UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

RONALDO RODRIGUES FERREIRA

The Transactional HW/SW Stack for Fault Tolerant Embedded Computing

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Prof. Dr. Luigi Carro
Advisor

Prof. Dr. Álvaro Freitas Moreira
Co-advisor

Porto Alegre, January 2015

CIP – CATALOGING-IN-PUBLICATION

Ferreira, Ronaldo Rodrigues

The Transactional HW/SW Stack for Fault Tolerant Embedded Computing / Ronaldo Rodrigues Ferreira. – Porto Alegre: PPGC da UFRGS, 2015.

129 p.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2015. Advisor: Luigi Carro; Co-advisor: Álvaro Freitas Moreira.

1. Compiler Design, Coverage, Error Detection, Error Recovery, Fault Injection, Hardening By Design, Latency, LLVM, Modular Redundancy, Register File, Rollback, Single Event Effects, Soft Error. I. Carro, Luigi. II. Moreira, Álvaro Freitas. III. Título.

ABSTRACT

Fault tolerance implementation in embedded systems is challenging because the physical constraints of area occupation, power dissipation, and energy consumption of these systems. The need for optimizing these three physical constraints while doing computation within the available performance goals and real-time deadlines creates a conundrum that is hard to solve. Classical fault tolerance solutions such as triple and dual modular redundancy are not feasible due to their high power overhead or lack of efficient and deterministic error recovery. Existing techniques, although some of them reduce the power and area overhead, incur heavy performance penalties and most of the time do not assume a feasible fault model. This dissertation introduces the Transactional HW/SW Stack, or simply Stack, to efficiently manage the area, power, fault coverage, and performance conundrum. The Stack introduces a new compilation strategy that assembles programs into Transactional Basic Blocks, together with a novel microprocessor, the TransactiOnal Basic Block Architecture (ToBBA), which provides fine-grained error detection and deterministic error rollback and elimination using the Transactional Basic Blocks (TBBs) both as a container for errors and as a small unit of data checkpointing. Two solutions to sustain the TBB semantics in hardware are introduced: software- and hardware-based. Stack's area, power, performance, and coverage were evaluated using ToBBA's hardware implementation model. The Stack attains an error correction coverage of 99.35% with 2.05 power overhead within an area overhead of 2.65. The Stack also presents a performance overhead of 1.33 or 1.54, depending on the hardware model adopted to support the TBB.

Keywords: Compiler Design, Coverage, Error Detection, Error Recovery, Fault Injection, Hardening By Design, Latency, LLVM, Modular Redundancy, Register File, Rollback, Single Event Effects, Soft Error.

Pilha HW/SW Transacional para Computação Embarcada Tolerante a Falhas

RESUMO

O desafio de implementar tolerância a falhas em sistemas embarcados advém das restrições físicas de ocupação de área, dissipação de potência e consumo de energia desses sistemas. A necessidade de otimizar essas três restrições de projeto concomitante à computação dentro dos requisitos de desempenho e de tempo-real cria um problema difícil de ser resolvido. Soluções clássicas de tolerância a falhas tais como redundância modular dupla e tripla não são factíveis devido ao alto custo em potência e a falta de um mecanismo para se recuperar erros. Apesar de algumas técnicas existentes reduzirem o *overhead* de potência e área, essas incorrem em alta degradação de desempenho e muitas vezes assumem um modelo de falhas que não é factível. Essa tese introduz a Pilha de HW/SW Transacional, ou simplesmente Pilha, para gerenciar de maneira eficiente as restrições de área, potência, cobertura de falhas e desempenho. A Pilha introduz uma nova estratégia de compilação que organiza os programas em Blocos Básicos Transacionais (BBT), juntamente com um novo processador, a Arquitetura de Blocos Básicos Transacionais (ABBT), a qual provê detecção e recuperação de erros de grão fino e determinística ao usar o BBT como um contâiner de erros e como unidade de checkpointing. Duas soluções para prover a semântica de execução do BBT em hardware são propostas, uma baseada em software e a outra em hardware. A área, potência, desempenho e cobertura de falhas foram avaliadas através do modelo de hardware do ABBT. A Pilha provê uma cobertura de falhas de 99,35%, com *overhead* de 2,05 em potência e 2,65 de área. A Pilha apresenta *overhead* de desempenho de 1,33 e 1,54, dependento do modelo de hardware usado para suportar a semântica de execução do BBT.

Palavras-chave: Compiladores, Cobertura, Detecção de Erros, Injeção de Falhas, Latência, LLVM, Redundância Modular, Banco de Registradores, Soft Error..

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ABFT	Algorithm Based Fault Tolerance
ACCE	Automatic Correction of Control flow Errors
ASIC	Application-Specific Integrated Circuit
AVF	Architectural Vulnerability Factor
BB	Basic Block
DFE	Data Flow Error
DMR	Dual Modular Redundancy
CCA	Control flow Checking Approach
CEDA	Control flow Error Detection through Assertions
CFCSS	Control Flow Checking by Software Signatures
CFE	Control Flow Error
CFG	Control Flow Graph
COTS	Commercial Off-The-Shelf
ECC	Error Correcting Code
ECCA	Enhanced CCA
FPGA	Field-Programmable Gate Array
GCC	GNU C Compiler
GPU	Graphics Processing Unit
HETA	Hybrid Error detection Technique using Assertions
HW	Hardware
ILP	Instruction Level Parallelism
IR	Intermediate Representation
ISel	Instruction Selection
ISA	Instruction Set Architecture
IVI	Instruction Vulnerability Index
JIT	Just In Time compilation

LDMR	Lightweight Dual Modular Redundancy
LICM	Loop Invariant Code Motion
LSQ	Load/Store Queue
MBU	Multiple Bit Upset
OOO	Out-of-Order
PC	Program Counter
RA	Register Allocation
RISC	Reduced Instruction Set Computing
RMT	Redundant Multi Threading
ROB	Reorder Buffer
RTL	Register Transfer Level
RVF	Register Vulnerability Factor
SDC	Silent Data Corruption
SEE	Single Event Effect
SET	Single Event Transient
SEU	Single Event Upset
SIHFT	Software-Implemented Hardware Fault Tolerance
SP	Stack Pointer
SRF	Spill Register File
SSA	Static Single Assignment Form
Stack	Transactional HW/SW Stack
SW	Software
SWAT	SoftWare Anomaly Treatment
TBB	Transactional Basic Block
TMR	Triple Modular Redundancy
ToBBA	TransactiOnal Basic Block Architecture
URISC	Ultra Reduced Instruction Set Co-processor
WRF	Working Register File
YACCA	Yet Another CCA

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Computers In The Wild

Computers fail. They always did, and always will do. But we still depend on them. Deal with it.

Computing systems are exposed to a myriad of threats during their life cycle. Design errors, intentional attacks from external sources, harsh operating environment, just to name a few. But still, users of these computing systems expect them to operate correctly with acceptable performance, and without exposing sensitive data or incurring any critical threat to them. *Dependability*, i.e., the delivery of service that it is possible to justify the trust put on it (AVIZIENIS ET AL., 2004), walks hand in hand with the design, production, and delivery of any computing system. But still, computers fail.

Embedded systems are a particular case of computing systems operating under tight physical and performance requirements such as low power dissipation, low energy consumption due to limited power source, area constraints, or even real-time requirements (MARWEDEL, 2011). Due to the connection between embedded computers and the environment they are deployed onto, natural faults become an important source of service failure. Natural faults are caused by the interaction between the computing system with natural phenomena, without any human intervention to produce the fault (AVIZIENIS ET AL., 2004).

Due to the aggressive transistor scaling industry is going forward, with predictions saying transistor size will be as low as 5.9 nm by 2026 (ITRS, 2012), two natural phenomena became of utter importance for the dependability of computing systems, especially the embedded ones deployed in harsh environments: accelerated aging and soft errors (Hamdioui et al., 2013).

Aging is a natural phenomena occurring when the device is reaching the end of its intended life cycle, leading to wear out failures such as performance degradation with its associated timing faults, and breakdown in transistor gates (Keane and Kim, 2011). Scaling transistors in such a steep speed accelerates the aging effect, because even small upsets in voltage caused by natural aging can disrupt the transistor state, taking the system faster to wear out (Moore, 2009).

Soft errors are produced by intermittent (or transient) faults, and can hardly be reproduced during system operation due to their probabilistic nature (Avizienis et al., 2004). Radiation induced soft errors are a common source of service failure in space and aeronautics applications, but transistor scaling increases the soft error rate in new technology generations (Baumann, 2005), making circuits vulnerable even at sea level. In this scenario, terrestrial systems such as automotive systems will be prone to radiation induced soft errors. Soft errors due to radiation effects can be traced down to Single Event Effects (SEE). SEE's occur when a highly energized particle hits the device, provoking current glitches. In bigger transistors, the most common SEE is the Single Event Upset (SEU), when that energized particle disrupts a memory element such as a flip-flop. However, smaller operational voltages in new transistor generations increase the susceptibility of Single Event Transients (SET) as well, when the energized particle hits logic (Petersen, 2011). Because more transistors occupy the same area, in new technology nodes there is also a higher probability of Multiple Bit Upsets (MBU), where several bits of data are flipped by the same single particle (Reed et al., 1997).

New generations of embedded computing systems cannot circumvent the scenario of high error rate, making the deployment of *fault tolerance* mandatory. Fault tolerance implementation always requires some sort of *redundancy* to achieve high **fault tolerance coverage** (Avizienis et al., 2004). However, redundancy does not come for free, and, as it is discussed in this dissertation, it creates a conundrum that cannot be easily solved by the system designer. Redundancy requires extra physical resources of power, area, and performance. A system designer can navigate in these optimization axes, but at some point s/he will either reduce physical overhead or compromise fault tolerance coverage.

Redundancy creates the following scenario and design trade-offs for power, area, coverage, and performance a system designer has to cope with:

- **Power and Area**: redundancy implies in extra components, either hardware or software ones. In fault tolerance, the usual approach is modular spatial redundancy, where distinct replicas of a hardware module have their output voted to do *fault masking*. The problem of this approach is that replicating hardware modules increases power dissipation (and area) by the same replication factor. Bottom line, although replication potentially increases coverage, replication makes power dissipation skyrocket;

- **Performance**: redundancy can also be applied in time either in hardware or software. If it is applied in software, an unit of execution (e.g., a thread) executes ahead of its replica, and at the end their results are voted. Clearly, as the delta of replication increases, the performance overhead also does. If the duplication is entirely in software, although peak power dissipation is not increased, energy consumption does because software executes for more cycles. In addition, error recovery mechanisms relying on time redundancy can jeopardize real-time behavior, a mandatory characteristic to be observed for most of embedded systems (HAMDIOUI ET AL., 2013);

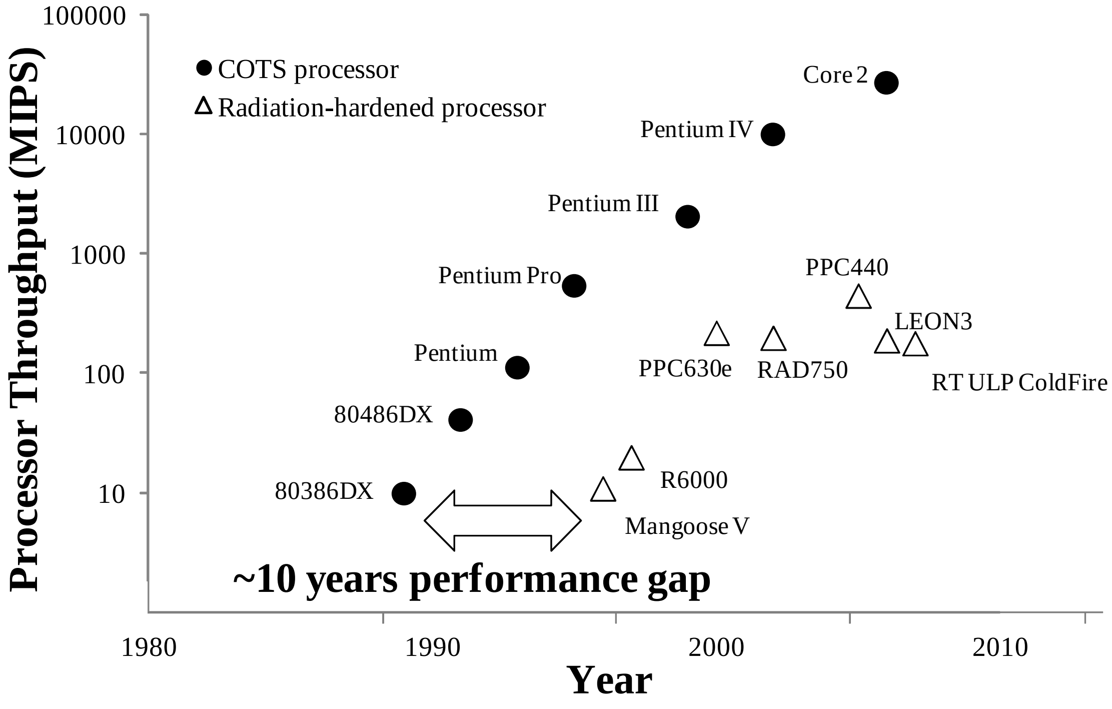
- **Coverage**: the redundancy degree can be adjusted to frame power and performance overhead within acceptable margins. However, when redundancy degree decreases, potentially, the fault coverage also does. No need to mention that a system with inadequate fault tolerance coverage cannot be considerable dependable.

In the next two sections, these physical trade-offs are discussed in more details, and the solution this dissertation proposes is briefly introduced.

1.2 The Area/Coverage/Performance/Power Conundrum

The overhead in terms of power dissipation, area occupation and performance of the additional circuitry needed to harden an embedded processor for SEE is typically far from negligible. In hardened chips targeted to low power embedded

Figure 1.1 – Performance gap between COTS (black circles) and radiation hardened microprocessors (white triangles) along decades

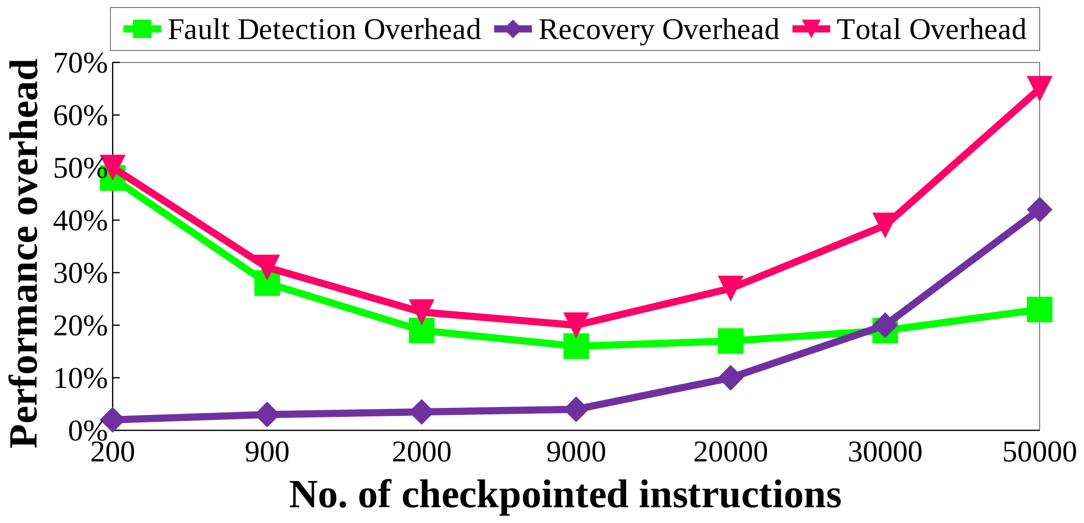


Source: Keys et al. (2008)

systems, the limited power budget and energy source will steep these chips into 'dark silicon' faster than general purpose microprocessors. *Dark silicon* (Esmaeilzadeh et al., 2011) mandates that increasing the transistors count with the same power budget will stop giving performance and energy benefits due to the end of Dennard scaling (Dennard et al., 1974). In radiation hardening, considerable area of the circuit is used to sustain reliable operation instead of performance, aggravating this scenario.

Engineers of application domains that cannot trade reliability for performance are having hard times with the low-performance offered by radiation tolerant architectures. Fig. 1.1 shows a comparison of commercial off-the-shelf (COTS) microprocessors against radiation hardened ones for performance (Keys et al., 2008). Rad-hardened microprocessors have a performance gap of approximately 10 years to their COTS counterparts at the same generation. The gap is even bigger in terms of unitary price: a 25 MHz radiation hardened RAD6000 microprocessor costs U$ 200,000 (Penix and Mehlitz, 2005), while an Intel i7 costs U$ 300. NASA launched a call for projects to design the next generation of rad-hardened

Figure 1.2 – Overhead of checkpointing an application when the number of executed instructions increase



Source: Chen and Yang (2013)

space microprocessors constructed only with COTS parts (NASA, 2013), with the main requirements being: i) high-performance and multicore parallel architecture; ii) adaptability in power consumption; iii) radiation hardened; and iv) programmability in C and compatibility with standard development and debug tools.

Due to their simplicity and generality, the usual solution for radiation hardening in circuits are the triple and dual modular redundancies (TMR and DMR, respectively) (BERNICK ET AL., 2005; MORGAN ET AL., 2007). TMR provides high error coverage, but in the dark silicon context it may not meet the power constraints due to its huge power overhead. DMR incurs in considerably less power overhead, but it has to be enhanced with some additional hardware to save periodically the architectural context to allow for error recovery instead of only error detection (BERNICK ET AL., 2005), known as 'checkpointing'.

Checkpointing is a technique used to create a consistent architectural state where the system can rollback in case an error is detected. The problem with checkpointing is that its efficiency is severely reduced depending on how many instructions are allowed to execute before the architectural state is stored. Fig. 1.2 summarizes the findings presented in Chen and Yang (2013) about the efficiency of redundant multithreading checkpointing, where two threads execute the same code and their store instructions are compared for error detection and rolled back to the last checkpoint to recover the error, which shows that in the best case scenario, the total overhead to recover the architecture to a consistent state is at least 25% of the total execution time. In critical embedded systems, which usually have some sort of real-time behavior, this high overhead may jeopardize its timing constraints.

Time redundancy for microprocessors is a DMR approach where the duplicated cores are not synchronized and one of them executes ahead of the other (ABATE ET AL., 2009). In these solutions, a large area of custom circuitry is necessary to put the cores back in a correct state in case of an error (ABATE ET AL., 2009), plus at least 200% performance and power overhead due to the time redundancy and the memory needed to checkpoint the architectural state. Time redundancy can also be applied in a single core system, by launching two identical threads with a time difference between them. This approach saves power due to the single core arrangement, but incurs in heavy performance costs.

In microprocessors, one of the most critical component in terms of reliability and peak power dissipation in the chip is the register file (BLOME ET AL., 2006). Registers are typically built with the most advanced technology transistor node available to save silicon area, and their cell node capacitance is shrunk as much as possible to increase performance. The register file is central to many existing architectural solutions designed to increase application performance, such as deep pipelining and speculation. On the other hand, these architectural constructs create a tough challenge for efficient hardening because they do not allow the precise worst case execution time computation (HAMDIOUI ET AL., 2013; WILHELM AND GRUND, 2014), not to mention the additional power required to make them work, leading to unfeasible hardening solutions when considering modern architectures in the context of constrained real-time and low power embedded systems.

The challenge that power imposes on the reliability of embedded computing has made efficient error correction a 'wishful thinking'. Research has concentrated on i) exact schemes of error detection based on the assumption of using bigger transistors for detection logic (AUSTIN, 1999); or, wherever, possible, on ii) exploiting the characteristics of the system domain to perform error correction based on accepting an error margin in the correction precision (YETIM, MARTONOSI AND MALIK, 2013). Besides missing the benefits of Moore's Law, the first approach does not apply because the old technology of today is already sensitive to radiation. The second approach is not feasible as a general solution because some domains do require an exact behavior when handling errors in computation, even though general purpose approximate architectures aimed to reduce the energy overhead wrt. exact computation exist (ESMAEILZADEH ET AL., 2012).

