A Compiler-Based Infrastructure for Fault-Tolerant Co-Design

Felipe Restrepo-Calle
University of Alicante
03690 Alicante, Spain
frestrepo@dtic.ua.es

Antonio Martínez-Álvarez
University of Alicante
03690 Alicante, Spain
amartinez@dtic.ua.es

Hipólito Guzmán-Miranda
University of Sevilla
41092 Sevilla, Spain
hipolito@zipi.us.es

F.R. Palomo
University of Sevilla
41092 Sevilla, Spain
rogelio@zipi.us.es

M.A. Aguirre
University of Sevilla
41092 Sevilla, Spain
aguirre@zipi.us.es

Sergio Cuenca-Asensi
University of Alicante
03690 Alicante, Spain
sergio@dtic.ua.es

ABSTRACT

The protection of processor-based systems to mitigate the harmful effects of *transient faults* (*hardening*) is gaining importance as technology shrinks. Hybrid hardware/software hardening approaches are promising alternatives in the design of such fault tolerant systems. This paper presents a compiler-based infrastructure for facilitating the exploration of the design space between hardware-only and software-only fault tolerant techniques. The compiler design is based on a generic architecture that facilitates the implementation of software-based techniques, providing an uniform isolated-from-target hardening core. In this way, these methods can be implemented in an architecture independent way and can easily integrate new protection mechanisms to automatically produce hardened code. The infrastructure includes a simulator that provides information about memory and execution time overheads to aid the designer in the co-design decisions. The tool-chain is complemented by a hardware fault emulation tool that allows to measure the fault coverage of the different solutions running on the real system. A case study was implemented allowing to evaluate the flexibility of the infrastructure to fit the reliability requirements of the system within their memory and performance restrictions.

Categories and Subject Descriptors

B.8 [**Performance And Reliability**]: Reliability, Testing, and Fault-Tolerance; B.8 [**Performance And Reliability**]: Performance Analysis and Design Aids; C.3 [**Special-Purpose And Application-Based Systems**]: Real-time and embedded systems; C.4.2 [**Performance of Systems**]: Fault tolerance

General Terms

Reliability, Verification, Design

SCOPES '10, June 28-29, 2010, St. Goar, Germany

Keywords

Fault-tolerance, Single Event Effect (*SEE*), Single Event Upset (*SEU*), Hardening, Co-Design

1. INTRODUCTION

Microprocessor performance has been dramatically increasing during the recent decades. This fact has been possible mainly because the progressive miniaturization of its electronic components. However, as technology shrinks, the voltage source level and the noise margins are reduced, forcing the electronic devices to be less reliable and microprocessors more susceptible to *transient faults* [3, 30]. These intermittent faults are caused by external events, i.e. induced by radiation, without provoking a permanent damage, but may result in incorrect program execution by altering signal transfers or stored values [27].

Although these faults are commonly found in the space environment, they are also present in a lower measure in the atmosphere [7] and even at ground level [2]. Therefore, fundamental information about minimum environmental withstand conditions (including *transient faults*) for electronic components has been established by several technical committees in each industrial field. These documents define detailed qualification requirements that electronic components must meet for their use. Some examples of them are: for aerospace applications, `ESA PSS-01-609` (The Radiation Design Handbook) [8]; for avionic systems, `IEC/TS 62396` (Process Management for Avionics - Atmospheric radiation effects) [11]; for military systems, `MIL-HSBK-817` (System Development Radiation Hardness Assurance) [6]; among others.

Applying redundant hardware has been the usual way to mitigate reliability problems. This strategy has been applied from low level structures (*ECC — Error-Correcting Code*, *parity bits*) to more complex components like functional units [1], co-processors [12], etc. In the same way, several approaches have exploited the multiplicity of hardware blocks available on multi-threaded/multi-core architectures to implement redundancy [31, 14, 9]. More recent techniques published in the literature [29] propose the selective hardening of the design, which means that a system needs to detect which parts of its circuitry are more vulnerable or have larger probability of provoking a catastrophic behavior of the design itself. However, these hardware-based approaches results very costly and for this reason unfeasible in many cases.

In recent years several proposals based on redundant software have been developed, providing both detection and error correction capabilities to applications. These works are especially motivated by the need for low cost solutions, ensuring an acceptable level of reliability [16] [20]. Some of them apply redundancy to high-level source code by means of automatic ruled transformations [22]. Some others use low level (assembler) instruction redundancy in order to reduce the code and execution time overheads caused by applying these methods and improve the detection rates [18, 17, 26]. However, only few of these techniques have been extended to allow the recovery of the system [21, 25]. Regarding tools for implement fault-tolerant software techniques, researchers have applied their own strategies, either by means of modifications over a known compiler to perform automatic code transformations [18, 26], designing ad-hoc tools [22] or applying transformation rules manually to the instructions [20].

Despite the wide number of different hardware-only and software-only methods, in many cases the optimal solution is an intermediate point, which combines software and hardware aspects (hardware/software co-design). Some recent works have shown the viability of this hybrid strategy [24, 4]. In this context, there are needed suitable tools which allow the designer to easily explore the design space in order to find the best trade-off that satisfy the performance, reliability, and hardware cost requirements of a design.

This paper presents a flexible compiler-based infrastructure for fault-tolerant embedded systems co-design. The novelty of our proposal is that the developed tools allow the implementation of different software based redundancy methods in a platform independent way. The compiler can selectively apply every software technique to get intermediate solutions that could be complemented with hardware redundancy to reach the fault coverage requirements and meet the system constraints at the same time. The tool-chain includes an instruction set simulator (*ISS*) that is able to get exact information about the code and performance overheads of the protection strategy adopted. Also the *ISS* can perform an estimation of the fault coverage offered by the software-based techniques applied, facilitating the early take of decisions to the designer. The present work completes the preliminary results obtained in [28] that show the flexibility of the proposal.

As a case study for validating our approach, we have developed a compiler front-end and back-end for the *PicoBlaze* soft-microprocessor [5] and a benchmark suite. As part of the case study, three software-based techniques have been implemented and evaluated using our infrastructure. The selective hardening feature has been explored applying one of the techniques to several subsets of registers from the microprocessor register file. Different trade-offs among code overhead, performance and fault coverage have been represented. Finally, the protected versions of the code produced by the compiler have been evaluated while running in the real system using the *FTUnshades* hardware fault emulation tool [10]. This is one of the tools used by the European Space Agency (*ESA*) to assess dependability for mission critical systems.

