new store is detected to the allocations in step 1, new check bits are created to replace the check

bits of step 3.

5. In the decoding part, the decoder function will be called every time a read from the protected allocation is found, in case of error detection we correct it and replace the damaged data and check bits. This function is also called periodically in case of error occurrence between reading and writing.

6. RESULTS & EVALUATION

After the implementation, the work has been evaluated in terms of the delay, and the size of the additional code, different tools for our evaluation have been used.

1. perf to measure delays and number of cycles [17].
2. A profiling tool has been implemented, an LLVM pass to count the number of instructions in our code.

Table 1 shows the CPU profiling results for Fibonacci series before and after implementing the following techniques: TMR, Hamming, TMR + Hamming and BCH code.

Table 1. Profiling results of Fibonacci Benchmark

Code	Execution Time (s)	No. of Cycles	No. of Instructions	Over-head (%)
Fibonacci	0.001286844	1,856,346	48	0
+ TMR	0.001419664	1,898,231	157	10.32
+ Ham	0.001359356	2,052,954	316	5.63
+ TMR + HAM	0.001424670	2,218,110	500	10.71
+ BCH	0.008445104	15,638,019	1961	556.26

The primary results show a very encouraging outcome, where code for memory protection has been automatically generated, this method supports LLVM high level languages and its CPU architectures. When using TMR, Hamming code, and TMR combined with Hamming, the memory of code is protected with a minimal overhead, and this was expected, since both techniques do not have very complicated computations when detecting and correcting errors. In the other hand, BCH has shown enormous overhead this is due to the intense computations of the encoding/decoding functions, this could be improved by increasing the data block size of the encoding/decoding functions, to have less function calls. Some compiler optimizations could be used to make the code run faster by automatically parallelizing function calls and reducing data dependency, especially in loops for better caching.

7. CONCLUSION & FUTURE WORK

Bit-flips originated from SEEs are a prominent problem in memory cards. EDAC and TMR codes have been used to resolve this problem effectively. These codes are commonly implemented in hardware using the redundant memory resources and encoding-decoding hardware. In processor

architectures where hardware memory EDAC is not at hand, the reliability of the processor architecture can be enhanced by enabling shielding via software. Codes and techniques that can be used for automatic software implementation of EDAC at compiler are discussed and compared. TMR is the fastest in terms of execution time, with an average of 10 % overhead, however it is more suitable for small sized code, because it consumes more memory. The EDAC codes implemented are the Hamming and the BCH codes, while both only add check bits to the memory, making them suitable for codes with larger sizes, the BCH codes is the slowest (5 times slower than the original overhead), because of its ability to detect and correct multiple errors. The overhead generated could be minimized using automatic compiler parallelization, which will be implemented and tested in the future. This work will be further extended towards multicore architectures protection, in addition to memory system protection that has been started in the 1st experiments of this work.

This work has automatically implemented protection code on the LLVM's supported high level languages. This was enabled using compiler passes to generate the protection code. The EDAC protection code could be implemented using the linker as additional step or directly using our pass. Different benchmarks have been tested, our implementation is able to protect integers and floating point, and will be extended to protect other data types in the future. The primary results show good performance for TMR and single error correction codes, compared to the state of the art the CDP technique used to protect memory with GCC compiler with an overhead of 86%-146%, and this is encouraging to implement CPU protection side by side with memory protection that have been implemented, with an overhead predicted to be in the norms of the state of the art technique EDDI protecting both memory and CPU with an overhead of 62%, furthermore this work will be extended to multicore CPU protection. Turning to multicore protection from SEEs will be achieved using automatic code generation for parallelism libraries call, starting with the pthread library and extending the work towards OpenMP [18].

REFERENCES

- [1] Shirvani, P.P. and E.J. McCluskey, *Fault-tolerant systems in a space environment: The CRC ARGOS project*. 1998.
- [2] Feng, S., et al. *Shoestring: probabilistic soft error reliability on the cheap*. in *ACM SIGARCH Computer Architecture News*. 2010. ACM.
- [3] Yu, J. and M.J. Garzaran. *Compiler optimizations for fault tolerance software checking*. in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. 2007. IEEE Computer Society.
- [4] Zhang, Y., et al., *DAFT: decoupled acyclic fault tolerance*. *International Journal of Parallel Programming*, 2012. 40(1): p. 118-140.
- [5] Oh, N., P.P. Shirvani, and E.J. McCluskey, *Error*

detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 2002. **51**(1): p. 63-75.

- [6] Mitropoulou, K., V. Porpodas, and M. Cintra. *DRIFT: decoupled CompileR-based instruction-level fault-tolerance*. in *International Workshop on Languages and Compilers for Parallel Computing*. 2013. Springer.
- [7] Reis, G.A., et al. *SWIFT: Software implemented fault tolerance*. in *Proceedings of the international symposium on Code generation and optimization*. 2005. IEEE Computer Society.
- [8] Effort, U.O., *The OpenIMPACT IA-64 Compiler*.
- [9] Walters, J.P., et al. *Software-based fault tolerance for the Maestro many-core processor*. in *Aerospace Conference, 2011 IEEE*. 2011. IEEE.
- [10] Lee, D., et al., *Respec: efficient online multiprocessor replay via speculation and external determinism*. *ACM SIGARCH Computer Architecture News*, 2010. 38(1): p. 77-90.
- [11] Lattner, C., *The LLVM compiler infrastructure*. The LLVM compiler infrastructure, 2010.
- [12] Butenhof, D.R., *Programming with POSIX threads*. 1997: Addison-Wesley Professional.
- [13] Sampson, A., et al., *ACCEPT: A programmer-guided compiler framework for practical approximate computing*. University of Washington Technical Report UW-CSE-15-01, 2015. **1**.
- [14] LLVM. *LLVM Cloning class*. 2016; Available from: http://llvm.org/docs/doxygen/html/Cloning_8h.html.
- [15] LLVM. *LLVM IRBuilder*. 2016; Available from: http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilder.html.
- [16] Lattner, C., et al., *LLVM Programmer's Manual*. 2008.
- [17] perf. *Linux profiling with performance counters*. 2015; Available from: https://perf.wiki.kernel.org/index.php/Main_Page.
- [18] Yliluoma, J. *Guide into OpenMP: Easy multithreading programming for C++*. 2016; Available from: <http://bisqwit.iki.fi/story/howto/openmp/>.

BIOGRAPHY

compilers, radiation tolerance using software, and adaptive software.

Yasser Nezzari State Engineering Degree in Control Engineering in 2014 from the Institute of Electrical and Electronic Engineering at the University of Boumerdes in Algeria. He is now a 2nd year PhD student in Surrey Space Centre researching software

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 in missions with SSTL, on ESA's ESEO mission and also the NASA-JPL/CalTech AReST mission.

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