The discussion of this section leads to the requirements that an architecture for reliable embedded computing has to meet to be feasible:

1. The checkpointing rate and the size of checkpointed data must be reduced to a bare minimum. This reduction overcomes the performance overhead presented in Fig. 1.2 and allows the reliability mechanism to be deployed even in real-time systems;
2. The vulnerability of the register file must also be reduced as much as possible. In an ideal setting, the register file cannot be duplicated in order to avoid the increase on sensitiveness to upsets;
3. The reliability mechanism must be power efficient because of the predicted 'power wall' of 3 Watts (ITRS, 2012);
4. Performance overhead must be minimum to reduce the performance gap of radiation hardened architectures;
5. Area overhead must be minimum to mitigate (or avoid) dark-silicon;
6. Software development has to be supported by a full-fledged production level compiler.

This work shows how power, area, error coverage and performance can be reconciled in fault tolerant general purpose embedded computing. The path taken in this dissertation for that is to rethink the HW/SW Stack supported by a new compilation strategy, reduced duplication, and bare-minimum checkpointing.

1.3 Error Correction Design Space

Error correction in microprocessors is challenging to implement due to the performance and power costs involved. A fault tolerance technique for constrained systems aims to reduce power and performance overhead without compromising error coverage, usually using TMR as baseline. To understand what limits error correction and how the Stack addresses these challenges, it is necessary to consider a complex design space.

Checkpointing architectural state. Checkpointing stores the processor state periodically, creating a correct state that the architecture can rollback to if an error needs to be recovered. Because checkpointing occurs concurrently with software execution, the performance and power overheads introduced to take these architectural snapshots come from two interdependent sources: (1) the number of instructions that are allowed to commit before the architecture state is stored; and (2) the location in the architecture where the checkpoint data is stored.

The number of instructions allowed to commit before creating the architectural snapshot should be the smallest possible. This instruction window composes the *granularity* of the checkpointing method. A **fine-grain** approach corresponds to a single or a few instructions; in the opposite side of the spectrum, a **coarse-grain** approach corresponds to a thread. The granularity of the checkpointing method impacts performance due to both the error detection latency and the error recovery latency. The checkpointing granularity also impacts hardware: a fine-grain technique requires less hardened storage but additional custom hardware, while a coarse-grain one requires more hardened storage but fewer hardware modifications even none, in some cases. A complete checkpointing taxonomy is given by Prvulovic, Zhang and Torrellas (2002).

Error detection and correction latency. In the software level, fault tolerance imposes two main sources of performance overhead: (1) the additional instructions or routines executed until the fault tolerance mechanism flags that the system has an error, known as error detection latency; and (2) the additional code executed to remove the error from the system, known as error correction latency. Error detection is a periodic task that executes concomitantly with software execution, thus it impacts systems performance even in error-free execution. To mitigate error detection costs, the fault tolerance technique may be dormant for several cycles, and activated in a given period. The longer the dormant period, the higher the error detection latency will be, which implies a larger size of the checkpointing data.

Hardware support for error correction. In the hardware level, the costs to support error correction correspond to how many redundant or hardened components are included within the Sphere of Replication (SoR), i.e., the logical domain of redundant execution to support a given fault tolerant mechanism and its fault model (Reinhardt and Mukherjee, 2000). For standard DMR and TMR systems, the SoR usually encloses two and three full copies of the microprocessor, respectively. In addition to the redundant components enclosed within the SoR, it is also necessary to consider how the systems memory components are protected, i.e., the main memory and the register file. The register file is the most sensitive component for soft errors of an embedded microprocessor (Blome et al., 2006), and, as such, it is mandatory to harden it with ECC or some other error correction technique if the chip is going to be used in a mission critical application. Because the chip is inside the SoR, the power dissipation cost to protect with ECC a DMR or TMR system can correspond to up to six times the original unhardened register file (Blome et al., 2006). This cost is important because much of the work in the error correction literature considers a hardened register file without considering the costs to implement that protection.

1.4 Rethinking The HW/SW Stack

The *Transactional HW/SW Stack* (or simply, Stack) is an integrated approach between software compilation and computer architecture designed to put together power, area, error coverage, and performance in fault tolerant embedded computing.

In the hardware layer of the Stack, the *TransactiOnal Basic Block Architecture* (ToBBA) introduces a novel mechanism for doing dual modular redundancy we call *Lightweight DMR* (LDMR). In the LDMR, the microprocessor control logic and pipeline are duplicated, but the register file and remaining controllers are not. The two cores inside the LDMR share a single register file, which requires a fierce policy on how ToBBA coordinates data read and write to the register file. Still in the hardware layer, ToBBA introduces a fast and predictable mechanism for error recovery. At this point, the hardware layer crosscuts the software layer in the Stack.

In the software layer of the Stack, the *Transactional Basic Block* (TBB) revisits how program's basic blocks (BB's) are defined and generated. The TBB is an atomic unit of execution that either finishes after computing correct data or fails in case of errors. In the software level, standard basic blocks communicate through the register file and memory, being the compiler job to reduce memory communication and maximize register communication. Differently from a standard BB, a TBB only communicates with others TBBs using the memory. By eliminating register communication between TBBs, register liveness (time between the last write and a read of a register) is also eliminated. Also because the elimination of register communication, in case a TBB fails to execute, the error recovery is simply to re-execute the TBB from its start, respecting some policy on how data are written and read from the register file.

The innovation behind the Stack is how it performs error recovery based on the TBB execution and fault semantics. In this work, two proposals are evaluated: (1) the TBB holds all the data envelope necessary to correct the error; and (2) ToBBA is enhanced with auxiliary hardware, the *Spill Register File*, which will store the data envelope necessary to correct the error. Recalling the conundrum we have discussed before, proposal (1) saves hardware area and power, but incur higher performance overhead. Conversely, proposal (2) incur less performance overhead but requires additional area and power.

To summarize, the proposed Stack has the following characteristics, which will be discussed throughout this dissertation.

- **Early Error Detection**: the ToBBA architecture has two in-order RISC cores executing in loose lock-step for general purpose computation. The instructions executing in the two cores are compared against each other during their entire life cycle in the pipeline, e.g., if an error is detected as early as in the fetch state, this instruction is discarded and the error can be corrected early;

- **Tight Error Containment**: the TBB is a basic block that defines all registers it needs for computation, and it also terminates them at the end of its execution. The execution of a TBB does not share any registers with another TBB, eliminating the need for register file coherency;

- **On-Line and Efficient Error Recovery**: due to tight error containment, if a TBB finishes with no errors, that TBB is asserted as correct and committed. Otherwise, an error is detected and only the faulty TBB is re-executed without any software checkpoint;

- **Memory Correctness**: because errors are contained inside a single TBB, only the store instructions can corrupt memory. The proposed architecture guarantees that only the correct execution of store instructions are allowed to modify the memory and that the incorrect ones are discarded before they can corrupt the memory;
- **Low Power within Small Area**: ToBBA, in comparison with standard TMR and even DMR arrangements, does not duplicate the register file, leading to aggressive power and area savings. Even when it introduces the Spill Register File, there area and power savings with respect to TMR;
- **High Predictability**: because the TBB prevents the propagation of errors to other program regions, the error recovery latency in the worst case is equal to the number of instructions of the TBB where the error was detected. The latency of the proposed error recovery scheme can be computed during compilation, a much desired characteristic for real-time systems;
- **Stateless Design**: because there is no need for register file coherency and memory is correct, ToBBA becomes stateless, meaning that it can just be reset and replay the TBB execution with no need to checkpoint its not-existing internal state to do error recovery.

Experimental evaluation supports mean performance overhead of 1.54 for a MiBench (Guthaus et al., 2001) subset in proposal (1) i.e., using only the TBB as data envelope for error correction; and 1.33 in proposal (2) i.e, using the auxiliary Spill Register File as data envelope for error correction. The measured area overhead is 2.65 with respect to a single core, and the power overhead is 2.05, also with respect to a single core. The measured error correction coverage based on VHDL fault injection is 99.35%, and 99.88% of error detection coverage, leading to 0.12% of silent data corruption (SDC). These power, area and coverage results show that the Stack provides TMR-like error coverage with DMR-like hardware costs.

The measured performance overhead is considerably smaller than state-of-the-art techniques.

In summary, the contributions of this dissertation are:

- A new compilation strategy based on the elimination of live-out register-to-register communication, making the basic block an atomic unit of execution and error containment;
- The auxiliary Spill Register File, which reduces the performance overhead associated with the TBB;
- The LDMR arrangement, which is capable of executing the TBB and error recovery based on the TBB fault semantics;
- The implementation of the compilation strategy in LLVM, showing the real usage scenario and evaluation of the compilation strategy;
- Putting it all together, a new hybrid HW/SW error correction technique that does not make unfeasible assumptions about the fault model or the error detection latency.

1.5 Text Organization

This dissertation is organized as follows:

- **Chapter 2** introduces the TBB, its instruction ordering, execution semantics, and the algorithm to transform an original program into a transactional one with TBB's. That Chapter also discusses the implementation of the TBB generation in the full-fledged production level LLVM (Lattner and Adve, 2004) compiler, and how the TBB concept crosscuts the HW layer of the Stack;
- **Chapter 3** presents the LDMR, and how it is realized in the in-order ToBBA architecture. That Chapter also discusses in details the error recovery mechanism, its corner cases for the in-order ToBBA, and the comprehensive fault model ToBBA can handle. The Chapter finishes by discussing how the LDMR and ToBBA crosscuts the SW layer of the Stack;

- **Chapter 4** presents the Spill Register File (SRF), an auxiliary hardware where the live-out values of the TBBs are stored. The SRF removes the need for adding additional load and store instructions in the TBB, although it still relies in the TBB instruction ordering and semantics. This Chapter discusses the hardware implementation and how the TBB generation needs to change in order to be compatible with the SRF;

- **Chapter 5** evaluates the Stack for performance overhead, area occupation, error coverage, and error recovery latency. That Chapter sustains the claims made in this introductory Chapter and shows how the elicited requirements for fault tolerant embedded computing are met;

- **Chapter 6** compares the Stack with existing techniques in the published literature for fault tolerant computing. That Chapter shows how the Stack advances the state-of-the-art;

- **Chapter 7** concludes this dissertation, summing up the discussions made in the text, and some possible future work.

2 TRANSACTIONAL BASIC BLOCK

2.1 Definition and Instruction Formation

This section introduces the TBB definition and how the error handling and rollback mechanism works in the software layer of the Stack. The TBB algebraic description is presented in Section 2.2. The TBB definition and execution semantics is as follows:

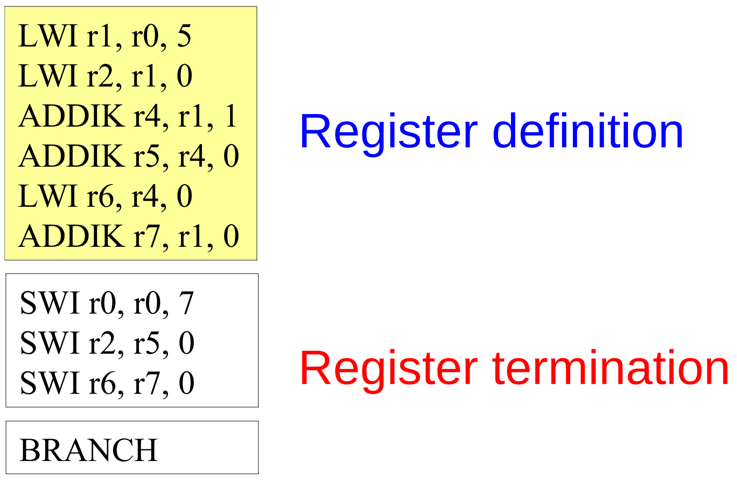
The TBB is a basic block that starts with a sequence of load, arithmetic and logic instructions for register definition*, followed by a sequence of store instructions for* register termination*, and ends with a terminator instruction finishing the TBB. The execution of a TBB is only concluded when no error is detected. If an error is detected, the Stack error handling starts the rollback mechanism to re-execute the TBB from its first instruction.*

The TBB definition contains two distinct basic block 'segments': i) register definition; and ii) register termination. Fig. 2.1 presents a TBB to illustrate these two segments using a MIPS-like ISA to facilitate the understanding.

The **register definition** segment computes the architectural register file, i.e., the data a TBB are currently operating over. This TBB segment only contains load, arithmetic, and logic instructions (Load|ALU for short). The **register termination** segment writes in the memory all data computed by a TBB that is required by another portion of the program, i.e., the 'live-out' values of the current TBB have to be stored in memory before the next TBB starts executing. The register termination segment only contains store instructions.

The TBB is an atomic unit of execution in terms that its execution either completes correctly or a rollback is triggered to correct the error. To enable the simply

Figure 2.1 – Segments of the Transactional Basic Block



re-execution of the TBB when an error is detected, if a TBB A defines a value x that is also used by TBB B, B cannot assume the register containing the value x computed by A is correct when execution reaches B, i.e., register communication between two different TBB's is not allowed.

A central concept in the TBB is 'live-out' value. A *live-out* is any value defined in a standard basic block (BB) that is later used in a different BB. In a TBB, all live-out values in registers must be spilled, i.e., stored in memory before the TBB execution ends. In this way, the spilled live-out value of a TBB has to be filled into a register before its use in any other TBB. Therefore, the transformation of a BB into a TBB involves the definition and use (def–use) chain of values in the control flow graph (CFG). Spilling all live-out values of all BB's and enforcing the instruction ordering of the register definition and termination segments define a TBB. The transformation presented in Section 2.2 formalizes these concepts.

The TBB generation requires a compiler to support it, because the ordering of the register definition and termination segments must be enforced, as well as the spilling of all BB's live-out values. In this work, we have implemented the TBB generation in the LLVM (Lattner and Adve, 2004) open-source compiler. The benefit of implementing the TBB generation in LLVM is that we can generate TBB's for any architecture LLVM supports. Because the TBB requires the software to be compiled beforehand, thus ToBBA does not offer binary compatibility for previously compiled programs. The LLVM implementation details are discussed in Section 2.3.

The TBB definition has explicit connections with the hardware layer, i.e., how

ToBBA implements error handling based on the TBB's instruction ordering. In short, the TBB defines how the register file state is constructed. Based on the register definition and termination segments and their respective instruction ordering, ToBBA selects the appropriate error handling and rollback actions depending on the segment being executed. The connections between the TBB and ToBBA are listed and briefly discussed in Section 2.4 to overview the hardware layer before going deeper inside it.

2.2 Transforming Programs into Transactions

2.2.1 Preliminaries and Definitions

The TBB generation in this work was implemented in the LLVM (Lattner and Adve, 2004) compiler, and, as such, much of the algebra and constraints discussed in this Section are due to how LLVM works and represents programs.

In brief, the transformation that takes a program with standard BB's and gives an equivalent program with TBB's operates in two points of the LLVM compiler. The first one is in the middle-end, after the source code is compiled and transformed into the LLVM's Intermediate Representation (IR). The LLVM IR is an 'almost' target-agnostic language designed to be amenable for program transformations, such as loop unrolling and vectorization [1] . More on the LLVM implementation is discussed in Section 2.3.

After the program is compiled in LLVM, a standard control-flow graph (CFG) is created. A CFG is a directed graph $C = (V, E, e_o, E_e)$ where V is the set of BB's and E is the set of possible branches between the elements of V. In any CFG, there is an *entry* basic block e_o which has no predecessor, and a set $E_e \in E$ of *exit* blocks that terminates the program (or function) with no successors (Allen, 1970).

LLVM respects the classical CFG definition of Allen (1970) with an additional constraint: the basic blocks are defined in Static Single Assignment Form (SSA). In SSA, all variables are assigned once, and it also requires that any variable use must have a prior definition of that used variable (Rosen, Wegman and Zadeck, 1988). The adoption of SSA in LLVM means that in LLVM IR there is an infinite

[1]For a discussion about the target-independence in LLVM IR, readers should check `http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-October/043719.html`

number of registers, being the job of the *register allocator* to lower SSA into the available registers of the architecture.

The TBB generation requires a LLVM IR without ϕ nodes. ϕ nodes are used in SSA to assign a value depending on the predecessor of the current BB. Let us say that in the current BB, there's a variable x that is assigned the variable y. However, the value of y depends on which block is the predecessor of the current BB, because the predecessors also assign a value to y. In SSA, a data-flow statement that depends on control-flow resolution is represented as $x \leftarrow \phi(y_1, y_2)$, where y_1 and y_2 are the y instances created on each predecessor BB. These ϕ nodes can be removed in the IR level, because they do not exist in the final generated assembly code anyway. In this work, we just assume they were removed. In LLVM, ϕ removal is accomplished with the auxiliary function **DemotePHIToStack** [2].

Lastly, it is necessary to define a *program*. A program $P = \langle F, G \rangle$ is a tuple where F is the set of *functions* of P and G is the set of *global variables* of P. Any function $f \in F$ can read and write any $g \in G$. This definition slightly differs from LLVM, where it would correspond to a 'module'.

The program transformation encompasses three steps that need to be executed in that given order. The first step is to promote the return values of all functions into a global variable. This value promotion is necessary to spill the return value. Otherwise, that return value would reside only in the registers. This first step is presented in Section 2.2.2. The second step is to split basic blocks with function calls in such a way that the call instruction becomes the terminator of the new basic block. Notice that this step modifies the CFG. This step is given in Section 2.2.3. The third step is the actual live-out spilling briefly discussed before, which is given in Section 2.2.4.

2.2.2 Promoting Functions' Return Value Into Global Variable

Most of microprocessors store a function's return value into dedicated registers. For instance, in MIPS registers **$v0** and **$v1** are dedicated to temporarily store the return values produced when a function returns. However, preserving a register from the called function to the callee creates a register communication between these two

[2] `http://llvm.org/docs/doxygen/html/DemoteRegToStack_8cpp_source.html#l00110`

functions, which by the TBB definition is not allowed.

The solution in the TBB generation is simply to spill the return value of the called function into a global variable of the same type as the function. In that way, any other TBB in the program can fill a register with that global variable whenever it needs to access that return value. In the next, the function $Type(value)$ gives the return type of the argument if it is a function, and the type if it is a variable.

Algorithm 2.1: PromoteFuncRetToGlobalVar

```
Data: $P = \langle F, G \rangle$
Result: $P = \langle F, G' \rangle$ where $G'$ contains the new spill global return variables.
1 begin
2     $G' \leftarrow G$
3     forall the $f_k \in F$ do
4         if $Type(f_k) = void$ then
5             skip
6         Create new global variable $g_k$ where
7         $Type(g_k) = Type(f_k)$
8         $g_k \leftarrow$ undef
9         $G' \leftarrow g_k \cup G'$
```

There are two points that need to be clarified in Algorithm 2.1. Line 4 skips the iteration if the return type of the function f_k is 'void'. Void functions do not return value, thus, there is nothing to spill, and, as such, it is not necessary to create a global variable for f_k. Line 8 initializes the new global variable g_k with the 'undef' value. In LLVM, an undef value says to the compiler that the program is well-defined no matter the value an undef variable is assigned. Assign undef to g_k is correct because this global variable is going to store the return value of f_k, thus before f_k returns g_k will contain its correct value.

2.2.3 Splitting Basic Blocks Around Call Sites

After a global variable g is created for each non-void function f of the program P, it is now possible to spill the return value of f to g. The first step in the TBB generation is to schedule each function call in an unique BB. This is simply done by splitting every BB with 'call' instructions around the call, creating a new BB to hold the necessary register fills to hold the function's arguments before the call instruction, which terminates the BB. It is also necessary to spill the call's return value.

The rationale behind this algorithm is the following. Microprocessors pass arguments to functions through the use of dedicated registers for that, thus, there's register communication also between the called and the callee functions. However, if the callee BB were only to have load instructions to fill those arguments into the dedicated registers, after the call instruction is committed, the execution flow goes to the entry block of the called function. In the microprocessor level, because of how TBB is defined, the instructions executed *before* the entry block are only register definition ones. Thus, we might consider them as being part of the entry block of the called function. This makes the function call to respect the TBB.

The other aspect to be considered is when the called function returns. Recall that the called function has to spill its return value after the callee's call instruction. The last instruction executed in any of the exit blocks of the called function mandatorily is a return, a terminator one. The callee BB already finished executing, because its last instruction was the call to the called function. Therefore, the callee BB will fall through to the next BB. If this next BB were to use the return value of the called instruction, it now can just fill a register with the respective spill variable holding the value it needs. Thus, the return value also respects the TBB definition.