Next section provides background information about the fault model and important terminology. Section 3 presents our compiler-based infrastructure. Section 4 describes the mentioned case study, including the experiments and their results. Finally, Section 5 concludes the paper and suggests directions for future research.

2. FAULT MODEL AND TERMINOLOGY

In this paper we will focus on the well known *Single Event Upset* fault model. In this *SEU* fault model, only one bit-flip of a storage cell occurs throughout the execution of the program. This effect is caused by the ionization provoked by an incident charged particle. Despite its simplicity, the *SEU* fault model is widely used in the fault tolerance community to model real faults because it closely matches the real fault behavior [22].

In order to evaluate the system's reliability, we classify the injected faults according to their effect on the expected program behavior as it was proposed by Reis et al. [25]. If the fault provokes that the program completes its execution, but does not produce the expected output, this fault is called *Silent Data Corruption — SDC*. If the program completes its execution and produces the expected output, the fault is categorized as *unnecessary for Architecturally Correct Execution — unACE*. Finally, if the fault causes the program to abnormally finishes its execution or to remain forever into an infinite loop, we categorize this fault as *Hang*. Note that *SDC* and *Hang* are both undesirable effects (categorized together as *ACE* faults). In addition, it is worth noting that software-based techniques necessary introduce redundancy, and this causes two important facts to take into account. Firstly, these techniques increase the execution time of the programs, therefore the probability of fault occurrence is higher than for the original program whose execution time is smaller. Secondly, redundancy increases the number of bits present on the system, increasing the number of bits that are susceptible to fault. Therefore the fault coverage offered by a specific hardening strategy is directly related with the percentage of *unACE* faults and the execution time overhead.

3. COMPILER-BASED INFRASTRUCTURE

The most desirable features that a hardening platform must supply can be expressed as:

- flexible: that is, easy to extend its hardening capabilities.
- *API*–based internals: to easy interfacing.
- *hardware–agnostic*: to provide an *uniform isolated hardening core* for each supported microprocessor/microcontroller.
- retargetable output: to provide reusing of code.
- flow–control based analysis and code–injection routines.

- automatic reallocation/minimization of architecture resources to optimize fault coverage.

In order to implement these tasks we have evaluated a number of tools within this context. In this way, well–known compilers such as *gcc*, with a robust tool–chain and an efficient back-end system are not suitable for us at this moment taking into account:

- the optimization strategy was not thought for providing redundancy.
- the lack of an *API* regarding the compiler internals (*gcc*).
- a new redundancy stage would have several offtopic–interactions to deal with.

In short, the need for having a deep knowledge in the way code is to be transformed, have lead us to develop an entire *instruction–level* hardening infrastructure from scratch. That is, a hardening strategy based on assembler code. Moreover, we don't discard a further interaction with a high-level compiler such as *gcc*, in the sense that it can generate assembler code able to feed our hardening platform.

The compiler have been designed using *ANTLR* 3.2 [19] (a `LL(*)`-based framework for constructing recognizers, compilers, and translators from grammatical descriptions). The hardening overall platform is written using *C*#.

The proposed infrastructure establishes a complete software-hardened development environment because it not only allows to design and implement software-based techniques, but also permits to automatically apply them into the programs. The infrastructure is made up of two main components. The first one is a flexible multi-target compiler supporting several common hardening routines. The second one corresponds to an instruction set simulator (*ISS*) responsible for the functional validation of the hardened programs and their preliminary fault coverage evaluation. Fig. 1 shows the general scheme of the proposed infrastructure.

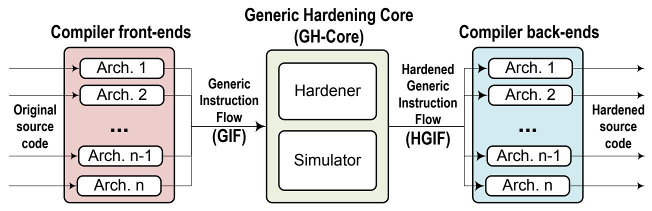


Figure 1: Infrastructure for Fault-Tolerant Co-Design

The main advantages of our proposal are:

- It is based on a *Microprocessor Generic Architecture* that permits to handle multiple microprocessor targets. This architecture is useful to provide a uniform hardening core compatible with usual microprocessors.
- It has a *Generic Hardening Core* (*GH-Core*) that allows to design and implement different techniques in a platform independent way. According to the reported results by different researchers, better accuracy level is achieved when low level instruction redundancy is employed. So, the *GH-Core* is guided by instruction level code transformation rules (assembler) that allow to automatically generate hardened code.
- The infrastructure is conceived to implement a wide suite of techniques, allowing to apply them in a *selective* way. For instance, if applying a particular set of hardening routines results inconvenient according to the requirements of the application (e.g. if the maximum execution time is exceded), it can be applied partially depending on the critical program's resources or sections. This means that the designer can achieve a protected version of the software by choosing from different techniques to apply and even exploring the design space that provide each one of these techniques.

The *compiler front-ends* take the original source code from a supported target, perform lexical, syntactical and semantical analyses, and finally generate a *Generic Instruction Flow* (*GIF*) as output. This flow represents an intermediate high level abstraction of a program that allows to perform a platform independent implementation of the hardening routines (in the *GH-Core*). After the hardening process, the *Hardener* produces a *Hardened Generic Instruction Flow (HGIF)*, which is taken by one of the *compiler back-ends* to generate the *hardened source code* for the selected specific microprocessor (see Fig. 1).

Using this scheme, the infrastructure compiler is also flexible in the sense that is possible to process a code written for a supported architecture and generate the hardened source code targeting the same original architecture or a different one by means of the several back-ends.

3.1 Microprocessor Generic Architecture

The generic architecture provides a workspace for the *Generic Hardening Core*. It gathers together every common elements from different architectures in order to facilitate the design and implementation of the state-of-the-art software-based techniques in a technology independent way. The *Microprocessor Generic Architecture* is defined by means of *Generic Instructions (GI)* and provides the necessary functionalities to suitably perform *Memory Management* and analyses to the *Control Flow Graph*. In this way, every supported microprocessor target defined by means of an *ISA* (Instruction Set Architecture) has a generic dual representation (*ISA*$'$) made up of generic instructions.