In the following, we use the LLVM type 'CallInst' to represent the set of call instructions existing in a function. We also use a function $Parent(value)$ that returns the BB a instruction is member of, and the function $Next(value)$, which gets the next instruction of the instruction passed as argument in the BB $Parent(value)$. The BB splitting is done with the LLVM auxiliary function **SplitBlock** [3]. The **SplitBlock** function takes as arguments the BB to be split, and the instruction where the BB will be split around. This function creates two BB's as follows. Let us say that a BB b will be split around its instruction i_k, and that $b = i_1 \dots i_{k-1}\ i_k \dots i_n$. Calling $SplitBlock(b, i_k)$ produces the BB's $b_{before} = i_1 \dots i_{k-1}$, and $b = i_k \dots i_n$. It also creates one unconditional branch $b_{before} \to b$ connecting these BB's. **SplitBlock** returns b. Therefore, **SplitBlock** modifies the CFG of the program P.

The algorithm to split the BB's in order to leave all function calls inside its own BB is given in Algorithm 2.2.

[3]`http://llvm.org/docs/doxygen/html/BasicBlockUtils_8cpp_source.html#l00273`

Algorithm 2.2: SplitBBAroundCallSites

```
   Data: $P = \langle F, G \rangle$ and the CFG $C_P$ of the program $P$.
   Result: $C'_P$ with the split BB's around the 'call' instructions.
1  begin
2      forall the $f_k \in F$ do
3          forall the $c_l \in CallInst\ of\ f_k$ do
4              $b \leftarrow SplitBlock(Parent(c_l), c_l)$
5              $c_{l+1} \leftarrow Next(c_l)$
6              if $c_{l+1}$ is not a terminator function then
7                  $SplitBlock(b, c_{l+1})$
8              Let $f_c$ be the called function of $c_l$
9              if $Type(f_c) = void$ then
10                 skip
11             Get $g_k \in G$ of $f_k$
12             Create a 'store' instruction after $c_l$ spilling the return value of $f_c$
```

Several details in Algorithm 2.2 need to be clarified. Line 4 splits the BB $Parent(c_l)$ into two blocks around the c_l instruction. However, it is still possible to have more call instructions in the BB from which c_l is member. To solve that, Line 7 checks if the next instruction c_{l+1} is a terminator one. If c_{l+1} is a terminator, it can be a branch, a return instruction or a call instruction. If it is a call instruction, in the next iteration of the outer for loop, the BB containing that call will be split, scheduling these two consecutive call instructions in two different BB's. If c_{l+1} is a branch or return instruction, the BB is correct, and nothing needs to be done. However, if c_{l+1} is not a terminator instruction, than it is necessary to split the BB again. Otherwise, the c_l instruction would not be the terminator in the $Parent(c_l)$ BB. The second call to **SplitBlock** in line 7 splits the block again, guaranteeing that c_l is the terminator instruction.

After the BB's are split around the 'call' instructions, the last step is to spill the return value of the function called by c_l. Actually, Algorithm 2.2 transformed the value c_l into a *live-out* of the BB $Parent(c_l)$. The live-out spilling is handled in the next section. Recall that the called function f_c when finishes executing stored its return value into dedicated registers for that. These registers are alive in f_k and must be spilled. Line 12 does exactly that.

Now that all function return values were spilled into global variables, these return values can now be referenced in an orthogonal fashion as any program variable.

2.2.4 Spilling Live-Out Values

The last step in the TBB generation is to spill all live-out values in the def–use chain, eliminating every register communication in the program P. In short, for every definition in the def–use chain, there must be a 'store' instruction spilling that definition, and for every use in the chain, there must be a 'load' instruction filling a register when that use is needed. Recall that Algorithm 2.2 had already spilled the return values of all functions of P. In this case, it is just necessary to reload them when it appears as a use in the def–use chain.

In LLVM, spilled values are stored in the function's stack (not to be confused with the Transactional Stack). Stack values are represented by 'alloca' instructions [4], which are typed pointers to the memory location where the actual value is. Thus, for all spilled live-out value there must be an associated alloca in the stack. The exception are the functions' return values spilled to global variables, which have their own allocation in the program memory space.

In the following, a function $f \in F$ is treated as a set of BB's to simplify the notation. Thus, $|f|$ is the number of BB's in f, and $f = \{b_1, \ldots, b_{|f|}\}$. A BB $b \in f$ will be treated as a set of instructions, thus, $|b|$ is the number of instructions in b, and $b = \{i_1, \ldots, i_{|b|}\}$.

The algorithm to spill all live-out values in a program is given in Algorithm 2.3.

Algorithm 2.3 makes some subtle decisions that need to be clarified in details for its correct understanding. Line 5 references a value v_{def} defined by some instruction i. This assertion creates a constraint on what kind of instruction i can be: it has to define a value, such as arithmetic and logic instructions. Store, function calls, memory dereferencing, just to name a few, are excluded.

Next, line 6 introduces the variable i_v, which is a value in the function v that *uses* the value v_{def}. In this moment, the def–use chain of f for the value v_{def} is being iterated.

Line 7 checks if the instruction i_v is a load. In case it is, there's nothing to do because the value v_{def} is already being filled by i_v. Therefore, if the value v_{def} happens to be a live-out, it will be detected when the value i_v has its def–use chain iterated later.

[4]`http://llvm.org/docs/LangRef.html#alloca-instruction`

Algorithm 2.3: SpillAndFillDefUseChain

```
   Data: $P = \langle F, G \rangle$
   Result: $P = \langle F', G \rangle$ where the BB's of all $f \in F'$ are TBB's.
1  begin
2      forall the $f \in F$ do
3          forall the $b \in f$ do
4              forall the $i \in b$ do
5                  Let $v_{def}$ be the value defined by $i$
6                  forall the $Instructions\ i_v\ in\ f\ using\ v_{def}$ do
7                      if $i_v$ is load then $v_{def}$ is already being reloaded by $i_v$
8                          skip
9                      if $Parent(i_v) \neq b$ then $v_{def}$ is a live-out of $b$
10                         Let $Slot_v$ be the stack slot of $v_{def}$
11                         $Slot_v \leftarrow$ null
12                         if $i$ is load then $v_{def}$ already has a stack slot
13                             $Slot_v \leftarrow$ Get existing stack slot of $v_{def}$
14                         else $v_{def}$ does not have a stack slot yet
15                             $Slot_v \leftarrow$ Create new stack slot for $v_{def}$
                           /* Fill and spill $v_{def}$ on its def-use chain   */
16                         Create load instruction before any store in $Parent(i_v)$
                           to fill $Slot_v$
17                         Create store instruction before terminator in $b$ to spill
                           $Slot_v$
```

The live-out detection is done in line 9, which simply checks if the instruction i_v is inside the same BB as i. In case that assertion is true, v_{def} is a live-out of b, i.e., that value is used outside its parent BB b. Being a live-out, v_{def} must be spilled inside its parent, and filled right before its use in the BB that uses it, $Parent(i_v)$ in this case. Line 16 inserts a load instruction in $Parent(i_v)$ to do the fill, and line 17 inserts a store instruction in b to do the spill. Notice that the spilled live-out is in the scope of f, thus it resides in the f stack, a *slot* in LLVM terminology as shown in line 11.

The TBB instruction ordering is enforced in line 16 and in line 17. In these lines, the load instruction is created in the register definition segment, i.e., before any store instruction, and the store instruction is created in the register termination segment, i.e., right before the terminator instruction.

One implementation issue worth discussing is the enforcing of the load and store ordering in the generated code. In LLVM, after the IR is generated, it is necessary to lower it into machine code, which is the last step in code generation. This step is implemented in the compiler *back-end*. The first step in code generation is Instruction Selection (ISel), which takes the LLVM IR and generates an equivalent machine code to the target architecture based on pattern matching. The second step is Register Allocation, which lowers the infinite LLVM virtual registers into the available ones in the register file. These two code generation steps may remove and create new instructions, change their relative other, etc. In LLVM, however, it is possible to force that memory-related instructions cannot move or be removed. This is achieved by marking all load and store as ‘volatile’ [5].

2.2.5 TBB Generation Example and Further Discussion

In this section, we will generate the TBB for a small program to illustrate the generation algorithms presented so far before discussing in Section 2.3 how these algorithms were implemented in LLVM. The code shown in this section is the actual LLVM IR generated with the current implementation of the TBB generation. As it will be discussed, there’s a lot of space for optimizing the TBB code.

The source code used in this example is the following small C program.

```
#include <stdio.h>

int func1(int arg);
int func2(int arg);
void func3(int arg);

int main() {
  int a = 10;
  int b = 9;
  int c = 0;
  if (a < b) {
    c = func1(b - a);
  } else {
    c = func2(func1(a + b));
  }
  func3(c);
  return c;
}

int func1(int arg){
  printf("arg func1: %d\n", arg);
```

[5]http://llvm.org/docs/LangRef.html#volatile-memory-accesses

```
  return arg * 2;
}
int func2(int arg){
  printf("arg␣func2:␣%d\n", arg);
  return arg + 5;
}
void func3(int arg){
  printf("arg␣func3:␣%d\n", arg);
  printf("%d\n", arg);
}
```

The program's CFG as generated with LLVM is shown in Figure 2.2. Notice that the BB's are written in LLVM IR.

The standard CFG has four named BB's. The 'entry' one is the CFG entry block, which is responsible for initializing the function stack (in this case, Figure 2.2 shows the CFG of the 'main' function). Stack initialization is done with the 'alloca' instruction in LLVM, as mentioned before. Notice that the 'alloca' only reserves the stack slot and does not initialize the variable being allocated, thus the preceding store instructions are necessary.

The 'call' instruction does the actual function calling. For instance, in the 'if.else' BB, there are two call's, one for the *func1* and the other for *func2*. Finally, the program ends in the exit block named 'if.end' in the example.

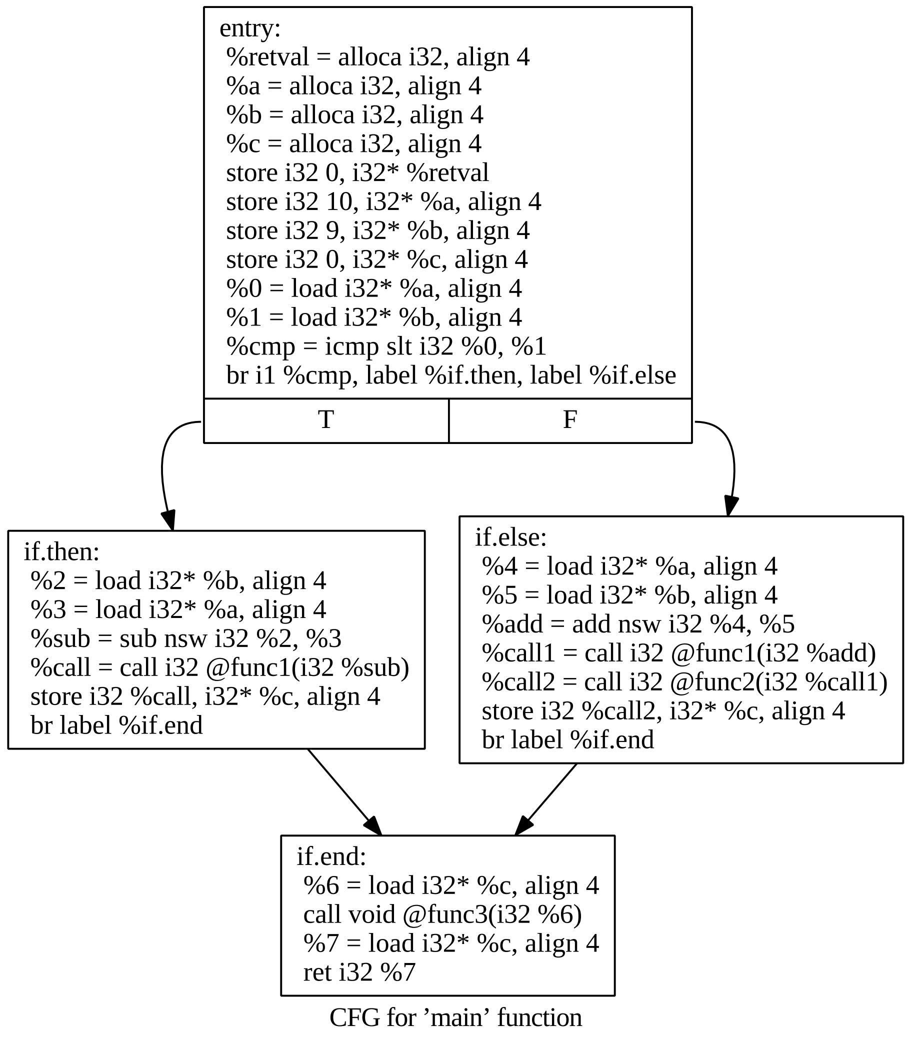
The first step is to split all BB's around the call instructions so that every call is placed inside their own BB. Figure 2.3 shows the results of applying Algorithm 2.2 in the 'if.else' BB shown in Figure 2.2. This first step is required to force that all function return values become a live-out, which can be spilled and filled later.

With all the function return values transformed into live-out values, the TBB's can be generated by executing the live-out spiller given in Algorithm 2.3. The sample program transformed into TBB's is shown in Figure 2.4.

In this sample program, program variables a and b are live-outs of the BB 'entry'. As such, they are filled in the blocks 'if.then' and 'if.else'. One example of the forced spill and fill is the IR variable **%add** defined in the BB 'if.else'. **%add** is a live-out used in the block 'if.else.split'. Therefore, this variable is spilled in the block that defines it (store volatile instruction in the 'if.else' BB), and filled in the block that uses it (load volatile in the 'if.else.split' BB).

In the current implementation, some useless spills are created, and they will be

Figure 2.2 – Control-Flow Graph of the sample program using standard Basic Blocks



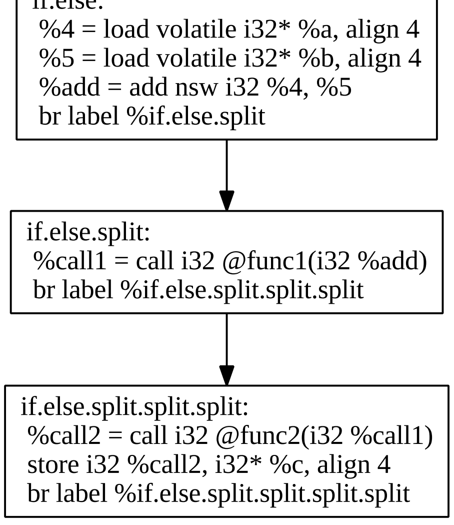
```
entry:
 %retval = alloca i32, align 4
 %a = alloca i32, align 4
 %b = alloca i32, align 4
 %c = alloca i32, align 4
 store i32 0, i32* %retval
 store i32 10, i32* %a, align 4
 store i32 9, i32* %b, align 4
 store i32 0, i32* %c, align 4
 %0 = load i32* %a, align 4
 %1 = load i32* %b, align 4
 %cmp = icmp slt i32 %0, %1
 br i1 %cmp, label %if.then, label %if.else
```

```
if.then:
 %2 = load i32* %b, align 4
 %3 = load i32* %a, align 4
 %sub = sub nsw i32 %2, %3
 %call = call i32 @func1(i32 %sub)
 store i32 %call, i32* %c, align 4
 br label %if.end
```

```
if.else:
 %4 = load i32* %a, align 4
 %5 = load i32* %b, align 4
 %add = add nsw i32 %4, %5
 %call1 = call i32 @func1(i32 %add)
 %call2 = call i32 @func2(i32 %call1)
 store i32 %call2, i32* %c, align 4
 br label %if.end
```

```
if.end:
 %6 = load i32* %c, align 4
 call void @func3(i32 %6)
 %7 = load i32* %c, align 4
 ret i32 %7
```

Figure 2.3 – Control-Flow Graph of the BB's performing function calls after Algorithm 2.2 is executed in the original 'if.else' BB shown in Figure 2.2

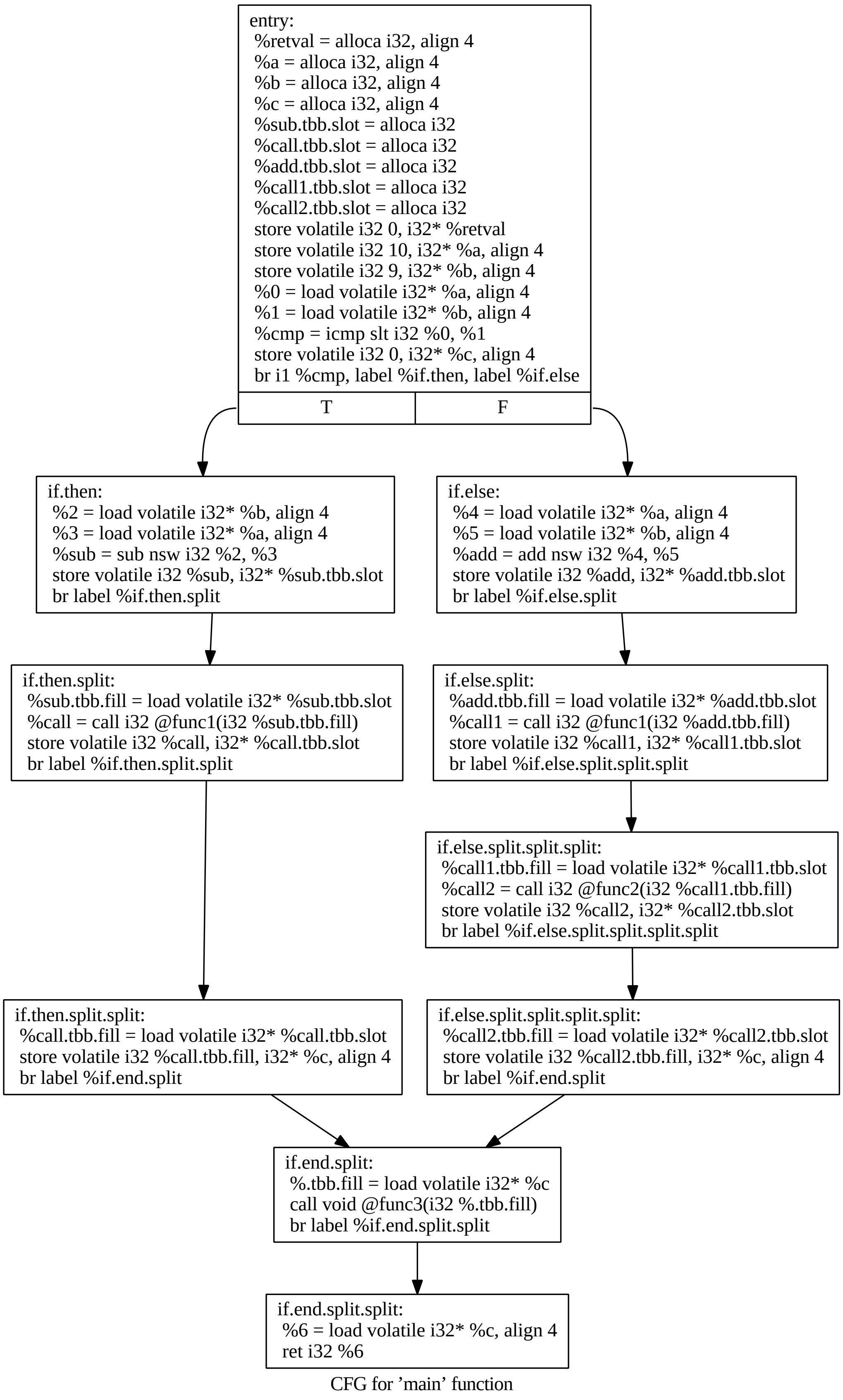


```
if.else:
 %4 = load volatile i32* %a, align 4
 %5 = load volatile i32* %b, align 4
 %add = add nsw i32 %4, %5
 br label %if.else.split
```

```
if.else.split:
 %call1 = call i32 @func1(i32 %add)
 br label %if.else.split.split.split
```

```
if.else.split.split.split:
 %call2 = call i32 @func2(i32 %call1)
 store i32 %call2, i32* %c, align 4
 br label %if.else.split.split.split.split
```

Figure 2.4 – Control-Flow Graph of the sample program using TBB's



```
entry:
 %retval = alloca i32, align 4
 %a = alloca i32, align 4
 %b = alloca i32, align 4
 %c = alloca i32, align 4
 %sub.tbb.slot = alloca i32
 %call.tbb.slot = alloca i32
 %add.tbb.slot = alloca i32
 %call1.tbb.slot = alloca i32
 %call2.tbb.slot = alloca i32
 store volatile i32 0, i32* %retval
 store volatile i32 10, i32* %a, align 4
 store volatile i32 9, i32* %b, align 4
 %0 = load volatile i32* %a, align 4
 %1 = load volatile i32* %b, align 4
 %cmp = icmp slt i32 %0, %1
 store volatile i32 0, i32* %c, align 4
 br i1 %cmp, label %if.then, label %if.else
T | F
```

```
if.then:
 %2 = load volatile i32* %b, align 4
 %3 = load volatile i32* %a, align 4
 %sub = sub nsw i32 %2, %3
 store volatile i32 %sub, i32* %sub.tbb.slot
 br label %if.then.split
```

```
if.else:
 %4 = load volatile i32* %a, align 4
 %5 = load volatile i32* %b, align 4
 %add = add nsw i32 %4, %5
 store volatile i32 %add, i32* %add.tbb.slot
 br label %if.else.split
```

```
if.then.split:
 %sub.tbb.fill = load volatile i32* %sub.tbb.slot
 %call = call i32 @func1(i32 %sub.tbb.fill)
 store volatile i32 %call, i32* %call.tbb.slot
 br label %if.then.split.split
```

```
if.else.split:
 %add.tbb.fill = load volatile i32* %add.tbb.slot
 %call1 = call i32 @func1(i32 %add.tbb.fill)
 store volatile i32 %call1, i32* %call1.tbb.slot
 br label %if.else.split.split.split
```

```
if.else.split.split.split:
 %call1.tbb.fill = load volatile i32* %call1.tbb.slot
 %call2 = call i32 @func2(i32 %call1.tbb.fill)
 store volatile i32 %call2, i32* %call2.tbb.slot
 br label %if.else.split.split.split.split
```

```
if.then.split.split:
 %call.tbb.fill = load volatile i32* %call.tbb.slot
 store volatile i32 %call.tbb.fill, i32* %c, align 4
 br label %if.end.split
```

```
if.else.split.split.split.split:
 %call2.tbb.fill = load volatile i32* %call2.tbb.slot
 store volatile i32 %call2.tbb.fill, i32* %c, align 4
 br label %if.end.split
```

```
if.end.split:
 %.tbb.fill = load volatile i32* %c
 call void @func3(i32 %.tbb.fill)
 br label %if.end.split.split
```

```
if.end.split.split:
 %6 = load volatile i32* %c, align 4
 ret i32 %6
```

CFG for 'main' function

removed in the final implementation. For instance, the blocks 'if.then.split.split' and 'if.else.split.split.split.split' only fill the return values of the functions called in their successors and spill them into the **%c** IR variable. Clearly, the return value could be spilled directly into the **%c**'s stack slot right after the function call instruction. Another useless fill is in the exit block 'if.end.split.split', which fills the IR variable **%c** and returns it. This fill could be removed and use the **%c** fill inside the block 'if.end.split'.