A *GI* is composed of the fields showed in Fig. 2.

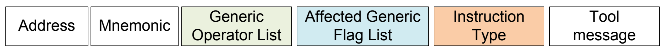


Figure 2: Format of Generic Instructions

The *Address* is the memory position where the instruction has been assembled by the compiler front-end; *Mnemonic* is simply the target's original mnemonic of the instruction (e.g.

ADD, JUMP, ...). The *Generic Operator List* is a linked list that contains the *Generic Operators* that are present in the instruction. Each operator is made up of three fields, they are: *operator type*, *addressing mode* and *real name*. The *operator type* defines the kind of operator, such as: Register, Literal, Address or Flag. The *addressing mode* can be: Absolute, Register Indirect, Immediate Literal, among others. Finally, the *Real Name* is the operator's original name. Next in the *GI*, there is the *Affected Generic Flag List* which is a linked list with the *generic flags* affected by the execution of the instruction. Each one of these is composed of *type* and *name*. The *Generic Flag Type* can be: Zero, Not Zero, Carry, Not Carry, Interrupt Enable, etc. The *Name* is the original flag name of the target architecture. Inside the *GI* there is also the *Instruction Type* that is used to classify the instructions according to its functionality. It is very important because the hardening process will depend on this type. For example, it is different to handle an arithmetic instruction and a control flow instruction during hardening. Some of the supported types are: interrupt, control flow, arithmetic, logic, I/O, shift/rotate, etc. Finally, the last field in a *GI* is *Tool Message* which is a log that the tool uses to register events.

Considering hardening purposes, as it was suggested by Reis et al. [26], we propose to classify in a special way those instructions whose function imply to cross the borders of the Sphere of Replication (*SoR*) [23]. The *SoR* is the logic domain of redundant execution. Therefore, when an instruction causes that some data enter inside the *SoR* (e.g. reading an input port, loading a value into a register or reading a value from memory), it will be classified as *inSoR*; and consequently when an instruction provokes data goes out from the *SoR* (e.g. writing on an output port, storing a value into the memory), it will be classified as *outSoR* (Fig. 3).

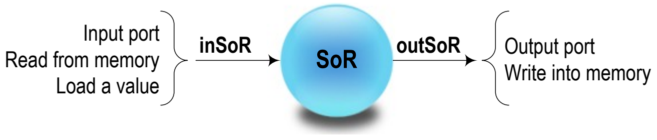


Figure 3: Instruction types related to crossing *SoR* boundaries

Note that the boundaries of the *SoR* could change according to the implemented technique. For instance, in *EDDI* [18] the memory subsystem is inside the *SoR*, so the instructions responsible to perform read/write operations over the memory do not cause that any data cross the *SoR* borders. In the same way, supposing partially replication of the register bank (i.e. not all of them replicated), those instructions against data stored in unprotected registers cause some data to cross through the *SoR* frontiers.

3.1.1 *Memory Management*

A common task for the software-based techniques is the insertion of instructions into the original code during compilation time. Therefore, it is necessary to supply the following memory management means within the hardener tool: identification of the memory map, extraction of the memory sections and modifications over the memory map. In developing this compiler, similarly to other approaches, it is assumed that the code being hardened does not exploit dynamic memory allocation: all the data structures are defined statically at compilation time. This is not a significant limitation for developers of embedded applications, which sometimes are forced to code standards that already avoid dynamic memory usage [13]. The following three possibilities were developed to keep update the memory map: *dilation*, *displacement* and *reallocation*.

Dilation. When one or more instructions are inserted during compilation time into a memory section, this section grows and some of the instructions addresses inside this memory section should be reassigned.

Displacement. If a dilation provokes that two or more memory sections share some addresses, which is an illegal situation, then the section must be completely moved and all its instructions addresses updated.

Reallocation. If there is a memory overflow caused by previous instructions insertions, then it is needed to perform a complete reallocation of the complete memory map. During this process, free memory space among memory sections is fully used. This situation takes place because of the typical reduced memory size in embedded systems.

3.1.2 *Control Flow Graph*

Since the *control flow graph* is the key for many techniques [17, 18, 26], our *Microprocessor Generic Architecture* allows to identify it from a given *GIF*.

The *control flow graph* is represented by a directed graph. In order to build it, first we must identify the program's *basic blocks*. A *basic block* is a group of instructions that are executed sequentially, without any jump instruction nor function call, excepting possibly the last instruction. Also, a *basic block* does not contain instructions being the destination of a call or jump instruction, excepting the first instruction. Each *basic block* represents a node in the graph. The control flow changes are represented in the graph as links among the nodes. Fig. 4 shows an example of a simple *control flow graph*.

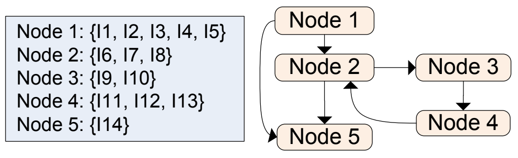


Figure 4: Control Flow Graph

In addition, when an instruction sends data outside of the *SoR*, it may provoke an unrecoverable error if that data is corrupted. In this way, it would be desirable to perform a verification of the data's correctness before it leaves the sphere. Therefore, in this paper we propose that the *nodes* (*basic blocks*) of the *control flow graph* should be subdivided into *subnodes* after each instruction classified as *outSoR* (Fig. 5).

3.2 Generic Hardening Core - *GH-Core*

The *GH-Core* has two main components: the *hardener* and the *ISS*.

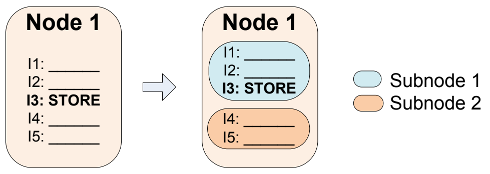


Figure 5: Subnodes

The *hardener* is based on the *Microprocessor Generic Architecture*. It is comprised of an Application Programming Interface (*API*) of hardening routines typical from software-based techniques algorithms. Developing new hardening algorithms is a task significantly easier by using this *API*.