Another characteristic of the TBB spills and fills is that both the load and store instructions use a *constant* memory reference, i.e., they directly operate from a stack slot. Static memory references are a key point where the TBB can be optimized at the architecture level, because they are amenable to load/store value prediction (Lipasti, Wilkerson and Shen, 1996) due to their high (constant, in fact) value locality. This optimization is the main future work of this dissertation, and as such, it will be discussed in details in Chapter 7

2.3 Compilation Flow in the LLVM Framework

In LLVM, the **front-end** implements the language parser, which builds an abstract syntax tree and generates basic LLVM IR for that program. The goal of the front-end is to provide a reasonable working version of the program in LLVM IR. The optimizations on the LLVM IR happens in the Target Independent Optimizer (sometimes also referred as *middle-end*).

The middle-end analyses and transforms programs using *passes*, which can operate over three levels: modules, functions, and basic blocks. The TBB generation was implemented as a **FunctionPass**, in the LLVM notation. A function pass can inspect individual functions of the program and change their CFG, adding, removing and modifying BB's whenever needed.

Because function passes can modify the function's CFG and instructions, a pass that will be used in the future can invalidate previous passes. Therefore, in the LLVM compilation flow, the TBB generation **must be the last pass** applied over the program, which is easily accomplished with the **llc** LLVM tool. In the LLVM compilation flow, firstly the source code is compiled in the front-end, and than the generated LLVM IR is further optimized with **llc**. This works uses the **clang**

C/C++ LLVM front-end.

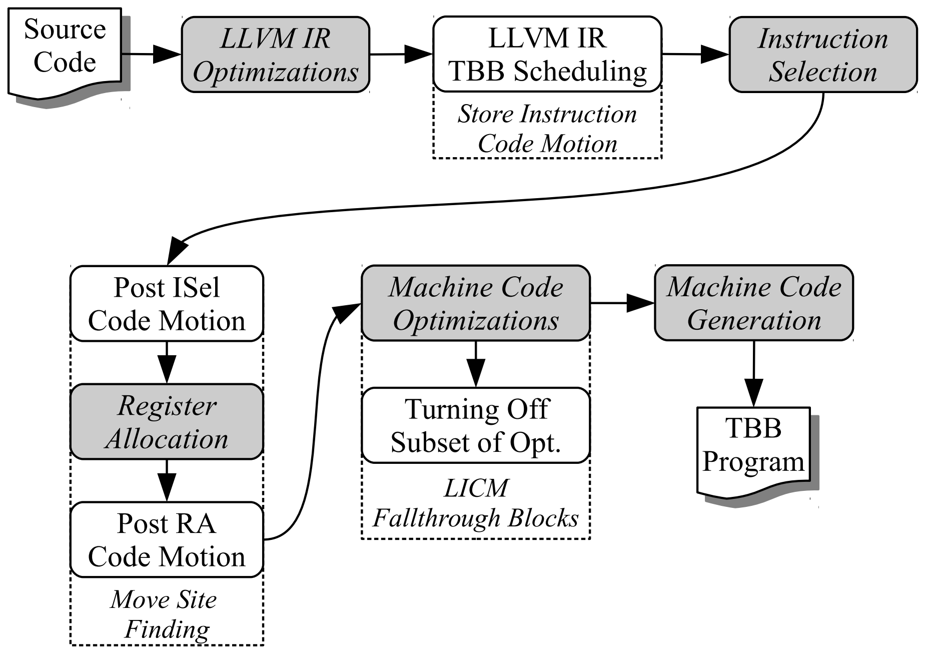
The last step in the LLVM compilation flow is the generation of the machine code by the Target Independent Code Generator (or *back-end*). The back-end lowers the LLVM IR, which is almost target independent, into machine code. Three steps in the machine code generation are important to the TBB generation: Instruction Selection (ISel), Register Allocation (RA), and Machine Code Optimization.

ISel lowers the LLVM IR into SSA-based machine code, thus it replaces LLVM IR by specific instructions of the target. This step potentially breaks the TBB instruction ordering by inserting new instructions. For example, we have observed that when a store instruction needs to operate over a constant, the constant is usually built with an **add** instruction, which is scheduled right before the store. ISel lowers the LLVM IR into a representation with some machine code but still without allocated registers (in fact, some registers might be allocated for some instructions that require the use of specific registers of the target architecture). To put the code created by ISel back into a TBB scheduling, a simple code motion step is implemented. For all basic blocks after ISel finishes, find the first store instruction of the basic block. We call this instruction move site. For all non-store instructions that are not function calls or branches placed after the move site, move them right before the move site. This movement guarantees that the basic block is a TBB.

The second step that breaks the TBB scheduling is the *Register Allocation*. In RA, it might be necessary to add load and store instructions in a basic block due to an insufficient number of available physical registers to allocate that block. The block is transformed back to a TBB by performing the same code motion we execute after ISel. This works because any store instruction added in the block has to be placed after the definition of the register it spills by RA. Thus, changing the order of these instructions does not break the register allocation.

The third and final step that might break the TBB scheduling is *machine code optimization*. Machine code optimization is the last optimization stage in LLVM right before functions' epilogue and prologue computation and the final assembly code emission. At this stage, the low-level machine code representation in LLVM is optimized to take into account intrinsics of the hardware architecture. This step is very important to produce high performance code. However, the machine code

Figure 2.5 – Algorithm for generating TBB programs in LLVM. Gray shapes represent standard LLVM's steps.



optimizations employed are aggressive, and they can change instruction ordering, breaking the TBB. In LLVM, two machine code optimizations break the TBB instruction ordering: (1) removal of unconditional branches; and (2) loop-invariant code motion (LICM). Optimization (1) removes the branch instruction of the basic block, merging it with the block that is the target of the unconditional branch. LICM (2) might place non-store instructions inside the register termination segment, which cannot be moved without breaking the register allocation. Therefore, these two optimizations must be turned off.

Figure 2.5 summarizes all the steps discussed so far in a flowchart. Gray shapes are compilation steps unaltered in the LLVMs compilation pipeline, emphasizing where in LLVM each TBB generation step is implemented.

The algorithm presented in Figure 2.5 is also valid for the Spill Register File (SRF) introduced in Chapter 4. In the SRF, the only modification is how the TBB is generated, thus the 'LLVM IR TBB Scheduling' changes. The remaining steps are still necessary in the SRF scenario because the register definition and register termination regions are still required to be enforced before final assembly code emission.

2.4 Crosscutting the HW Layer

Several concepts of the ToBBA architecture that will be discussed in Chapter 3 depend on the concepts and semantics of the TBB. In the following, these crosscutting concepts between the SW and HW layers of the Stack are discussed to facilitate whenever it is necessary to refer back to this Chapter.

- **Register Liveness**: by spilling all live-out values of the program, the TBB eliminates register liveness between BB's. The Register Vulnerability Factor (RVF) (Yan and Zhang, 2005), i.e., the probability that soft errors escape register file boundaries, is reduced to 0;

- **Error Handling and Rollback**: TBB's instruction ordering and the two register segments it defines (definition and termination) dictate how error handling is performed and how the error rollback machinery must operate. In short, these two register segments lead to two error handling scenarios, each one with a respective rollback policy to recover and eliminate the error;

- **Error Recovery Latency**: the TBB creates a container from which an error cannot escape because of live-out spilling (and the elimination of RVF). In the worst error scenario, the error is detected in the last instruction of the TBB and the rollback must re-dispatch all instructions again. Thus, the TBB creates tight bounds on error recovery latency;

- **Checkpointing Elimination**: as discussed in Chapter 1, checkpointing is the periodical storing of architectural state (and potentially, the memory as well) to create a correct previous state that the execution can rollback to in case an error is detected. In the TBB, because the error is contained inside it and does not propagate because of live-out spilling, the only data required to rollback the execution is the start address of the TBB currently under execution. The TBB boils the required data to restore the architecture to a correct state down to 32 bits only.

3 TRANSACTIONAL CORE

3.1 TransactiOnal Basic Block Architecture

The block diagram of the ToBBA architecture is depicted in Fig. 3.1. ToBBA is composed of two five-stage RISC Harvard cores with separated data and instruction memories. The main memory has a read/write port and a read-only port. The two inner RISC cores execute in lockstep, with one core being the master and the other the slave. ToBBA has a single register file shared among the two RISC cores. The RISC cores currently used in the ToBBA architecture are the open-source implementation of the MicroBlaze architecture (Kranenburg and Leuken, 2010).

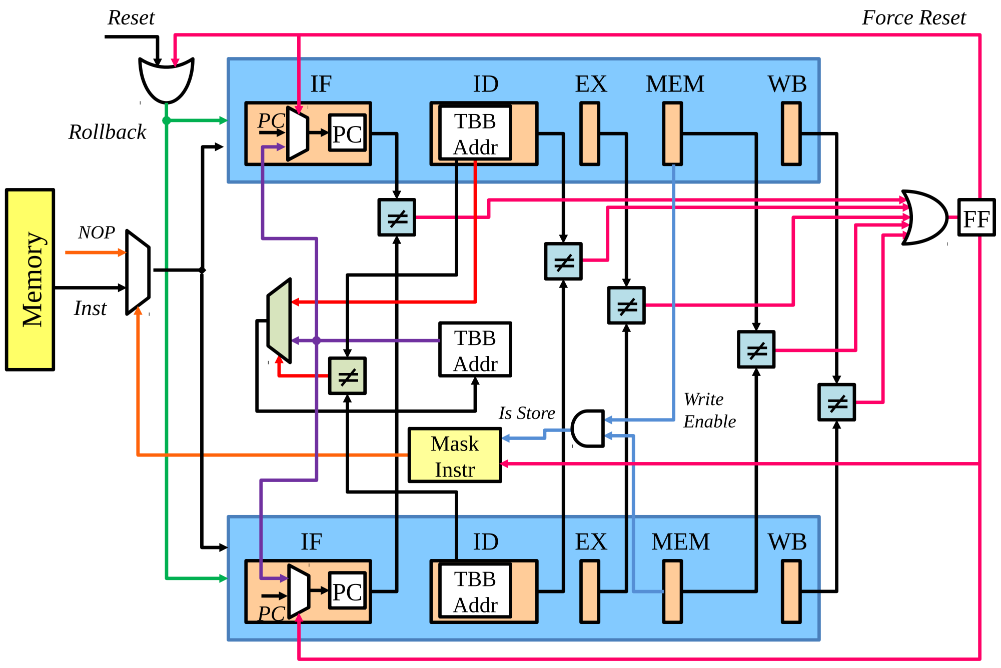
The register file has two operation modes:

- **'read/write' mode**, in which the executing instructions can write data to registers; and
- **'read-only' mode**, in which instructions can read data from registers but cannot modify them.

Only the master core may modify the register file when the 'read/write' mode is set. ToBBA assumes the data and instruction memories are protected with error-correcting code (ECC), and the register file is protected with some power efficient hardening strategy, such as a Register Value Cache (Blome et al., 2006). The costs associated with a protected register file with ECC are considered in the area and power experiments discussed in Chapter 5.

Error detection is performed when each instruction passes through each pipeline stage, when duplicated pipeline states are compared ('$\neq$' in Fig. 3.1). If they match,

Figure 3.1 – Block diagram of the ToBBA architecture



the execution is asserted as correct and the cores continue executing the next instruction in the TBB. Otherwise, an error is detected and the TBB is re-executed from the start. In Section 3.2, we discuss how the rollback and error recovery is implemented in ToBBA. The algorithm and architectural cases that need to be considered to implement the TBB rollback are detailed in Section 3.4.

The hardware required to set the register file in 'read-only' mode is a small logic that uses the 'Write Enable' signal that comes out of the MEM stage of the pipeline of each core and the signal 'Force Reset' which is set in case of errors. If the 'Write Enable' signal is set, the program is currently executing a store instruction, and thus the load and arithmetic instructions re-executed in the rollback cannot modify the register file. In this case, the architecture forces the cores to execute NOPs in case the 'Write Enable' and 'Force Reset' signal is set instead of the original instructions fetched from memory, which avoids the modification of the register file. This logic is implemented in the 'Mask Inst' module in Fig. 3.1.

3.2 Error Detection and Recovery

The comparison that asserts whether an executed instruction is correct is fine-grained at the signal level, and checks **all** architectural signals. If there is a mismatch at any signal, current execution is asserted as incorrect and the TBB is re-executed, otherwise, execution proceeds normally. To rollback correctly to the start of the TBB, the architecture has three registers that store the start address of the block (‘TBB Addr’ in Fig. 3.1). The ‘TBB Addr’ registers are placed in a ‘delayed TMR’ arrangement, in which the central register is updated one cycle after the two copies inside the cores in the instruction decode stage. This update operation is triggered whenever a new TBB commences, which is detected in the hardware by the conclusion of a control-flow instruction (conditional or unconditional branch) of the TBB being committed. In this case, if the two registers ‘TBB Addr’ inside the cores match and the current TBB commits, in the next cycle the ‘TBB Addr’ register of the rollback machinery will be updated.

The ‘delayed TMR’ scheme is necessary because, when a new TBB starts, the ‘TBB Addr’ registers inside the cores are updated with the start address of that new TBB. However, if there is an error exactly when these two registers are updated, i.e., in the last branch instruction of the TBB, the ‘TBB Addr’ register outside the cores still contains the start address of the previous TBB because it is updated one cycle after the other two ‘TBB Addr’ registers. If all the ‘TBB Addr’ registers were updated at the same cycle, in this specific scenario it would not be possible to perform the rollback.

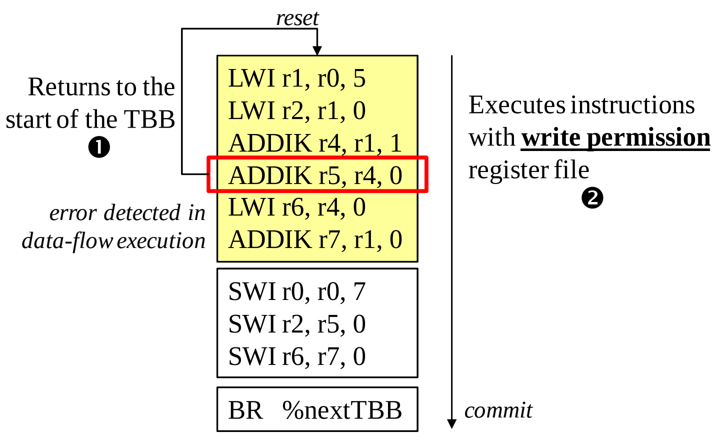
The error detection in the rollback machinery takes place at all pipeline stages. For instance, if an error is detected right at the fetch stage, it does not propagate to deeper stages, which reduces the detection and recovery latency. If any of the comparators flags an error, the ‘Force Reset’ signal is set, the program counter (PC) receives the address contained in the ‘TBB Addr’ and the rollback starts. If no error is flagged the PC receives the appropriate value according to the original control-flow of the program.

The TBB has a well-defined execution life-cycle in the architecture, which is composed of three ordered steps: i) data-flow execution; ii) transaction; and iii) commit.

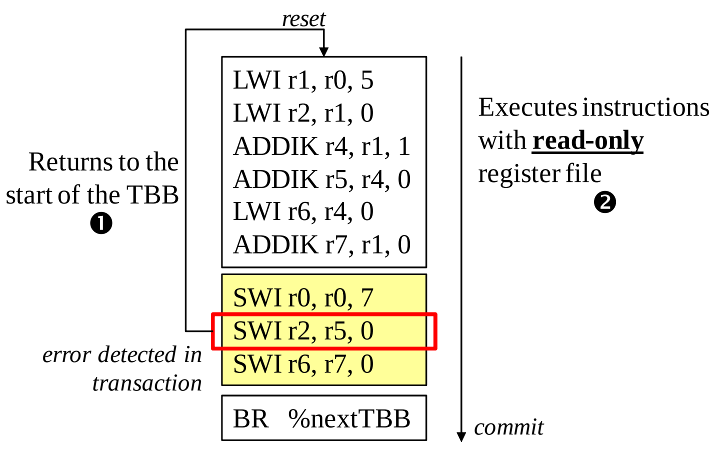
The **data-flow execution** step is composed of memory loads, logic and arithmetic instructions, implementing the TBB register definition presented in Chapter 2. If an error is detected in the data-flow execution step, the rollback unit signals the

Figure 3.2 – Transactional basic-block error scenario

(a) error detected in data-flow execution



(b) error detected in the transaction



error to the two MicroBlaze cores to re-execute the TBB from the start, based on the TBB address stored in the 'TBB Addr' register. That second TBB execution accesses the register file in 'read/write' mode, re-writing data to it. This error scenario is depicted in Fig. 3.2a.

The **transaction** step executes the memory stores of the TBB after the data-flow execution step finishes. When the TBB reaches the transaction, it is guaranteed that the register file is correct and contains the final computed values of the block. Therefore, if an error is detected in the transaction, the rollback unit signals the error and the TBB is re-executed from the start and this second execution of the TBB is done with the register file in 'read-only' mode. Otherwise, if writing data to the register file were allowed, this second TBB execution could fetch values from memory that were modified by the previous partial execution of the transaction step of the TBB and assign them to the registers, leading the TBB to produce wrong results. This error scenario is depicted in Fig. 3.2b. The transaction step ultimately terminates the registers defined in the data-flow execution, implementing the TBB register termination segment.

The **commit** step starts after the transaction steps finishes. At the commit step the destination addresses of the store instructions executed in the transaction are written in memory with their new values, setting the signal 'Write Enable', and the branch to the next TBB is decided and executed. Errors in the commit are handled in the same way as in the transaction step, i.e., the register file mode is set as 'read-only' and the TBB rollbacks.