The available options in the *hardener* able the designer to decide which technique will be applied, which is the replication level to be used in redundant instructions (i.e. duplication, triplication), which is the preferred recovery procedure to use (i.e. among different implementations of majority voters, in case of recovery techniques, or routines to be applied when a fault is detected). In this way, the hardener offers a complete control to the user and produce a new protected version of the *GIF* (*HGIF*).

Also, the *hardener* allows to design the hardening strategy by means of partial application of one or more software-based techniques. In case the tool is hardening a program and find that there are not enough available resources to keep adding redundancy (e.g. there are not enough registers to replicate), then the process continue without hardening until some resources are released. Furthermore, the selective hardening can be controlled by the designer, being able to determine which resources must be hardened. This feature permits to quickly explore the design space provided by the technique, obtaining a set of hardened versions to evaluate and determine their code overhead, performance degradation and reliability level.

The *ISS* has two main functionalities. On one hand, it assists the designer in the implementation of new software-based techniques. On the other hand, the *ISS* performs different analyses on the original and hardened *GIF*, and generates useful information to aid the designer in the co-design process.

As usual for the instruction simulators, the *ISS* presents information about the state of the resources of the architecture during and after the simulation process. Likewise, the tool allows to verify if the functionality of the hardened programs matches the original non-hardened programs functionality. This is possible by means of the `check-hardening` option that use information stored in the source code through a compiler *pragma* to know which the expected results are.

After the simulation process, the *ISS* presents a brief summary to inform the code and execution time overheads of the applied hardening technique. Also, it performs a characterization of the simulated programs, informing the percentage of executed instructions by its type: arithmetic, logical, control flow, etc.

Moreover, in order to evaluate the reliability provided by the applied techniques, the *ISS* is also able to inject *SEUs* during the simulation by means of *bit-flips* into the registers bits. The effect of the fault on the final outputs of the program are classified as was mentioned in Section 2: *unACE*, *SDC* and *Hang*.

The reliability results offered by the *ISS* are only preliminary estimations, because the simulation at this level does not take into account the micro-architectural details of the target microprocessor (for example, the user-hidden register such as those ones in pipeline). However, this is an useful information for tuning the hardening strategy and comparing different techniques. In order to obtain more realistic results, a hardware *SEU*-emulation tool is included to the infrastructure.

3.3 SEU-Emulation tool: *FTUnshades*

The *FTUnshades* system, described in [15], is a FPGA-based platform for the study of digital circuit reliability against radiation-induced soft errors. *SEU* affecting the circuit are emulated by inducing bit-flips in the circuit under study, by means of partial reconfiguration.

The system is composed of a FPGA emulation board and a suite of software tools for design preparation, testing of the emulated design, and analysis of the test results. The main software of the suite is the *FTUnshades Test aNalysis Tools* (*TNT*) program, which manages the communications with the board, the partial reconfiguration and the test campaign execution.

In the original version of *FTUnshades*, two instances of the circuit or module under test (*MUT*) are instantiated in the implemented design: *Target* and *Gold*. Faults are injected over the *Target* instance, whereas the *Gold* instance remains unchanged for comparison purposes.

The system has been extended for the study of microprocessor architectures. An exhaustive description of this extension can be found in [10]. Instead of two instances of the *MUT* (*Gold* and *Target*), the implemented design has just one instance of the *MUT* (*Target*), and the *Golden* instance is substituted by a *Smart Table* (see Fig. 6). This is needed because the typical cycle-by-cycle comparison would classify as *output error* the effect of faults that could be corrected if the *Target* microprocessor was given more processing time. The exact additional time, measured in clock cycles, that the affected microprocessor needs to output the correct value is called *recovery time*.

The *Smart Table* is an automaton which implements the *relaxed time restrictions* needed for the fault injection testing of microprocessors that implement software-based techniques.

The *Smart Table* can be configured in emulation-time (this is, after synthesis, implementation and *FPGA* programming). First, the *Smart Table* must be configured with the outputs of a *Golden Run* of the *Target* microprocessor. This means a whole emulation of the circuit processing workload is done, but *without* injecting any bit-flips. When being configured during a *Golden Run*, the *Smart Table* not only memorizes

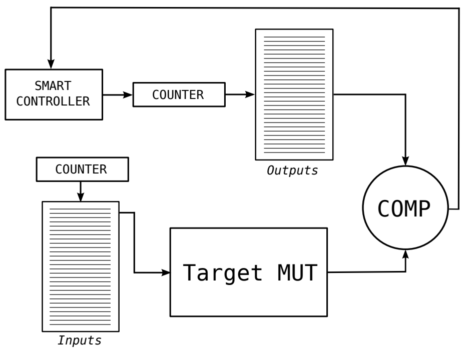


Figure 6: *FTUnshades* implementation approach using *Smart Table*

the sequence of the correct outputs, but also the time (in clock cycles) where the outputs change.

After filling the *Smart Table* with the *[expectedOutput, expectedCycle]* duplets, the most important parameter the user must configure is the *critical recovery time* (T_{crit}), which is the maximum recovery time allowed for the microprocessor. This means that if the microprocessor outputs the correct value in *expectedCycle* + T_{crit} *cycles*, the fault is classified as producing *no damage*. But if *expectedCycle* + T_{crit} + *1 clock cycles* have passed and the microprocessor has not output any value yet, the *Smart Table* classifies the fault as producing *timeout*. Note that, since the emulation stops at this moment, *timeout* can mean either that the program execution has frozen, or that the damage is so bad that the hardening technique cannot recover the correct values after T_{crit} + *1*. Since we want to relax the time restrictions of the test, but we want the output data sequence to be correct, if the microprocessor outputs a wrong value (*Output != expectedOutput*) before *expectedCycle* + T_{crit} + *1*, the fault is classified as producing *output damage*.

4. CASE STUDY

To assess the effectiveness of our infrastructure, we developed a compiler front-end and back-end for *PicoBlaze* microprocessor. This is an 8 bit soft-micro widely used in *FPGA*-based embedded systems. It supports the following main features: 16 byte-wide general-purpose data registers, 1K instructions of programmable on-chip program store, byte-wide Arithmetic Logic Unit (*ALU*) with *CARRY* and *ZERO* indicator flags and 64-byte internal scratchpad RAM. The *PicoBLaze* assembler uses the *KCPSM3* syntax [5].

The *PicoBlaze* front-end takes the original *KCPSM3* source code, performs lexical, syntactical and semantical analyses, and finally generates a *GIF* as output. It is worth noting that our *PicoBlaze* compiler front-end is a multiplatform tool that provides a very accurate error localization, compared with any other *PicoBlaze* compiler (including the official *KCPSM3* compiler).

After the hardening process, it is produced a *HGIF*, which is taken by the developed compiler back-end for *PicoBlaze*, transforming the flow back to the *KCPSM3* syntax.

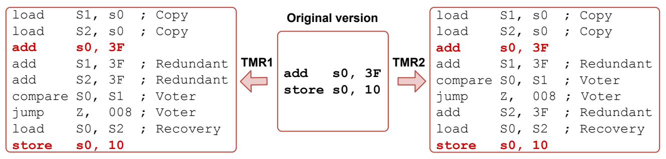
As part of the case study, we have implemented three software-based fault recovery techniques. These techniques are based on the well known Triple Modular Redundancy (*TMR*) approach. Two of them are aimed to protect specially arithmetic and logic instructions within a program, whereas the third one is an adaptation of an overall technique proposed by Reis et al. [25], the so called *SWIFT-R*.

First implemented strategy, called *TMR1*, can be summarized as follows.

1. First step is the identification of nodes (*basic blocks*) and subnodes in the program.
2. Build the flow control graph of the program.
3. Triplication of the desired instructions (in this case, arthmetic and logic instructions). Redundant instructions must operate using redundant register copies.
4. Insertion of majority voters and recovery procedures for protected registers at the following points: just before the last instruction of each node/subnode and also, just before any instruction being the destination of a jump or function call.
5. Release redundant register copies after each majority voter inserted. Once the registers value's correctness has been verified, its copies can be released.
6. Optionally, during the hardening process, additional majority voters and recovery procedures can be dynamically inserted in case there are not enough available registers to replicate. By means of this, registers copies will be released to continue with the hardening process.

Second implemented technique is called *TMR2*. It consists of detect and correct faults in the program data by computing the values twice and recomputing a third time if a discrepancy between the first two values occurs. This technique, in the same way that *TMR1*, is only applied to arithmetic and logic instructions.

Fig. 7 shows an example of the hardening of a simple program (*KCPSM3* syntax) using *TMR1* and *TMR2*.



```
load    S1, s0  ; Copy
load    S2, s0  ; Copy
add     s0, 3F
add     S1, 3F  ; Redundant
add     S2, 3F  ; Redundant
compare S0, S1  ; Voter
jump    Z,  008 ; Voter
load    S0, S2  ; Recovery
store   s0, 10
```

```
add   s0, 3F
store s0, 10
```

```
load    S1, s0  ; Copy
load    S2, s0  ; Copy
add     s0, 3F
add     S1, 3F  ; Redundant
compare S0, S1  ; Voter
jump    Z,  008 ; Voter
add     S2, 3F  ; Redundant
load    S0, S2  ; Recovery
store   s0, 10
```

Figure 7: Hardened program using *TMR1* and *TMR2*

SWIFT-R is an overall technique aimed to protect the registers values. Our adaptation of this method can be explained as follows.

1. Just like in *TMR1*, it is necessary to identify the nodes and subnodes of the program and build its flow control graph.

2. The first time that a data enters to the *SoR* must by triplicated. In this case, the *SoR* does not include the memory subsystem because it is assumed that the memory already has its own protection mechanisms [25]. So, for every instruction type classified as *inSoR* (read input ports, read from memory, load a value into a register), there will be created two additional copies. These copies always will be created using *LOAD* instructions to avoid additional memory or ports accesses.

3. Triplication of the operations using redundant copies. Replicated instructions are those ones whose type is one of the following: arithmetic, logic, shift/rotate.

4. Check the consistency of the data involved on the following instructions (by inserting majority voters and recovery procedures before execute them):

   - *outSoR* instructions. These ones cause that the data stored in a register leaves the *SoR* (store into a memory position or write into an output port).
   - Those instructions located just before a conditional branch. This verification is necessary because these instructions affect the flags and if the register value has been corrupted maybe the resultant flag state so has it, provoking a branch to an erroneous node in the flow control graph.

5. Redundant registers only can be released in the following situations:

   - If the register won't be used anymore in the program.
   - If next time the register is used is overwritten. Note this condition imply a detailed analysis to the flow control graph to avoid consistency loss.

While *TMR1* keeps register copies only until a check point occurs, *SWIFT-R* maintains those copies longer, until they are not needed anymore.

Two experiments were performed in the case study. The first one is aimed to the evaluation of the implemented techniques by means of applying the software-only techniques to a benchmark suite. On the other hand, a second experiment has been proposed to show the infrastructure possibilities in the fault-tolerant co-design field.

The benchmark suite used in the experiments is made up of the following test programs: bubble sort (*bub*), scalar division (*div*), Fibonacci (*fib*), greatest common divisor (*gcd*), matrix addition (*madd*), matrix multiplication (*mmult*), scalar multiplication (*mult*) and exponentiation (*pow*).

4.1 Evaluation of software-based techniques

Using the proposed infrastructure, it has been automatically hardened every test program applying the three software techniques. Although, these techniques provide mechanisms to protect both the control flow and the data, only the second was enabled because the memory restrictions of *PicoBlaze*. Therefore, for this case, the flags register and program counter (PC) are the firsts candidates to be hardened using hardware redundancy. Fig. 8 presents the code overhead results, whereas Fig. 9 shows the execution time overhead results, both figures normalized to a baseline with the non-hardened version.