One important consideration is the execution of successive TBBs in the pipeline. Consider that TBB 'B' is executed after TBB 'A'. Also consider that the last instruction of TBB 'A' is in the execute (EX) stage of the pipeline (i.e., TBB 'A' has not committed yet). The next TBB 'B' can enter the instruction fetch (IF) stage of the pipeline. If an error is detected during the EX stage in TBB 'A', the TBB 'A' can safely be re-executed as no instruction of the TBB 'B' modified the register file yet, because the commit step of the TBB 'A' has set the register file as 'read-only' and this lock was not released. If the execution of 'A' were correct, the lock on the register file would have been already released when the first instruction of 'B' reaches the EX stage, thus the program execution would be correct. All the cases needed to rollback the TBB are detailed in Section 3.4 and in Table 3.1.

The only data that needs to be stored to perform error recovery is the start address of the TBB being executed, i.e., the 'TBB Addr' registers. Thus, because the amount of stored data are very small, the transactional core can perform fine-grained checkpointing. This reduces the error recovery latency to just a few instructions, as it will be shown in Chapter 7.

3.3 Register Liveness and Error Recovery Latency

Full-fledged production compilers do a lot of work to maximize register usage among basic blocks, which reduces the number of spill instructions and increases the re-use of produced data between the basic blocks. Spill instructions are additional load and store instructions created to store and re-load live-out values from the main memory. These instructions are inserted when there are no available registers to be allocated during Register Allocation, forcing these shared values to be stored in memory. On the other hand, maximizing the register re-use, i.e., *liveness*, is not a good measure for reliability, because the data that the register holds will be exposed to errors during more cycles (Yan and Zhang, 2005). The TBB **eliminates** register liveness, and, as a result, it increases reliability.

The length of the basic block has a very important implication in the rollback machinery: it makes the worst case latency of the error recovery mechanism deterministic and known at compilation. In the worst case scenario, the last instruction of the TBB is the erroneous one, which will force the re-execution of all instructions. If the error is detected in the first TBB instruction, the recovery latency is only two cycles (one cycle to detect the error, and one cycle to prepare the rollback internal signals).

3.4 Rollback Machinery Algorithm

The rollback mechanism is based on the straight comparison between the two RISC cores inside ToBBA. This comparison is implemented by matching the inputs of all the registers of the master core against the inputs of the slave one, which takes into account not only the pipeline registers, but also the internal state registers. This approach improves the number of faults detected and reduces the fault detection latency vis-à-vis a solution that compares only the output of the cores. In ToBBA's fault detection mechanism, all faults that propagate to register inputs are detected.

In case of fault detection, memory writing is the only action that is immediately blocked, i.e., the 'Write Enable' signal is blocked in the same cycle the error was detected and it continues blocked until rollback finishes. All other actions, such as writing to registers, are allowed, even if they may propagate errors to the flip-flops and corrupt the register file. From the performance overhead point of view, it means this solution does not increase the paths inside the core hardware, thus it does not slow down the core frequency. The only constrained path is the memory write.

Apart from the comparison, the calculation of the rollback address is the critical point to this solution. The calculation takes place in the Decode stage and it is triplicated using the 'delayed TMR' scheme introduced previously because it cannot be recovered with DMR only. Every time a branch instruction reaches the decode stage, the rollback address is updated with the start address of the next TBB.

Choosing the correct address to recover the program execution is essential. Two possibilities can be chosen: to either go back to the current TBB or to branch to the next TBB. This decision depends on which instructions executed when a fault is detected. These cases are discussed next.

- **Fault detection during register definition:** in this case the pipeline is filled with instructions like loads, arithmetic operations and logic operations, and no write to the memory is performed in the MEM write stage. When a fault is detected, the cores are reset, and the fetch address is set to the current TBB. Since the TBBs start without any information stored in the register file, the rollback will fill all registers it needs to execute that TBB properly;

- **Fault detection during register termination:** in this case, at least one memory write instruction arrived at the MEM write stage. The cores are reset to the current TBB address, exactly as the previous case. However, the instructions being fetched during the rollback are now converted to NOPs until a MEM write operation is fetched. It means the whole TBB is not going to be executed, only the memory-modifying instructions. Since the TBB schedules the MEM write instructions to the end of the block, if a fault occurs in any of these instructions, it is safe to assume the register file is correct. Thus, the previous instructions that were not stores do not need to be executed again, which reduces the error recovery latency;

- **Fault detected during the transition between the current and the next TBBs:** the rollback address will be set to the current TBB if a fault is detected when a store instruction is in the MEM write stage and a branch instruction is being executed, meaning the memory might be corrupted. Otherwise, if the block transition was already executed (it is in the MEM write stage), the rollback mechanism can safely branch to the beginning of the next TBB.

All these corner cases of the rollback machinery when performing the TBB rollback given by the in-flight instructions in the pipeline are detailed in Table 3.1. In this table, 'Load|ALU' corresponds to the instructions in the register definition segment of the TBB. Notice that during rollback the 'Write Enable' signal is blocked, thus in Table 3.1 any single instruction does not modify the memory in any way.

3.5 ToBBA Fault Model

This section discusses the fault model considered to perform the fault injection experiments of Chapter 7. It also discusses how ToBBA can cope with hard faults and timing errors besides Single Event Upsets (SEU's), showing the applicability of ToBBA not only to harden against most widespread soft error fault model.

- **Single Event Transients (SET) and Upsets (SEU):** soft errors are non-destructive corruption in data caused by highly energized particles glitching a logical circuit current or changing the state of memory elements. If the fault occurs directly in memory elements we have a SEU. If the logical glitch is captured by memory elements, a SET is said to have occurred. Notice that SET's can be observed as SEUs only if the circuit's internal state, i.e., the memory elements, is observed for errors. In this work, the fault model ToBBA assumes is the soft error caused either by SET or SEU, but both being observed as an SEU. The SET and SEU assume that only one bit of the observable output deviates from the non-faulty computation;

- **Multiple Bits Upsets (MBU):** the rollback mechanism implemented in ToBBA is not limited to the detection and correction of single errors, as it works with multiple errors as well. The comparators depicted in Fig. 3.1 check if all bits of the entire signal match, and, thus, it does not matter by how many bits the mismatch is. This is an important difference with current error detection techniques based on signature checking in case of control-flow errors, such as (Chaudhari, Abraham and Park, 2013). In signature checking, depending on how many bits are flipped, the program can branch to a memory area that is not covered by the additional checking mechanism, which will ultimately lead to some sort of segmentation fault at the software level. In this case, the only feasible alternative to recover execution is to re-launch the entire application, incurring on heavy penalties to rollback the

Table 3.1 – Corner cases of the rollback machinery depending on the in-flight instructions in the pipeline

Pipeline Stage					**TBB Rollback Address**	**Register-file is 'read-only'**
Fetch	**Decode**	**Execute**	**MEM**	**Write-Back**		
Load|ALU	NOP	NOP	NOP	NOP	Current TBB	No
Load|ALU	Load|ALU	NOP	NOP	NOP	Current TBB	No
Load|ALU	Load|ALU	Load|ALU	Load|ALU	Load|ALU	Current TBB	No
Store	Load|ALU	Load|ALU	Load|ALU	Load|ALU	Current TBB	No
Store	Store	Load|ALU	Load|ALU	Load|ALU	Current TBB	No
Store	Store	Store	Load|ALU	Load|ALU	Current TBB	No
Store	Store	Store	Store	Load|ALU	Current TBB	No
Store	Store	Store	Store	Store	Current TBB	Yes
Branch	Store	Store	Store	Store	Current TBB	Yes
Bubble	Branch	Store	Store	Store	Current TBB	Yes
Bubble	Bubble	Branch	Store	Store	Current TBB	Yes
1st Inst of Next TBB	Bubble	Bubble	Branch	Store	Next TBB	No
2nd Inst of Next TBB	1st Inst of Next TBB	Bubble	Bubble	Branch	Next TBB	No
Branch	Store	Load|ALU	NOP	NOP	Current TBB	No
Bubble	Bubble	Branch	Store	Load|ALU	Current TBB	Yes
1st Inst of Next TBB	Bubble	Bubble	Branch	Store	Next TBB	Yes

system to a correct state. More on this will be discussed in Chapter 6;

- **Timing faults** can occur due to the variability in the manufacturing of transistors and due to natural aging and also manifest as transient errors, thus, for future technology nodes it is important to consider them in the fault model, as Chapter 1 discussed. In ToBBA, timing faults will be indistinguishably captured as an instance of a soft error. If any of the cores gets desynchronized at any time due to a timing fault, the error detection mechanism will be triggered and the execution will be rolled back to the beginning of the TBB;

- **Permanent Errors:** with a standard DMR arrangement, a fault tolerant system is not able to identify the faulty core, hence it is not possible to isolate it from the system (BERNICK ET AL., 2005). However, with ToBBA it is possible to know the system has a permanent error. In case of permanent errors, the proposed architecture is able to detect them with small modifications in the rollback machinery. The rollback can count how many times the same TBB has re-executed, and if this count reaches a determined threshold, a permanent error is assumed and flagged. The value of the threshold should be carefully defined according to the error rate to avoid long duration transients be classified as permanent errors. A standard DMR arrangement is also capable of detecting permanent errors.

3.6 Current Limitations and How To Overcome Them

The use of TBBs and its execution in ToBBA have some limitations on its current form:

Pre-compiled binaries. This work does not currently work with pre-compiled binaries. It could work if the TBB generation algorithm were implemented as a mechanism of binary translation. The challenge is to avoid breaking the correct register allocation when moving the instructions and to recompute functions' prologue and epilogue due to possibly additional instructions and memory slots.

Errors in cache memories. Although a full memory hierarchy is indeed important in microprocessor reliability due to its impact on performance (SANTINI ET AL., 2014), its presence or absence does not influence the fault coverage of the proposed technique. In case there were errors in the cache tag, the microprocessor would naturally treat it as a cache miss, forcing this cache line to be evicted and replaced. The problem lies in the cache data, which require some sort of parity check. Notice that error correction is not necessary, because in case the parity check flags an error, the erroneous cache line can just be evicted to correct the error.

Error detection overhead in the DMR core. The error detection model used in ToBBA compares all internal registers of the DMR core. Comparing that many bits might increase the circuit's critical path, reducing the synthesis frequency. The error detection model could be changed to a mechanism similar to DIVA (Austin, 1999), which only compares the instructions at the pipelines commit stage. The modification to DIVA would be, instead of adding the Checking stage in the pipeline, the duplicated out-of-order (OOO) pipelines would be compared only in the commit stage. If a mismatch is found, execution can rollback to the start of the TBB. This change would also allow for error detection and recovery in OOO architectures.

Permanent error mitigation. Although ToBBA's fault model can detect permanent errors, their mitigation can be challenging. It would be possible to selectively discard pipeline stages of each RISC core, and turning off the error detection for these stages. In this way, ToBBA would execute in a degraded mode for fault tolerance, which might be desirable in some situations at system designer's discretion. Permanent error mitigation will also be discussed in Chapter 7.

3.7 Crosscutting the SW Layer

The way that the ToBBA architecture works is totally dependent on how the TBB is defined and constructed. Thus, ToBBA and the TBB definition of Chapter 2 crosscuts in several ways.

- **Instruction Ordering at Compilation**: the rollback mechanism assumes that ToBBA will correctly set the register file mode according to the types of instruction being executed when an error occurs. If the TBB were not ordered, setting the register file mode either 'read/write' or 'read-only' would not work;
- **Re-execution & TBB Length When Rollback**: the bigger the TBB is,

the more expensive it is to recover the error because of the worst case error recovery latency discussed in Section 3.3. On the other hand, the smaller the TBB is, the more expensive it is to execute it due to excessive live-out spilling in the final program;

- **Instruction Ordering at Runtime**: the instructions in ToBBA as it is currently implemented have to execute in-order due to how error detection works. In addition, the error recovery assumes that when register definition segment ends and when register termination starts the register file mode can change from read/write to read-only. Supporting Out-Of-Order (OOO) execution is one of the future works planned for this dissertation, and more on that is discussed in Chapter 7;

- **Register Pressure and Forced TBB Splitting**: the TBB potentially increases register pressure because a TBB has to fill register more often, and it might happen that the TBB has to be split in case there are not enough registers available to fill all live-outs used. In this case, the TBB has to be split, increasing the number of live-out spillings in the program.

4 SPILL REGISTER FILE

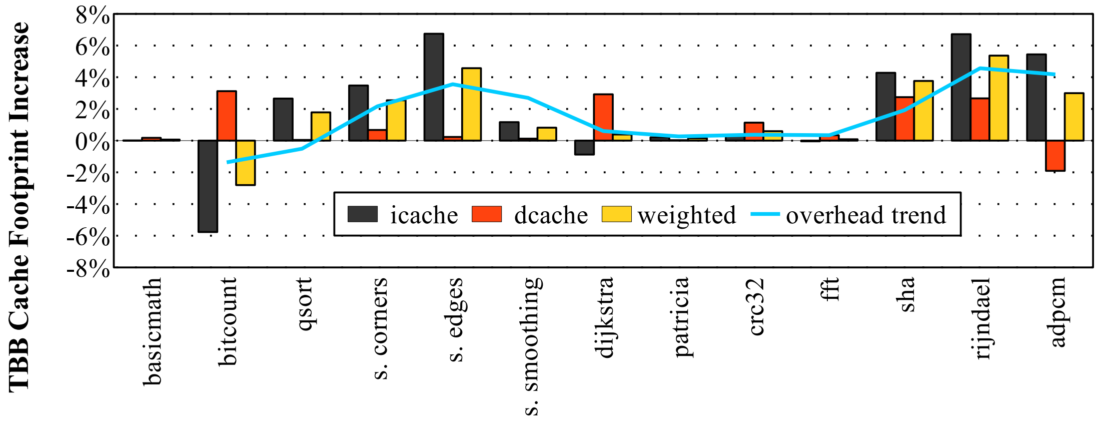
4.1 Sources of Overhead in the TBB

As it will be presented in Chapter 5, the measured geometric mean overhead of the original TBB and ToBBA is 1.54. This result is interesting, because additional spill code composed of load and store instructions could potentially jeopardize performance. As it will be discussed in Chapter 6, this reported overhead is already a considerably advance over state-of-the-art error correction methods. However, it is possible to reduce this overhead if we use hardware support. Before introducing this auxiliary hardware, the Spill Register File (SRF), it is important to understand the sources of the performance overhead in the original TBB. The performance overhead of the TBB is derived mainly from two sources: (1) the register allocator does not have a big room to sustain the optimum register usage among hot kernels, thus even slight changes on the register allocation incur huge performance penalties; (2) additional spill instructions incur more committed instructions.

4.1.1 Sensitiveness to Register Allocation

A compiler puts a lot of effort to maximize the time a register is live inside a basic block, increasing the basic block length. A bigger basic block might uncover hidden instruction level parallelism, boosting performance in superscalar architectures. A big basic block impacts the register allocator by increasing the lifetime of registers. This leads to higher register pressure and usage dependencies between allocated registers, i.e., there are fewer registers available to be used used (Goodman and Hsu, 1988). It is known that instruction scheduling influences the register allocator performance depending on the size and type of instructions executed in

Figure 4.1 – Cache footprint increase for both the data cache (dcache) and the instruction cache (icache). Negative values are footprint reduction. Overhead trend line corresponds to the TBB cache footprint increase weighted by icache and dcache sizes (32KB and 64KB, respectively).



the hot kernel of a program. In GCC, for the SPECfp95's '145.fpppp' program, instruction scheduling before register allocation causes a performance overhead of 1.63 (Makarov, 2004). This is due to the minimal cache footprint of '145.fpppp'. The cache footprint is the set of cache lines used by a program (Thiebaut and Stone, 1987). A small cache footprint makes computation and data movements between registers the prominent bottleneck in performance, and the program becomes more sensitive to register allocation. Figure 4.1 shows the cache footprint increase of TBB programs over the same standard programs. The TBB programs presenting cache footprint increase are exactly the same that exhibit performance overhead, because the TBB generation changes the instruction schedule before and after register allocation. Hence, the TBB instruction scheduling affects the performance of the original register allocation.

4.1.2 Additional Committed Instructions

The spill instructions, although being load and store ones to constant memory locations, what favors a reduction in cache miss, still contribute to a higher number of committed instructions. Therefore, their latency also contributes to increase the TBB performance overhead. This is the reason why programs that exhibit almost no cache footprint increase (or even reduction, such as 'bitcount') might still present performance overhead. The changes in cache footprint were not enough to cancel the increased latency of the additional instructions.

4.2 The Spill Register File Hardware

The goal of the Spill Register File (SRF) is to significantly reduce the performance overhead introduced by the TBB when ToBBA is error-free. With the use of the SRF, a single instruction can be used to spill registers containing live-out values instead of adding additional load and store instructions, as discussed in Chapter 2. These registers containing live-out values are copied to the SRF before the store instructions, thus respecting the instruction schedule imposed by the TBB. The error correction policy of the TBB introduced in Chapter 2 does not change with the introduction of the SRF.

4.2.1 Modifications Required in the TBB and in ToBBA

In the SRF, the TBB is modified in two ways: (1) the additional spill instructions are not created; (2) all live-out values of the TBB must be copied to the SRF before the register termination region of the TBB starts. Although due to (1) the additional load and store instructions are not created, the TBB instruction scheduling is not modified. To implement (2), the SRF adds a new instruction, the **COPY_SRF**. This instruction works as a *fence* (or barrier) to live-out values, and is scheduled between the TBB's register definition and termination regions. Placing this instruction between these two regions is necessary to keep the error correction policy unchanged. A side-effect of the COPY_SRF instruction is that it has the same semantics and scheduling of a standard memory barrier instruction, what allow out-of-order execution. More on this is discussed in Chapter 7.

When a TBB executes the COPY_SRF instruction, the original ToBBA register file, which we will call Working Register File (WRF), is copied to the SRF, i.e., all registers are assumed to be live-out registers. When a TBB using a live-out register copied previously to the SRF needs to correct an error, the error correction interrupt recovers the live-in registers of the TBB back to the working register file using the SRF. Live-in registers are the registers used by a TBB that is defined in another TBB. Figure 4.2 shows a TBB and its live-in registers. In this example, the stack pointer (SP), R3, and R4 need to be recovered from the SRF to the Working Register

Figure 4.2 – TBB sample from ‘dijkstra’, showing the live-in registers of the block, and the COPY_SRF instruction. This sample uses ARM instructions.

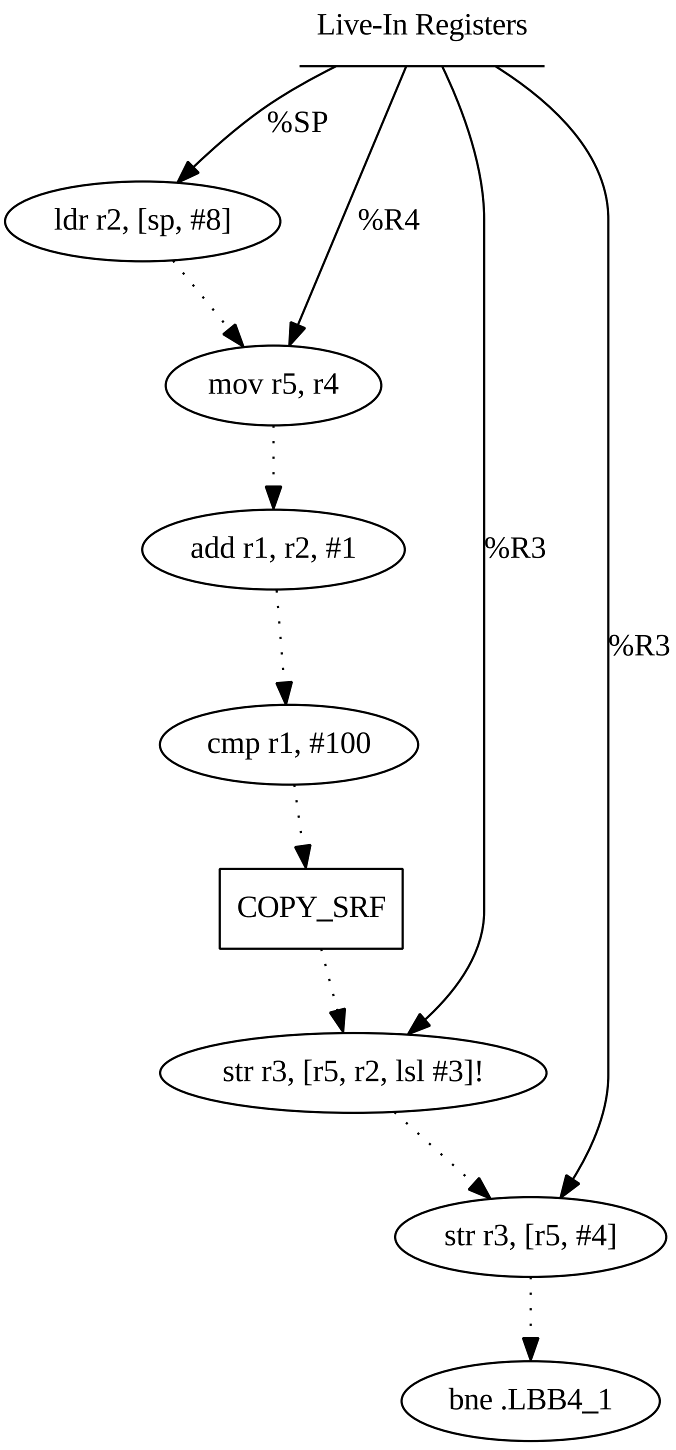
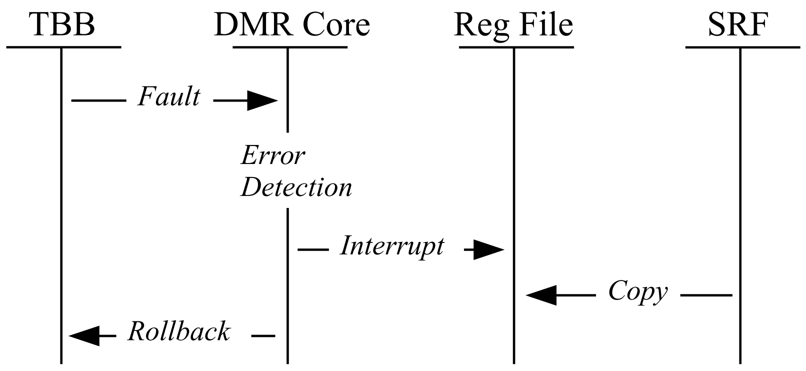


Figure 4.3 – Flow from fault occurrence to error correction depicting ToBBA with its Working Register File (Reg File) and the SRF.



File in case this TBB has to rollback.

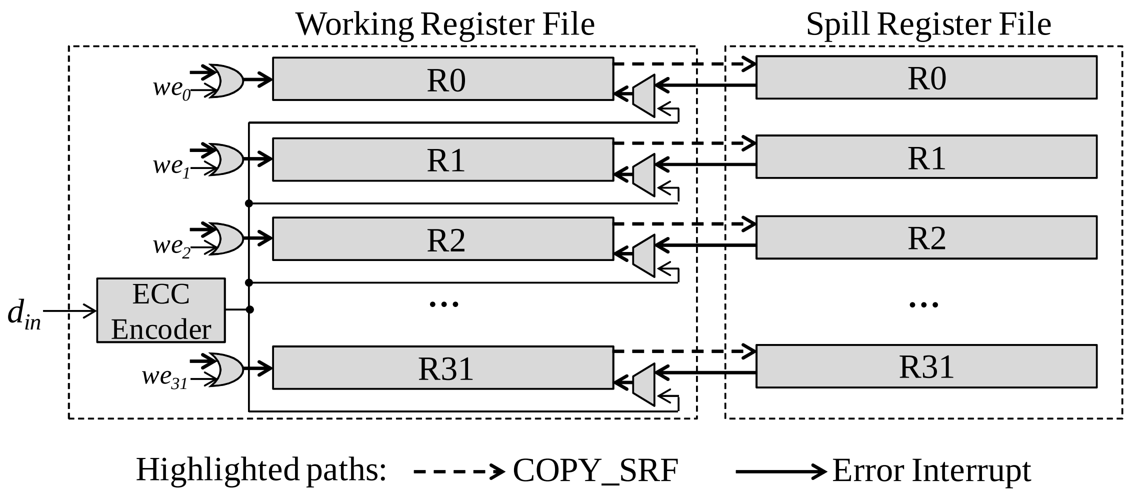
In the hardware level, three modifications in the error detection and correction model of the DMR arrangement inside ToBBA are required: (1) the addition of the Spill Register File; (2) a new interrupt thrown when an error is detected; and (3) the support for the COPY_SRF fence instruction. The assumption that the memory and the Working Register File must be hardened is still valid. The associated costs with (1) will be discussed in Chapter 7. Both (2) and (3) are straightforward modifications with no impact in performance and power.

Notice that the SRF does not need to be protected with ECC. If there's an error in the SRF, program execution is known to be correct and the SRF data are not going to be used. If there is an error that corrupt program execution, the SRF data are correct and can be used to rollback the TBB. As aforementioned, the Working Register File needs to be protected with ECC, because a latched error on it cannot be allowed to propagate to the SRF.

4.2.2 Interrupt-Triggered Soft Error Correction Hardware

Before introducing the SRF hardware in details, Figure 4.3 shows the actions executed from when a fault occurs in the software until the architecture rollbacks the TBB to correct the error. After the fault occurs, the first step is to perform error detection based on the comparison of internal signals between the duplicated cores. The contribution of this dissertation starts from this point, where after the error is detected, the DMR core throws an interrupt. The interrupt handler copies the SRF back to the Working Register File (Reg File in Figure 4.3), recovering all

Figure 4.4 – SRF hardware and its data path. d_{in} represents the data available in the write port, and $we_0 \ldots we_{31}$ are the write enable signals.



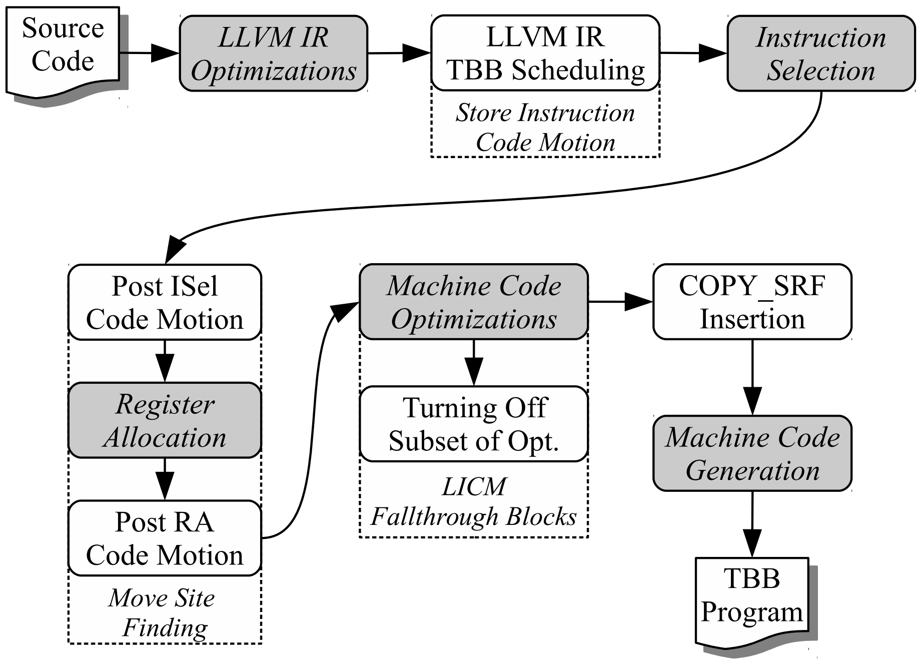
the live-in registers of the TBB (recall Figure 4.2 depicting live-in registers). After the working register file is recovered, rollback sets the execution back to the first instruction of the TBB to correct the error.

The SRF hardware is shown in Figure 4.4. The write port d_{in} (encoded with ECC) and write enable signals $we_0 \ldots we_{31}$, are used during regular error-free execution. The dashed arrows in Figure 4.4 show the operation performed by the COPY_SRF instruction, which copies the contents of each Working Register File (WRF) register to its SRF counterpart. The solid bold lines indicate the operations required when an error is detected and rollback is required. The multiplexers at the input of WRF registers select the SRF as input, and the write enable of every WRF register is forced to one, triggering the parallel write of all WRF registers. Thereby, a very short rollback latency is attainable. Moreover, compared to a regular ECC-capable register file, the only addition to its write path is a 2×1 multiplexer, hence the SRF does not increase significantly the critical path of the WRF. Note that the reading circuitry required during normal operation is omitted in Figure 4.4 for the sake of clarity.

4.3 Modifications in the TBB Generation

Figure 4.5 depicts the modified flow inside LLVM to generate the TBB for the SRF.

Figure 4.5 – Algorithm for generating TBB programs in LLVM for the Spill Register File. Gray shapes represent standard LLVM's steps.



The first modification when generating TBBs for SRF with respect to the algorithm given in Chapter 2 is to skip Algorithm 2.1, i.e., there is no need to create a global variable for each function anymore. This is because with the SRF there is no need to spill the live-out registers with store instructions, the function's return value included. As such, the second modification is to skip Algorithm 2.2, i.e., it is not necessary to schedule function calls on their own TBB. Recall that this scheduling was necessary to guarantee that the function call was the last instruction before the branch, allowing for error recovery without re-executing the function in case the error was in the 'call' instruction itself. Finally, we also skip Algorithm 2.3, because these live-out values will be stored in the SRF instead of being spilled manually.

The only FunctionPass required in the LLVM IR level right before Instruction Selection is to move all store instructions of the original basic block right before the terminator instruction. This means that any original store in the basic block will be scheduled like the TBB, i.e., the register definition and register termination regions are formed in the SRF. In addition, the original load and store instructions need to be marked as 'volatile' as before. Notice that although the SRF does not add additional load and store instructions to fill and spill live-out values, because the instruction scheduling has changed, there will be still impact during register allocation.

The remaining passes after Instruction Selection remain the same, i.e., the ‘Post ISel Code Motion’ and the ‘Post RA Code Motion’ are still required, as well as turning off the Loop Invariant Code Motion (LICM) and Unconditional Branch removal (fallthrough blocks) Machine Code Optimization passes. In the back-end, the only modification is the insertion of the COPY_SRF instruction.

Before function’s prologue and epilogue, and the code generation, we add the COPY_SRF instruction right before the first store instruction for all basic blocks in the program. Placing the COPY_SRF instruction does not change register allocation, because it does not use any register. At the end of this step, the final machine code with TBBs to be used with the SRF is generated.

5 EXPERIMENTAL EVALUATION

5.1 Methodology

Error coverage. The fault injection campaign was done using the VHDL code of the hardware architecture. The architecture was deployed in a Xilinx Virtex-5 FPGA, where at each program cycle one fault was injected in one signal of the entire netlist by a *saboteur* (JENN ET AL., 1994) module. The fault injector uses the netlist after FPGA post-synthesis. The netlist is modified so that each net is instrumented and its value can be corrupted by the saboteur module at any time synchronously in the clock cycle. After the software stops executing, the controller compares the memory of the current execution with the golden execution. If there is a mismatch, then the fault became an error, otherwise it was corrected or architecturally masked. The fault model assumed is the soft error, with only one bit chosen at random being changed when the fault is injected and the bit-flip is forced to last one clock cycle. This fault model correctly accounts for Single Event Upsets (SEU) (PETERSEN, 2011, CH. 2, P. 14). Faults were also injected in the rollback machinery and its internal components, thus the case where the fault tolerance mechanism is corrupted is considered. We used as benchmark six small applications: bubble sort (bbsort), least squares (lsquares), CRC32, minimum spanning tree computation (kruskal), all pairs shortest paths in a graph (floyd), and matrix multiplication (matmul). The fault injection campaign for the entire benchmark is comprised of **672,348,891** injected faults. The algorithms and their data input used in the error coverage experiments are presented in Table 5.1. The fault injection results in terms of error detection and correction coverage are discussed in Section 5.2.

Stuck-at fault injection. ToBBA's VHDL was synthesized at gate level with

Table 5.1 – Benchmark used to evaluate the Stack for error coverage.

Acronym	Application Description and Data-Set
bbsort	Bubble Sort of 10 elements
lsquares	Least Squares of 24 pairs of points
crc32	CRC in 32 bits of the sentence 'The quick brown fox jumps over the lazy dog
kruskal	Kruskal minimum spanning tree of 7 nodes (greedy algorithm)
floyd	Floyd-Warshall all-pairs shortest paths of 7 nodes (dynamic programming algorithm)
matmul	10×10 matrix multiplication

Synopsys Design Compiler, and that gate level version was exported to Verilog. The synthesized ToBBA Verilog was fed into Synopsys TetraMAX, and fault injected based on a waveform generated with ModelSim. The waveform was created with a full execution of the bubble sort algorithm ordering 10 integer elements of a vector. In total, **121,708** stuck-at faults were injected with TetraMAX. Each fault was injected from the start of the simulation, and the stuck-at lasted until the program has finished its execution. The stuck-at fault injection encompassed all input and output ports of all gates in the circuit, for both stuck-at-zero and stuck-at-one.

Area occupation and power dissipation. The power and area results obtained for ToBBA were obtained with the Cadence RTL Compiler using 65 nm transistor technology from the ToBBA description in VHDL. The power and area results for the register file were computed using CACTI 6.5 (Muralimanohar, Balasubramonian and Jouppi, 2009). In the area and power results, we synthesized ToBBA in 300 MHz. As the baseline architecture for calculating the area and peak power overhead, we adopted a single MicroBlaze with its original register file of 32 registers synthesized in the same frequency as ToBBA. These results are discussed in Section 5.3.

Performance overhead. The performance results discussed in Section 5.4 were obtained in two ways. In the fault injection campaign, the 'saboteur' module that injected the faults also captured the instruction in which the fault was injected. With this information, we have computed the error recovery latency from ToBBA's RTL model presented in Section 5.4.1. Performance overhead was measured with

Table 5.2 – gem5 simulation configuration.

Parameters	**Values**
Processor	In-order, single core, 1GHz, 2-wide decode width, ARM v7-a ISA
L1 cache	2-way, 64KB Data/32KB Inst, 64B line size, LRU
L2 cache	8-way, 2MB, 64B line size, LRU
Main memory	DDR3-1600, 512 MB
Functional units	2 Integer ALU, 1 Load/Store Unit, 1 Float ALU
Branch prediction	Tournament, 4K-entry BTB

gem5 (Binkert et al., 2011) for an in-order architecture using the ARM v7-a instruction set. We have used the following MiBench (Guthaus et al., 2001) subset as benchmark: basicmath, bitcount, qsort, susan (corners, edges, smoothing), dijkstra, patricia, crc32, fft, sha, rijndael, and adpcm. These programs were selected because they do not depend on pre-compiled binaries, i.e., their full C source code is available. The standard and TBB programs were executed for both the small and large data sets to make it possible to evaluate how the TBB program scales. Table 5.2 lists the gem5 simulation configuration parameters used in the performance experiments. LLVM 3.4 was extended to include the algorithm for generating TBB programs presented in Chapter 2 and in Chapter 4 for the Spill Register File. The benchmarks were compiled with all LLVM code optimizations (O3 flag) for the baseline versions. The TBB programs were also compiled using the O3 flag with the loop-invariant code motion (LICM) and unconditional branch removal disabled.

Energy consumption. McPAT (Li et al., 2009) was fed with the execution statistics obtained with gem5 for the baseline and the TBB program versions to extract energy consumption. The simulated architecture described in Table 5.2 was modeled in McPAT, which invokes CACTI (Muralimanohar, Balasubramonian and Jouppi, 2009) to model the memories' energy consumption.

Table 5.3 – ToBBA's fault injection results

Application	**Correction**[a]	**Total Coverage**[b]	**No. of Faults**	**Masked %**
bbsort	99.85475%	99.92214%	10,808,861	44.20838%
lsquares	99.04251%	99.74389%	882,007	44.97413%
crc32	99.57377%	99.97360%	20,169,300	13.88979%
kruskal	98.82113%	99.89848%	254,465,240	44.97101%
floyd	99.67682%	99.88852%	109,994,127	49.08854%
matmul	99.12455%	99.86334%	276,029,356	43.20838%
geomean[c]	99.34822%	99.88164%	–	37.16270%

[a]Error correction coverage only.
[b]Total coverage includes error correction coverage and error detection coverage (errors detected but not corrected).
[c]Geometric mean of the entire benchmark.

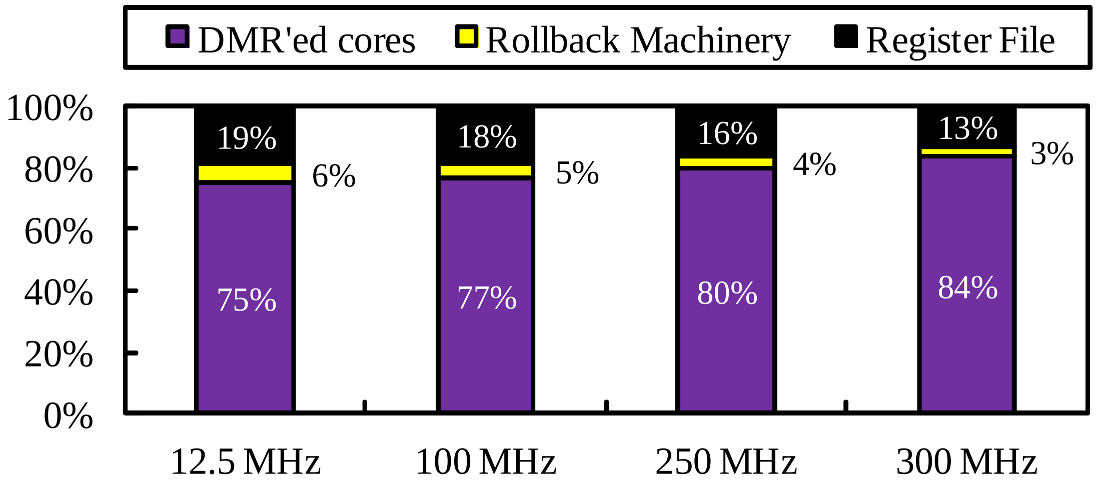
5.2 Error Coverage

The results of the ToBBA's fault injection campaign in terms of SEU error correction and error detection coverage are presented in Table 5.3. The fault injection campaign for the entire benchmark is comprised of **672,348,891** injected faults. For the entire benchmark, the error detection coverage of ToBBA, i.e., the correction and detection error coverage summed up, is 99.9%. Considering only the corrected errors, the ToBBA's average error correction coverage is 99.3%.

Because the fault injection campaign considered faults in *all* architectural components, there was a small number of errors that were not detected, i.e., Silent Data Corruption (SDC) were observed. These SDC's are due to faults injected in the address of the data that will be written in memory just after the comparator inside ToBBA and right before memory is actually written. To reduce the probability of an SDC case, we could add more comparators until the probability of not detecting an error is acceptable for the system designer, paying the costs of area, power, and latency. Another solution would be ToBBA to encode all data with ECC so the memory controller could later check if the ECC is correct before writing the data in memory. SDC errors account for **less than 0.1%** of all the fault injection campaign.