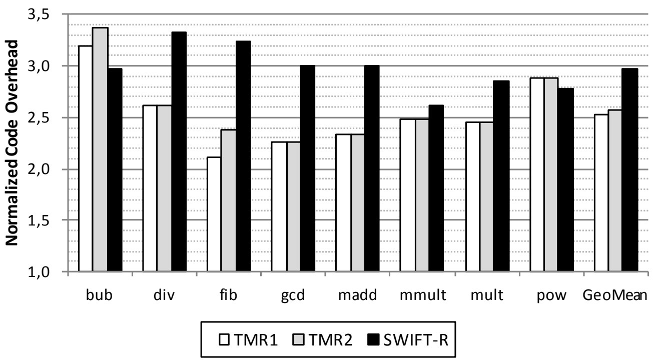


Figure 8: Normalized Code Overhead

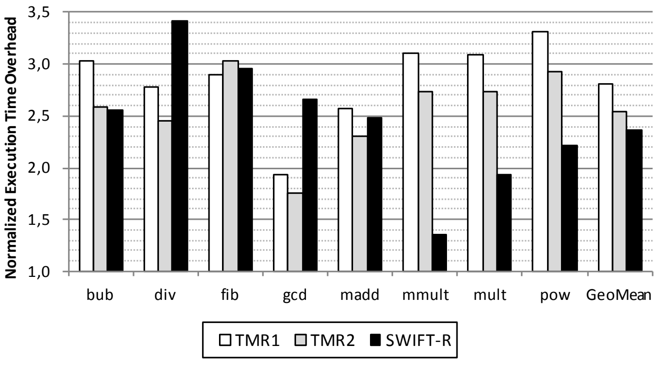


Figure 9: Normalized Execution Time Overhead

TMR1 and *TMR2* present almost the same impact over the code, the geometric mean (calculated across all benchmarks) of their normalized code overhead is $\times 2.52$ and $\times 2.58$ respectively. While *SWIFT-R* has a slightly higher code overhead of $\times 2.97$. However, regarding Fig. 9, notice that *SWIFT-R* offers the lowest impact over the execution time ($\times 2.37$), whereas *TMR1* and *TMR2* cause a performance degradation of $\times 2.81$ and $\times 2.54$ respectively. These overhead analyses can motivate important design decisions, e.g if one of the evaluated techniques provoke unsuitable overheads, according with the requirements of a specific application.

The following experimental setup has been configured to assess the reliability offered by the techniques. For each benchmark (original and hardened versions) were performed 5.200 executions in the *ISS*. According to the fault model, there was only one *SEU* simulated during each program's run. Fault was simulated by a bit-flip in a randomly selected bit from the microprocessor registers bank (16 byte-wide registers for *PicoBlaze*) during the program execution. Fig. 10 shows the fault percentages for every benchmark.

In average the *unACE* percentages are: 83.95% for non-hardened version, 85.68% for *TMR1*, 84.44% for *TMR2* and 94.77% for *SWIFT-R*. This means that these percentages of

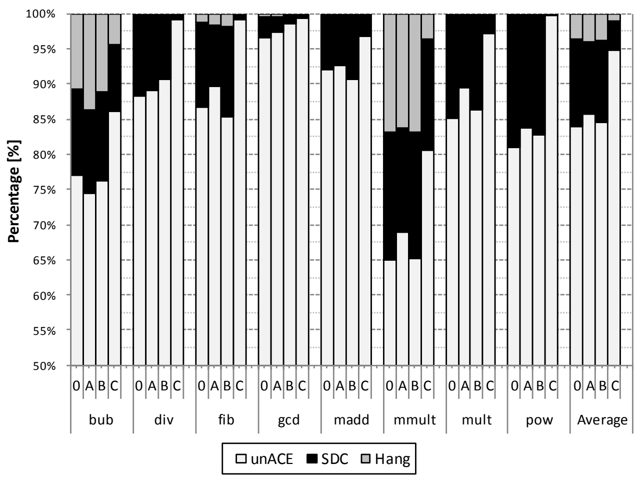


Figure 10: Fault classification percentages for non-hardened version (0), *TMR1* (A), *TMR2* (B), *SWIFT-R* (C) using the *ISS*

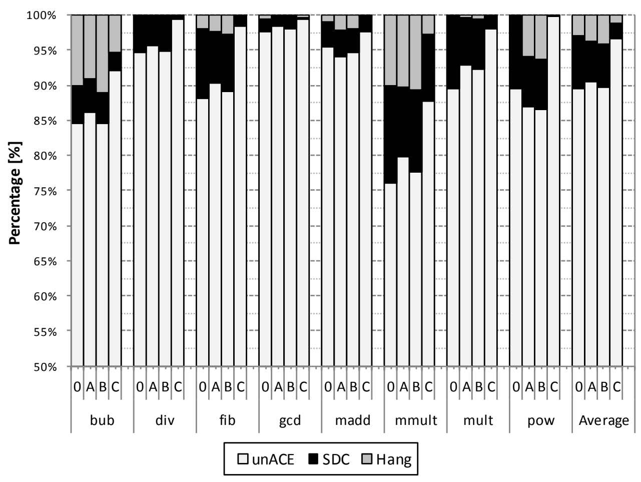


Figure 11: Fault classification percentages for non-hardened version (0), *TMR1* (A), *TMR2* (B), *SWIFT-R* (C) using the *FTUnshades*

the injected faults do not provoke any undesirable behavior and can be considered an estimation of the fault coverage offered by the techniques. Note that *TMR1* and *TMR2* do not offer a considerably fault tolerance increase, whereas *SWIFT-R* does.

Next, the *FTUnshades SEU*-emulation tool was used to evaluate the real system running all the benchmarks. The system was implemented using the official *Xilinx PicoBlaze* netlist. The fault injection campaign was equal to the used for the *ISS*: 5.200 *SEUs* injected (one per run) during a randomly selected clock cycle in a randomly selected bit of the register file. For all the test T_{crit} was defined as 20 clock cycles, giving to the microprocessor 20 more clock cycles than the usual time usual to recover the system to a fault-free state.

Fig. 11 shows the results obtained for each version of the system, by using the *FTUnshades*. Results have been classified in the same way as with the *ISS* evaluation. In this case the average fault coverage offered by the methods are: 89.50% for non-hardened version, 90.56% for *TMR1*, 89.77% for *TMR2* and 96.59% for *SWIFT-R*. As it can be seen these results confirm the hypotheses expressed during the preliminar reliability evaluation with the *ISS*. Although the *FTUnshades* percentages are slightly higher than those ones obtained with the *ISS*, they maintain the same tendencies.

Reliability results jointly with overhead results must be taken into account in the next decisions about system design, representing several trade-offs among code size, performance and reliability. For instance, if execution time is critical, it is necessary to discard some approaches, selecting those ones that meet the performance constraints, for further fault coverage improvements.

4.2 Co-design Space Exploration

Using the proposed infrastructure it is possible to follow different co-design strategies depending on the systems requirements. Generally such requirements can be directly a system constraint (silicon area, performance, power consumption,. . .) or may target any kind of fault tolerance metrics associated to a specific application (fault coverage, recovery time,. . .). Anyway, in order to demonstrate the infrastructure possibilities, an in deep analysis of the *matrix multiplication* algorithm running on *PicoBlaze* was performed.

Not only applying different software-only techniques is possible within the proposed hardening compiler, but also it is possible to apply a specific technique in a selective way. In this case study, an optimized version of the *mmult* has been written and incrementally hardened, by applying the *SWIFT-R* technique to several sets of selectively-chosen registers. For example, in the hardened version obtained with the set $0 - A - E$, only these registers are protected. Note that the sixteen *PicoBlaze* registers are numbered from 0 to F (hex).

Fig. 12 shows the overheads results for each one of the selectively hardened versions of *mmult*. These results are normalized with a baseline built with the optimized non-hardened version.

As for the benchmark suite, a similar fault injection campaign has been performed using the *ISS* to assess the reliability of each one of the selectively hardened versions of *mmult*. Results are depicted in Fig. 13.

In the same way, the hardware of the system have been implemented for all the software hardened versions. The reliability of the systems has been tested using the *FTUnshades* by means of a fault injection campaign designed using the same features as for the benchmark suite test campaign. Obtained results using *FTUnshades* are presented in Fig. 14.

Once again, comparing the reliability results maintain the same tendencies and their differences are sustained by the explanations given in Section 3.2 when the *ISS* was presented.

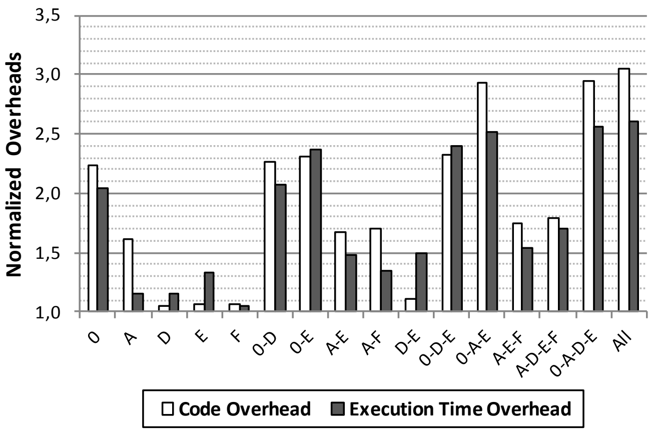


Figure 12: Normalized Code and Execution Time Overheads for *mmult* selective hardened versions

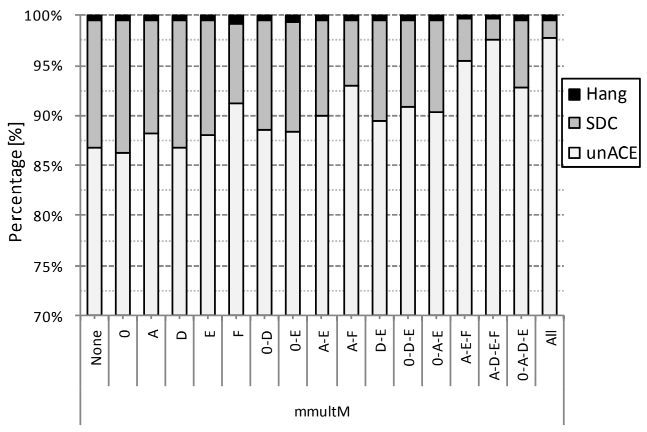


Figure 13: Fault classification percentages for *mmult* selective hardened versions using the *ISS*

Analyzing overheads results together with reliability ones is a very important key because it permits the designer to find out several possible combinations of the protections applied to the software, in order to decide which parts of the system will be hardened in this way, and which others is better to protect by means of redundant hardware. In this case, it is worth noting for example, that the hardened version provided applying *SWIFT-R* only to the register set $A-D-E-F$ would be an interesting choice from the software side, because it offers high reliability (98.18% are *unACE* faults in *FTUnshades*) and at the same time, its code and execution time overheads results are acceptable, $\times 1.79$ and $\times 1.70$ respectively. However, these decisions must be strictly connected with the requirements of every specific application.

Comparing the obtained results using the *ISS* and the *FTUnshades* may appear to be unnecessary to use the hardware *SEU*-emulator. However, it is essential to assessing the final system after applying hardware-based hardening techniques, which in this case study have not been included.

5. CONCLUSIONS AND FUTURE WORK

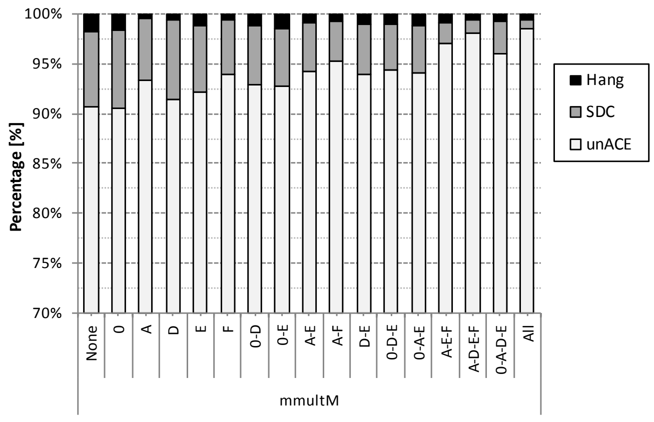


Figure 14: Fault classification percentages for *mmult* selective hardened versions using the *FTUnshades*

This paper presents a compiler–based hardening infrastructure able to lead the co-design of fault-tolerant hardware/software systems.

The overall infrastructure facilitates the exploration of the design space between hardware-only and software-only fault tolerant techniques.

As a trade–off between accuracy and flexibility we calculate the fault–coverage with both an *Instruction Set Simulator* and a *SEU* emulation tool.

The advantages of the resultant mixed hardware/sofware implementation are illustrated by means of an exhaustive case study. In this context a compiler front–end for the well-known *PicoBlaze* soft-micro have been developed, as well as a sort of fault–redundant techniques to assess the different fault–coverage when applied.