The results presented in this section are interesting because until now in the published literature there is not a single methodology that saves power and area wrt. TMR with the same high error correction coverage as TMR. TMR is really

Figure 5.1 – ToBBA's relative area occupation of each architectural unit



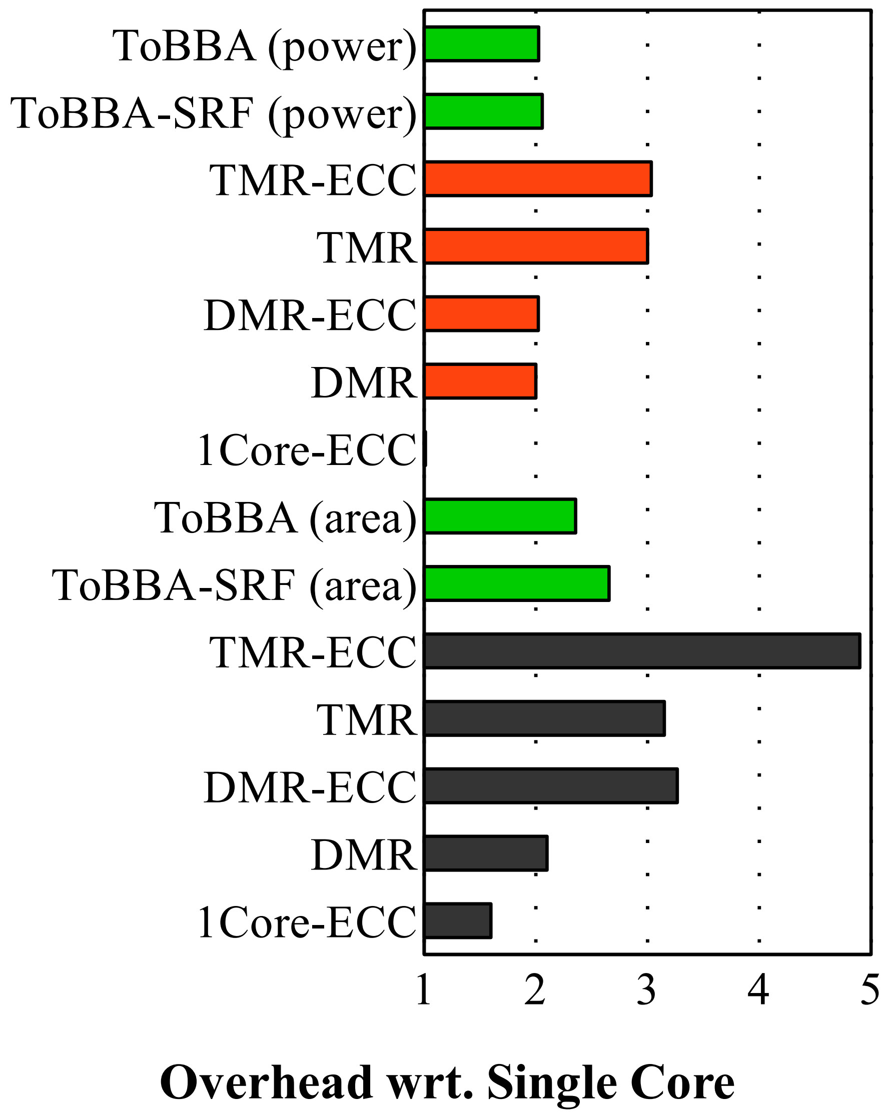
difficult to beat in performance because its performance overhead is negligible and it scales linearly with the circuit's complexity (HENTSCHKE ET AL., 2002). Usually, to overcome the high area occupation and power consumption TMR imposes, fault tolerance solutions either relax their requirements in performance or error coverage, as discussed in Chapter 6.

5.3 Power & Area Characterization

Fig. 5.1 shows the relative area occupation of each ToBBA's architectural unit, which shows the rollback machinery accounts for negligible part of the total area. The observed variation in area when the frequency of operation increases is due to the extra effort the synthesis tool does to achieve the higher frequencies, although all versions being functionally equivalent.

The area occupation and peak power dissipation overhead results are presented in Figure 5.2. In this figure, the baseline is a single-core with a register file without ECC. In the area and power evaluation, five scenarios were created to allow for a better comparison with ToBBA: (1) single-core with ECC (1Core-ECC); (2) DMR without ECC (DMR); (3) DMR with ECC (DMR-ECC); (4) TMR without ECC (TMR); and (5) TMR with ECC (TMR-ECC). In this chart, ToBBA's results are presented in the green bars next to 'TMR-ECC' for both the versions with (ToBBA-SRF) and without (ToBBA) the SRF. The reader should refer to Chapter 1 for the implications of using or not ECC in DMR and TMR systems.

Figure 5.2 – Area occupation (gray bars) and peak power dissipation (orange bars) overhead wrt. unhardened single core with one register file (baseline). The chart shows data using unhardened and ECC protected register files for five configurations: single-core with ECC (1Core-ECC), DMR and TMR without ECC (DMR, TMR), DMR and TMR with ECC (DMR-ECC, TMR-ECC), and ToBBA with and without the SRF (green bars).



The area and power results are based on two observations: (1) a register file protected with ECC roughly occupies three times as much area as a standard, unhardened register file (BLOME ET AL., 2006); (2) the SRF does not need to be protected with ECC. Hypothesis (1) is a known fact. The rationale behind (2) is that, if there's an error in the SRF, program execution is known to be correct and the SRF data are not going to be used. If there is an error that corrupt program execution, the SRF data are correct and can be used to rollback the TBB. In ToBBA, the Working Register File needs to be protected with ECC, because a latched error on it cannot be allowed to propagate to the SRF.

In area, 'ToBBA-SRF' has an overhead of 2.65 wrt. a single core, and 'ToBBA' has an overhead of 2.35, still being better than 'DMR-ECC' and standard 'TMR'. Notice that 'DMR-ECC' is not capable of error correction. The use of the SRF occupies 18% less area than 'TMR', the first configuration that is capable of error correction. 'ToBBA-SRF' increases 12% the area overhead compared with 'ToBBA'.

In power, 'ToBBA-SRF' has an overhead of 2.05 wrt. a single core, and 'ToBBA' has an overhead of 2.02, which is in pair with 'DMR' and 'DMR-ECC'. Notice that for power dissipation, ToBBA provides error correction as 'TMR' does but with the same costs of only error detection as 'DMR' and 'DMR-ECC' do. 'ToBBA-SRF' increases 2% the power overhead compared with 'ToBBA'.

5.4 Performance & Error Recovery Latency Analysis

5.4.1 Error Recovery Latency

Fig. 5.3 shows the worst case and average error recovery latency's with standard deviation for the rollback machinery to recover program execution to the start of the TBB. As discussed before, the worst case latency is always the length of the TBB. The measured average recovery latency of the rollback is 6.17 cycles with 2.09 of standard deviation. These results show that, on average, the rollback latency is less than $^1/_3$ of the total number of instructions of a TBB.

The reduced error recovery latency and its deterministic worst case computation is a key contribution of the Transactional Stack. Error recovery is known to be very expensive in fault tolerant systems, which incurs in performance overhead from 25% to almost 100% of the entire application (CHEN AND YANG, 2013), as Chapter 1

Figure 5.3 – Worst case and average error recovery latency with standard deviation in number of executed cycles to rollback after an error is detected

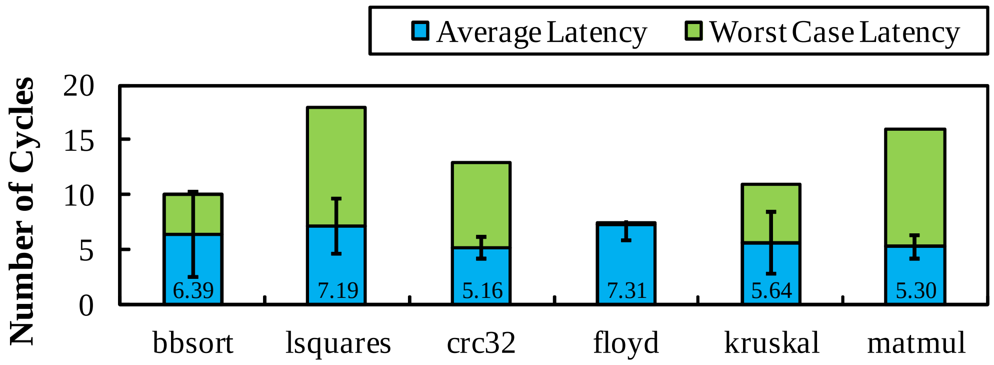
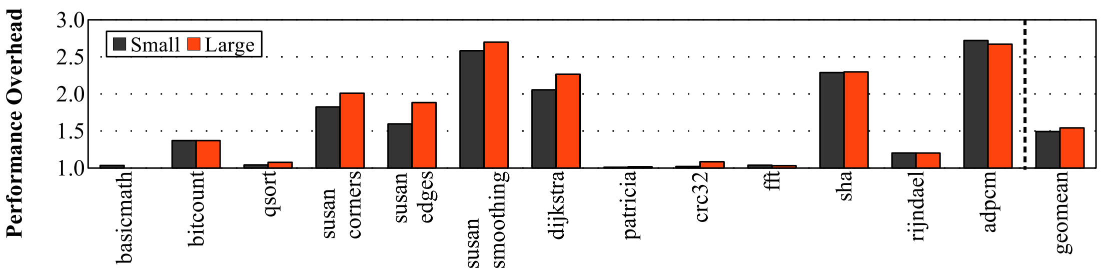


Figure 5.4 – ToBBA's performance overhead without the Spill Register File. These results were obtained for the MiBench subset using the small and large data sets to show how the TBB scales. The measured geometric mean for large data set ('geomean' in the chart) is 1.54.



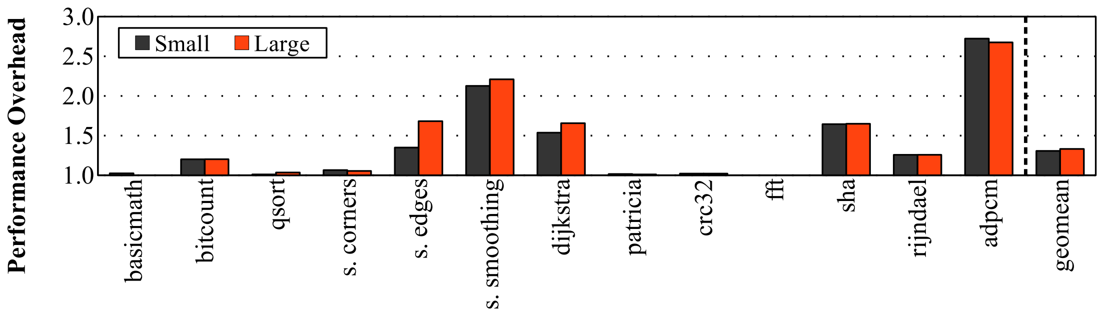
has discussed. This overhead depends on how many instructions are needed to be rolled back upon error detection. In the Stack, this overhead is reduced to the bare minimum, because the rollback has as upper bound the length of the TBB.

5.5 Performance Overhead Without the Spill Register File

The additional spill instructions created to eliminate register-to-register communication between TBBs incur performance overhead. Figure 5.4 shows the measured performance overhead for the MiBench subset when the original program is transformed into an equivalent version that uses TBBs instead of standard basic blocks.

In Figure 5.4, five programs incur performance overhead higher than 2: 'susan corners', 'susan smoothing', 'dijkstra', 'sha', and 'adpcm'. The measured geometric mean overhead was 1.54 ('geomean' in the chart). This result is interesting, because additional spill code composed of load and store instructions could potentially jeop-

Figure 5.5 – Performance overhead of ToBBA-SRF's TBB programs using the Spill Register File instead of additional spill instructions. The measured geometric mean for large data set ('geomean' in the chart) is 1.33.



ardize performance. As it will be discussed in Chapter 6, this reported overhead is already a considerably advance over state-of-the-art error correction methods.

5.6 Performance Overhead With the Spill Register File

Figure 5.5 presents ToBBA-SRF's performance overhead for the selected MiBench subset, and Figure 5.6 shows ToBBA-SRF's performance improvement by using the SRF over creating additional spill instructions. The geometric mean performance overhead dropped from 1.54 to 1.33 due to the SRF and the removal of additional spill instructions. This reduction in performance overhead led to a geometric mean improvement of 13.81%.

Recalling Figure 4.1, we observe that the increase on cache footprint incurs performance overhead due to register allocation. Analyzing ToBBA-SRF's performance improvement, we can observe that the programs with higher improvement roughly correspond to the programs with the smaller increase on weighted cache footprint. For instance, 'rijndael' is the program with the highest cache footprint, and it is also the program that measured the worst performance improvement (-4.64%). The performance improvement vs. cache footprint relation is not linear, as we can observe with 'susan corners' (47.56% of improvement). Susan corners is heavily data flow but with deep control flow for the n variable inside its outer loop. The register holding n is spilled frequently without the SRF. By removing these spills, performance considerably increases as we can see in Figure 5.6.

Programs that are not very sensitive to register allocation incur in very low per-

Figure 5.6 – ToBBA-SRF performance improvement over TBB implemented with additional spill instructions. The measured geometric mean for large data set ('geomean' in the chart) is 13.81%.

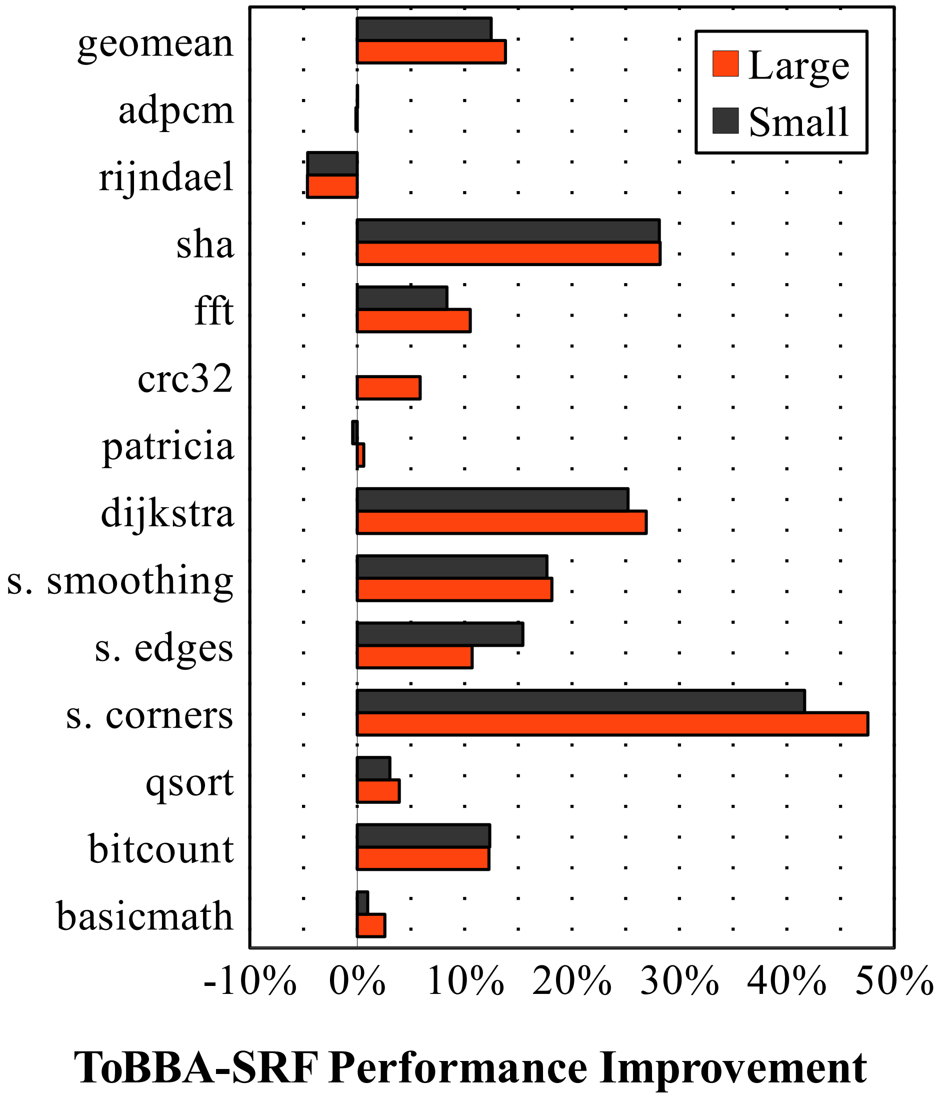
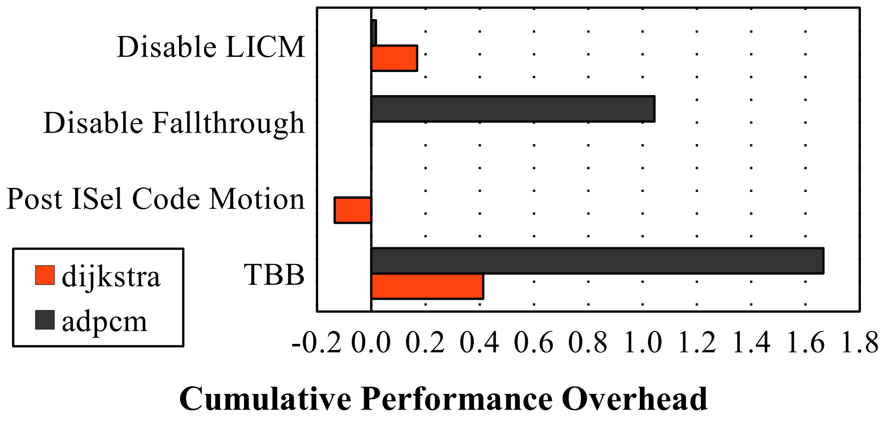


Figure 5.7 – Cumulative performance overhead off all modifications in the original program.



formance overhead. For instance, 'susan corners', 'qsort', and 'patricia' present an overhead of 1.03, 1.05, and 1.01, respectively. Considering the programs that are close to the mean, e.g., bitcount (1.20) and 'rijndael' (1.25), their overhead are yet still lower than the techniques published so far for error correction. Differently from any technique in the literature, two programs exhibit **performance speedup**: 'basicmath' (0.97), and 'fft' (0.92). For these two programs, the new register allocation favored a better register usage, leading to slight yet desirable speedup.

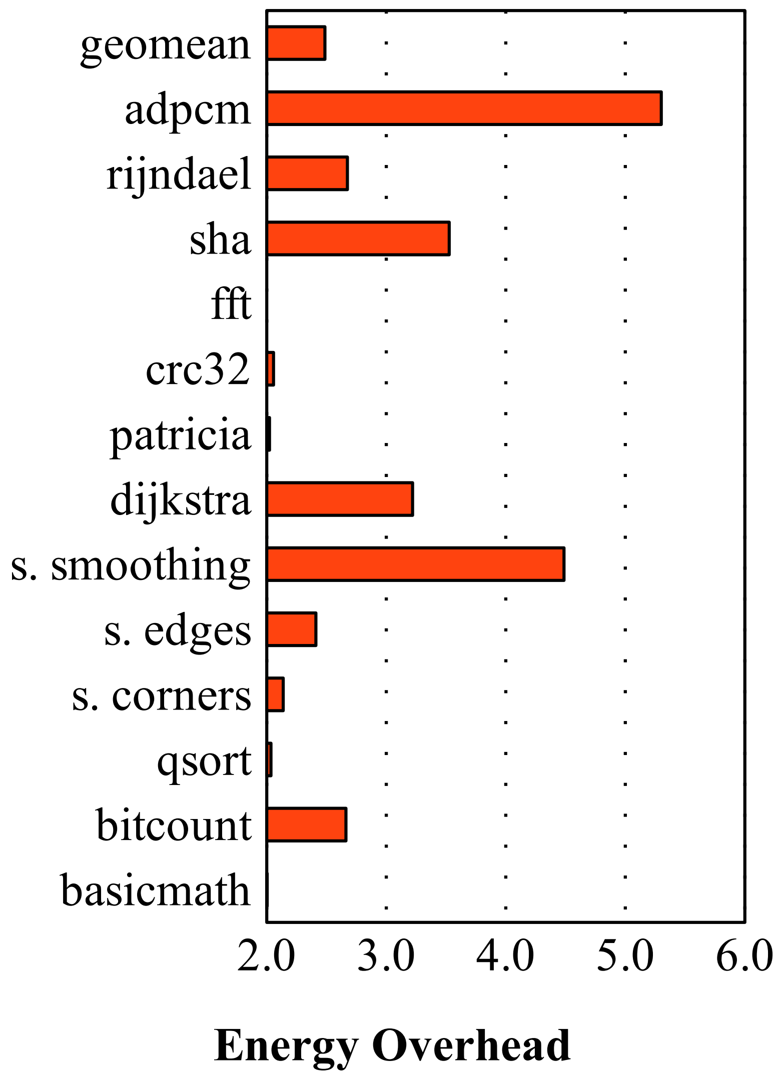
Although ToBBA-SRF's removal of spill instructions reduce overhead, there is inherent overhead due to the modifications in code structure. Figure 5.7 shows the cumulative performance overhead of every modification in the LLVM compilation pipeline for two programs that exhibit considerable overhead ('dijkstra' and 'adpcm'). The 'Post RA Code Motion' is not reported because it did not incur in overhead for these two programs. Much of the overhead comes from the TBB scheduling due to the modifications it imposes in register allocation. The removal of unconditional branches (disable fallthrough in the chart) incurs heavy overhead for adpcm because it has a loop with several iterations that contains a high number of control flow statements.

5.7 Energy Consumption

Figure 5.8 shows the energy consumption overhead of the TBB considering a full duplicated Working Register File. These results overestimate the actual energy consumption, because the SRF has much less access to it than the WRF has. Unfortunately, it was not possible to model the SRF in McPAT. However, these results give a rough idea how the TBB would behave in terms of energy consumption overhead. The measured geometric mean overhead is 2.48, still less than a TMR arrangement, which would incur at least an overhead of 3.