It has been depicted a novel co–design space exploration strategy by means of a selective application of the fault tolerance techniques on different microprocessor resource sets of interest (mainly general purpose registers).

As a result, this new strategy suggests the implementation of automatic co–hardening tasks within the presented platform and opens up interesting new boundaries in space exploration. As future work, the hardening development environment will be extended to support 32-bit soft-core microprocessors, such as *Xilinx's MicroBlaze* and *LEON3*.

6. ACKNOWLEDGMENTS

This work makes part of *RENASER* project (`ESP2007-65914-C03-03`) funded by the 2007 Spain Research National Plan of the Ministry of Science and Education in which context this work has been possible. The work presented here has been carried out thanks to the support of the research project 'Aceleración de algoritmos industriales y de seguridad en entornos críticos mediante hardware' (`GV/2009/098`) (Generalitat Valenciana, Spain).

7. REFERENCES

[1] T. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *32nd Annual International Symposium on Microarchitecture, (MICRO-32)*, pages 196–207, 1999. Haifa, Israel, Nov 16-18, 1999.

[2] R. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, page 121, April 2002.

[3] R. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Trans. on Device and Materials Reliability*, 5(3):305–316, Sept 2005.

[4] P. Bernardi, L. Bolzani, M. Rebaudengo, M. Reorda, F. Vargas, and M. Violante. A new hybrid fault detection technique for systems-on-a-chip. *IEEE Transactions on Computers*, 55(2):185–198, Feb 2006.

[5] K. Chapman. *PicoBlaze KCPSM3. 8-bit Micro Controller for Spartan-3, Virtex-II and Virtex-II Pro.* Xilinx Ltd., 2003. October 2003.

[6] DoD. MIL-HDBK-817, Military Handbook System Develop Radiation Hardness Assurance. Technical report, Department of Defense. USA, 1994.

[7] R. Edwards, C. Dyer, and E. Normand. Technical standard for atmospheric radiation single event effects (SEE) on avionics electronics. In *IEEE Radiation Effects Data Workshop (REDW)*, pages 1–5. IEEE, 2004.

[8] ESA. The Radiation Design Handbook ESA PSS-01-609. Technical report, European Space Agency, 1993.

[9] M. Gomaa, C. Scarbrough, T. Vjaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. *IEEE MICRO*, 23(6):76–83, Nov-Dec 2003.

[10] H. Guzman-Miranda, M. Aguirre, and J. Tombs. Noninvasive fault classification, robustness and recovery time measurement in microprocessor-type architectures subjected to radiation-induced errors. *IEEE Transactions on Instrumentation and Measurement*, 58(5), May 2009.

[11] IEC. IEC/TS 62396-1. Technical report, International Electrotechnical Commission, March 2006.

[12] A. Mahmood and E. McCluskey. Concurrent error-detection using watchdog processors - a survey. *IEEE Transactions on Computers*, 37(2):160–174, FEB 1988.

[13] MISRA. *MISRA-C:2004 Guidelines for the use of the C language in critical systems.* Motor Industry Software Reliability Association, 2004.

[14] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed design and evaluation of Redundant Multithreading alternatives. In *29th Annual International Symposium on Computer Architecture*, pages 99–110, 2002. Anchorage, AK, May 25-29, 2002.

[15] J. Napoles, H. Guzman, M. Aguirre, J. Tombs, F. Munoz, V. Baena, A. Torralba, and L. Franquelo. Radiation environment emulation for VLSI designs A low cost platform based on xilinx FPGAs. In *IEEE International Symposium on Industrial Electronics, ISIE 2007*, 2007.

[16] N. Oh, S. Mitra, and E. J. McCluskey. (EDI)-I-4: error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers*, 51(2):180–199, 2002.

[17] N. Oh, P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1), 2002.

[18] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1), 2002.

[19] T. Parr. ANTLR - ANother Tool for Language Recognition. `http://antlr.org`, Sep 2009. Internet.

[20] M. Rebaudengo, M. S. Reorda, and M. Violante. A new software-based technique for low-cost Fault-Tolerant application. *Annual Reliability and Maintainability Symposium, 2003 Proceedings*, pages 25–28, 2003.

[21] M. Rebaudengo, M. S. Reorda, and M. Violante. A new approach to software-implemented fault tolerance. *Journal of Electronic Testing-Theory and Applications*, 20(4):433–437, 2004.

[22] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. *First IEEE International Workshop on Source Code Analysis and Manipulation, Proceedings*, pages 33–42, 2001.

[23] S. Reinhardt and S. Mukherjee. Transient fault detection via simultaneous multithreading. In *27th International Symposium on Computer Architecture*, pages 25–36, 2000. Vancuver, Canada, Jun 12-14, 2000.

[24] G. Reis, J. Chang, N. Vachharajani, S. Mukherjee, R. Rangan, and D. August. Design and evaluation of hybrid fault-detection systems. In *32nd International Symposium on Computer Architecture, Proceedings*, pages 148–159, 2005. Madison, WI, Jun 04-08, 2005.

[25] G. A. Reis, J. Chang, and D. I. August. Automatic instruction-level software-only recovery. *IEEE Micro*, 27(1):36–47, 2007.

[26] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: software implemented fault tolerance. *CGO 2005: Int Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[27] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-Controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization*, Vol. V:1–28, 2005.

[28] F. Restrepo-Calle, A. Martínez-Álvarez, S. Cuenca-Asensi, F. Palomo, and M. Aguirre. Hardening development environment for embedded systems. 2010. In the 2nd HiPEAC Workshop on Design for Reliability (DFR'10) held in conjunction with The 5th Int. Conf. on High Performance and Embedded Architectures and Compilers. Pisa, Italy, Jan 25-27, 2010.

[29] P. Samudrala, J. Ramos, and S. Katkoori. Selective triple modular redundancy (stmr) based single-event upset (seu) tolerant synthesis for fpgas. *IEEE Transactions on Nuclear Science*, 51(5), Oct. 2004.

[30] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on

the soft error rate of combinational logic. In *International Conference on Dependable Systems and Networks, Proceedings*, pages 389–398, 2002.

[31] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *29th Annual International Symposium on Computer Architecture*, pages 87–98, 2002. Anchorage, AK, May 25-29, 2002.