The highest energy overhead observed was for the 'adpcm' program with 5.30. The smallest one was 'basicmath' with 2.003. Notice that 'basicmath' was one of the programs that presented performance speedup with the SRF. Therefore, we can expect that the actual energy consumption overhead would also be smaller than 2, leading to energy savings. This was the case with 'fft', which has an energy overhead of 1.65. This result is even better than a DMR arrangement.

Figure 5.8 – TBB's energy overhead. There results overestimate the SRF usage, because it counts twice the Working Register File energy consumption. The SRF would consume much less energy than the WRF due to less frequent access to it.



The energy consumption overhead is a difficult measure to compare with others in the literature because none of the published techniques report energy. However, it is possible to expect that the other techniques would incur higher energy overhead because of their higher performance overhead.

5.8 Stuck-At Outcome in ToBBA's Rollback Machinery

ToBBA can be divided into two main components depending on where the fault occurs: (1) the duplicated RISC cores and their pipelines; and (2) the rollback machinery. This section analyses permanent faults in (2), because permanent error analysis in DMR systems is a well understood topic in the literature.

ToBBA's rollback machinery in its current form is not protected, i.e., it does not employ redundancy to protect itself against faults. Even in its current design, we expect that it does not propagate or produce an observable error when any of its signals is stuck-at-0. On the other hand, we do expect the rollback machinery to produce an observable error when any of its sensitive signals is stuck-at-1. The rollback machinery has five critical components besides the comparators:

1. **isFaultyStore:** flags if the erroneous instruction is a Store, i.e., the TBB was executing its 'transaction' region and an error was detected. This signal is used to set the register file into the 'read-only' mode;

2. **'TMR Addr' register:** contains the address where the execution has to rollback to;

3. **forceReset:** resets the two cores, flushing the pipeline, and setting the program counter back to the value stored in the 'TBB Addr' register;

4. **isRecovering:** is a signal that while in 1 the two RISC cores are freezed;

5. **faultDetected:** flags if there is a deviation between the two RISC cores, this is the rollback's output signal.

The way the rollback works makes it immune to several stuck-at faults, depending on which signal has the stuck-at error. The most critical one is the 'faultDetected' signal: (1) in case of a stuck-at-zero, the rollback might produce false negatives if

Table 5.4 – Stuck-at outcome for each rollback internal signal

Signal	**Stuck-at**	**Outcome**
isFaultyStore	0	Not observed
	1	Detected
'TBB Addr' register	0	Not observed
	1	Not observed
forceReset	0	Not observed
	1	Detected
isRecovering	0	Not observed
	1	Not observed
faultDetected	0	Not observed
	1	Detected

there is actually an error besides the stuck-at in this signal; (2) in case of a stuck-at-one, the rollback will produce false positives even if the two cores do not have an error. In fact, this is the catastrophic error in ToBBA, because a stuck-at-one in the 'faultDetected' signal would lead the architecture to stall in the rollback.

The other signals are less critical than the 'faultDetected' one. In case there are either a stuck-at-zero or a stuck-at-one in any of them, these errors do not propagate to the 'faultDetected' signal. The problem arises if there is a soft error in the architecture, leading the 'faultDetected' signal to use wrong data due to the previous stuck-at fault. However, besides the 'TBB Addr' register, each of the other signals are just one flip-flop. They could be TMR'ed at negligible cost.

If there is either a stuck-at-zero or a stuck-at-one in any of the comparators that make the two cores to deviate from each other, the 'faultDetected' signal will flag it, and the architecture will also stall in the rollback. A stuck-at in the comparators can only be detected by an observer external to the rollback machinery.

Table 5.4 shows the outcome of the injected stuck-at-zero and stuck-at-one in every internal signal of ToBBA's rollback machinery.

Injecting a stuck-at fault in the 'TBB Addr' register and in the 'isRecovering' signals does not create an observable error. This is because the values of these signals are just used when a soft error is detected by the rollback. A stuck-at-0 during a soft-error free run does not influence the RISC core state. On the other hand, the injected stuck-at-1 on the forceReset, isFaultyStore, and faultDetected signals do cause ToBBA to produce a wrong behavior. This is because these three signals change what the RISC core is doing at the moment. The stuck-at-1 fault in the ‘isFaultyStore’ signals can be indirectly detected by the change in the register file mode from ‘write permission’ to ‘read only’. Therefore, ToBBA’s rollback machinery works as it should even in the presence of permanent errors.

6 RELATED WORK

6.1 Literature Organization

The literature review starts with fault tolerance techniques for soft error detection and recovery implemented in software only. *Software-Implemented Hardware Fault Tolerance* (SIHFT) techniques operate in two granularities: instructions or threads. In case of instructions granularity, SIHFT techniques protect either control-flow or data-flow. SIHFT techniques are reviewed in Section 6.2.

The second part of the review discusses fault tolerant techniques implemented in hardware. In the hardware review, the focus is microprocessor fault tolerance, thus the discussion will revolve around on how to enhance microprocessors with soft error detection and recovery. The hardware techniques are reviewed in Section 6.3.

The third and last part is devoted to techniques that use a combination of software and hardware fault tolerance techniques. The idea behind them is to use the best of software and hardware techniques to improve fault coverage, and possibly reducing the associated overhead. Those techniques are usually called *Hybrid* in the literature, and are reviewed in Section 6.4.

In the end of each part, a subsection named 'Comparison with the Stack' discusses how the Transactional Stack positions itself in the literature.

6.2 Software-Based Techniques

6.2.1 Control-Flow Error Detection

A **control-flow error** (CFE) occurs when either a wrong or illegal branch is executed. An executed branch of the program is said to be *legal* if and only if it is an existing branch instruction in the program and the condition to execute it is satisfied; a branch is said to be *wrong* if the executed branch exists as an instruction in the program but its condition to execute cannot be satisfied; and an executed branch is *illegal* if its instruction does not exist in the original program (i.e., a non-branch instruction was transformed into a branch during program execution).

By definition, a CFE cannot exist if the program execution is not corrupted: an illegal branch cannot exist in a correct program execution; a wrong branch cannot exist because it is always possible to satisfy the logical conditions of all branches if program execution is correct. A CFE can be created by a radiation-induced SEU in three scenarios: i) a non-branch instruction being executed changes into a non-valid branch, i.e., the operation code data is corrupted; ii) the target address of a valid branch is corrupted; and iii) one of the variables composing a logical expression that activates a branch is corrupted. Scenarios i) and ii) lead to an illegal branch, and scenario iii) leads to a wrong branch.

The detection of transient CFE was established in the literature with techniques that check assertions during runtime. The general idea is to compute signatures identifying each basic block, and checking the signatures generated during compilation and runtime. If they do not match, an error is signaled. Control-flow errors were firstly identified by the usage of watchdog processors, which are intrusive in the hardware design (Saxena and Mccluskey, 1989). These hardware intrusive techniques are discussed in Section 6.3.1.

SIHFT techniques for CFE detection based on the signature-checking scheme in software such as the Control-flow Checking Approach (CCA) (Kanawati et al., 1996) have appeared, but with a coverage rate of only 38% and a performance overhead of 1.5. The CCA works by identifying branch-free regions in the source code, and proceeds by inserting assertions in the entry and exit points of these regions. By doing so, it is possible to detect a jump to an incorrect branch-free region. As pointed out by Alkhalifa et al. (1999), due the overhead of inserting redundant code and checkers into the branch-free regions, CCA can incur in undetected control-flow errors, as well as in overhead on execution time. The same authors have proposed a further extension of CCA, the Enhanced CCA (ECCA) (Alkhalifa et al., 1999), in order to reduce the overhead of redundant code. By checking only the jumps between branch-free regions, both CCA and ECCA fail to detect errors within such a region. Moreover, they cannot correct an error once an error is detected.

The technique Control-Flow Checking by Software Signatures (CFCSS) detects control-flow errors by comparing signatures of the basic blocks generated at compile-time with the ones computed at run-time (Oh, Shirvani and Mccluskey, 2002). CFCSS is also based on inserting redundant checking code on control variables, and, as pointed out by Goloubeva et al. (2003), it may fail to detect control-flow errors when a basic block has multiple preceding blocks due to the effect the authors called aliasing. The CFCSS technique incurs in overhead of approximately 1.5 on execution time and program size. The Yet Another CCA (YACC) is another technique which is comparable to CFCSS, but, as claimed by its authors, it incurs slightly less overhead than ECCA (Goloubeva et al., 2003).

The most efficient CCA technique for control-flow error detection is the Control-flow Error Detection through Assertions (CEDA) (Vemu and Abraham, 2006). Albeit being similar in nature to the techniques presented above, CEDA incurs in a small overhead for most of the case studies used to validate it - excluding the worst case situation, it causes roughly 1.2 overhead on the execution time.

CEDA was the preceding work of the ACCE method, the Automatic Correction of Control-flow Errors, proposed by the same authors of CEDA in (Vemu, Gurumurthy and Abraham, 2007), which incurs in approximately 1.2 of overhead in execution time to produce in average 70% of correct answers in fault-injection campaigns. However, ACCE is not capable of correcting errors that occur within a basic block, i.e. illegal jumps inside the same BB; hence, the use of complementary techniques is required. When ACCE is enhanced with data-flow correction, its coverage rate achieves an average of 91.6%, but the performance overhead is significantly higher.

6.2.2 Data-Flow Error Correction

A **data-flow error** (DFE) is caused by a soft error that corrupts variables within a basic block. A DFE might lead to erroneous results or even to a CFE, in case the corrupted variable is used in a logical expression or assertion guarding a branch. Techniques for DFE detection and correction can be classified into *algorithm-specific* or *general-purpose*. Algorithm-specific methods offer higher fault coverage, usually incurs in less performance overhead, but, as the name say, can only be used in a very specific application scenario.

Algorithm-Based Fault Tolerance (ABFT) is a technique devised to protect matrix operations against transient hardware faults (Huang and Abraham, 1984). ABFT explores specific properties of the matrix operation being hardened to achieve fault-tolerance. ABFT is capable of detecting and correcting single errors occurring in matrices. As originally ABFT depends on the algorithm being protected, ABFT is not a general-purpose SIHFT technique, but this seminal work has been used as basis for further research on SIHFT. Moreover, ABFT is concerned with the data-flow of the matrix operation algorithm, and is not applicable to protect it against CFE. Itturriet et al. (2012) have implemented ABFT in a dedicated hardware design, showing the ABFT feasibility beyond SIHFT. ABFT was even evaluated in new Graphics Processing Unit (GPU) cards, a hardware device that was not even imagined when ABFT was designed. In the case of GPU reliability, the On-Line ABFT allows for error detection in parallel with the matrix multiplication computation (Ding et al., 2011).

Decimal Hamming (Argyrides et al., 2011) is a SIHFT technique that applies Hamming in program variables for a class of programs where the program's output is a linear function of the input. Although with a limited application class of programs, Decimal Hamming provides interesting results in terms of performance overhead of, in average, 1.05 for protecting a Linked List. However, the performance overhead reported to protect a Hash Table is of at least 1.4. Thus, Decimal Hamming is sensitive to the program being protected, and should be carefully used to avoid jeopardizing application's performance.

Program checking (Blum and Kannan, 1995) is a technique to check if the results a program has computed are correct or not. For program checking be feasible, the checking mechanism must be asymptotically smaller than the algorithm being checked. Otherwise, program checking would be equivalent as recomputing the results. A class of program checking that has drawn attention is software invariants.

Software invariants can be used to detect errors in the data caused by soft errors through the automatic detection of pre- and post- conditions, and loop invariants of programs (Rebaudengo, Reorda and Violante, 2003). This method is based on state-of-the-art invariants detection tools, such as Daikon (Ernst et al., 2007). Despite the low overhead imposed by this technique, its detection rate of soft errors is low (Pytlik et al., 2003). For instance, Pattabiraman, Kalbarczyk and Iyer (2007) report a maximum coverage of 75% in their invariant checking technique. Consequently, invariant checking requires the adoption of complementary techniques. In addition, good invariants are extremely hard to find automatically, because in the general case this is an undecidable problem (Blass and Gurevich, 2001).

An important class of data-flow techniques is *instruction duplication*, which, as the name says, explores data diversity by duplicating the executing instructions and adding some comparator mechanism to check if the two copies executed correctly. The seminal work on instruction duplication is the ED4I method (Oh, Mitra and Mccluskey, 2002). In ED4I, checking points are created in the store and branch instructions, so that these two classes of instructions are duplicated and have their results checked after they finish. ED4I increases the probability of error detection by applying a multiplicative integer constant in the copies, so that the duplicated instruction uses diverse data than the copied instruction. An important issue with ED4I is that that multiplicative constant might cause data overflow, so a careful design is necessary when applying ED4I. The ED4I paper does not report any results on performance overhead, but it is commonly accepted that instruction duplication incurs in performance overhead of at least a factor of 2 (Rebaudengo et al., 1999).

6.2.3 Redundant Multi-Threading

Another SIHFT approach is to perform soft error recovery by redundant multi-threading (RMT) (Mukherjee, Kontz and Reinhardt, 2002). In RMT, two threads execute in parallel, the leading and the trailing ones. RMT explores time redundancy by executing one thread ahead of the other, and by comparing their computed results at the end. RMT has an average overhead in execution time of approximately 1.4 compared to the execution of the single threaded version (Mukherjee, Kontz and Reinhardt, 2002). Furthermore, RMT requires the application that is being hardened to be designed to take advantage of thread level parallelism. Another issue is that the *entire* thread is replicated, which causes additional performance overhead due to register pressure and higher core occupation. An approach that replicates only a selected portion of the code instead of the entire thread as in RMT is Selective Replication (VERA ET AL., 2010). Although less resource consuming than RMT, Selective Replication still has the problem of execution checkpointing and rollback.

6.2.4 Compiler-Guided and Program Transformation Reliability

The Architectural Vulnerability Factor (AVF) (MUKHERJEE ET AL., 2003) is a metric to estimate the probability that the bits in a given hardware structure will be corrupted or not by a soft error when executing a certain application. The AVF is calculated as the total time the vulnerable bits remain in the hardware architecture. For example, the register file has a 100% AVF, because all of its bits are vulnerable in case of a soft error. This metric is influenced by the application due to liveness: for instance, a dead variable has a 0% AVF because it is not used in computation. Although not a technique by itself, AVF is an important metric for the following discussion.

The AVF is a general metric to compute the hardware vulnerability, and it can be refined if the hardware structure itself is taken into account when computing the AVF. The Register Vulnerability Factor (RVF) (YAN AND ZHANG, 2005) refines the AVF to the register file. Given the knowledge extracted from the RVF metric, Yan and Zhang (2005) propose a simple instruction scheduling program transformation to enhance software resilience. Because the register file is more vulnerable during read-after-write and read-after-read operations, if a soft error corrupts the first operation, the erroneous value will be committed in the computation. To solve this issue, the proposed code scheduling delays all write operations and anticipates the read ones. Albeit being interesting to understand the register file vulnerability, the proposed instruction scheduling mechanism based on the RVF is not efficient for software resilience, as the authors report due to low coverage (YAN AND ZHANG, 2005).

The Instruction Vulnerability Index (IVI) (REHMAN ET AL., 2011) is a refinement of the RVF metric that considers any hardware structure, not only the register file. Similarly to (Rouf and Kim, 2010) discussed next, Rehman et al. (2011) also report that one important factor for vulnerability is loop-unrolling, and they present an algebra that helps finding the best degree of unrolling. Jones and O'boyle (2008) also report similar results.

Rouf and Kim (2010) evaluate how the GCC built-in optimization passes influence the vulnerability to control-flow errors. The authors show that loop modification techniques have big impact on vulnerability, and they report that loop-unrolling is the transformation that most impact vulnerability. In fact, the compilation process should be carefully designed because software resilience and coverage are very sensitive to compiler optimizations. Parizi et al. (2013) analyze how the ACCE technique behaves when a program is firstly optimized with different program transformation passes. ACCE was implemented in the LLVM compiler, and the LLVM IR code was protected with ACCE. As expected, the same program can exhibit totally different fault coverage depending on the LLVM IR optimizations applied before ACCE.

Encore (Feng et al., 2011) is a fault recovery technique based on program static analysis. Encore computes idempotent regions in the control-flow graph that do not have write after read dependencies. With these regions identified, the architecture can rollback to the previous checkpoint with very low latency. The reported average performance overhead for error recovery is 14%. Encore assumes an error detection mechanism with an error detection latency of 10, 100, and 1,000 cycles, citing ReStore (Wang and Patel, 2005) as such a mechanism. However, ReStore only detects 50% of the injected soft errors within a error detection latency window of 100 cycles, forcing Encore's error correction coverage to be much lower than 50%. Another issue with Encore is that it is not capable to recover control-flow errors.

SWIFT (Reis et al., 2005) is a soft error detection technique that duplicates all instructions in the program and checks if the store instructions executed correctly. If the store is wrong, computed data are wrong and the error is detected. The measured geometric performance overhead was 1.41 for an Itanium 64 (IA64) architecture. Because the instructions are duplicated, in an architecture with smaller issue width the overhead would be higher. This result means that SWIFT is using unused resources of the processor. Notice that the Itanium is a 6-issue core, with 2 Integer

ALU, and 128 general purpose registers (Sharangpani and Arora, 2000). By executing a single application at a time, the register file will not be under pressure even when cutting it by half to hold the duplicated instructions.

FASER (Xu et al., 2013) builds up on SWIFT, i.e., it also duplicates all the instructions inside the basic block and add checking instructions to compare their results, to enable error recovery, which is also based on the computation of live registers for each basic block. However, SWIFT and FASER overhead would be much higher than the reported one for an embedded architecture. In this dissertation, the instructions are not duplicated, reducing the register pressure effect, and requiring a smaller register file. In fact, in the performance experiments we have used a register file with 32 registers, the usual size for embedded computing. Notice that both SWIFT and FASER assume that the register file is protected with ECC, thus the power and area overhead would be much higher than ToBBA given that these techniques would have to protect a register file with 128 general purpose registers. Finally, both FASER and SWIFT mantain a ghost copy of the register file, which duplicated instructions operate over. Therefore, these two techniques require a duplicated register file, both protected with ECC.

6.2.5 Comparison with the Stack

Although the Stack is not a pure SIHFT technique, much of the motivation behind the TBB comes from the SIHFT works. The most important of them is the findings of Yan and Zhang (2005) about register liveness. All the TBB design was created with the goal of reducing RVF to a minimum.

Compared to the existing CFE handling techniques, the TBB is the only software construct capable of providing a small and cheap checkpointing mechanism for error recovery and elimination. The ACCE technique by Vemu, Gurumurthy and Abraham (2007) fails to correct the error, because ACCE is focused on recovering program execution to the BB where the error has occurred. However, after execution is restored some rollback mechanism is necessary to recover the program's state. On the other hand, error detection is well established and provide reasonable error detection coverage with acceptable performance overhead. CEDA (Vemu and Abraham, 2006), e.g., correctly detects the CFE in program execution.

For DFE error handling, the general-purpose techniques that incurs in small performance overhead do not offer high fault coverage. Instruction duplication, which is the general-purpose technique with highest coverage, is performance hungry. In that matter, the TBB overhead is much below instruction duplication. On the other hand, algorithm-specific techniques perform much better than the TBB for obvious reasons. A solution is to use ToBBA coupled with accelerators, for which algorithm-specific hardening solutions exist. Ferreira et al. (2014) put ToBBA and a matrix accelerator hardened with ABFT (Itturriet et al., 2012) together, using ToBBA as a small core that dispatches heavy computation to the accelerator.

In terms of performance overhead, the TBB is aligned with the error detection schemes discussed in this section, the exception being RMT. RMT can take advantage of sparing cores in the architecture to duplicate the threads. However, RMT incurs in the heavy checkpointing costs we have discussed in Chapter 1.

6.3 Hardware-Based Techniques

6.3.1 Control-Flow Monitoring

One of the first approaches for control-flow monitoring was the *watchdog co-processor*, a hardware device that can listen to the executed instructions in the microprocessor, memory or the bus in a multiprocessor system. Saxena and Mc-cluskey (1989) propose a watchdog approach where each basic block is enhanced with a checksum, created by sending a checksum before the first instruction of each BB and the checker after the last instruction. The error recovery latency is determined by the number of instructions of each BB, as it is the case in ToBBA. Saxena and Mccluskey (1989) are the first to show that this overhead grows linearly with the BB length. Performance overhead is not discussed in that paper.

Bernardi et al. (2005) propose an approach where each basic block has a signature generated during compilation as CEDA (Vemu and Abraham, 2006), coupled with a hardware module called 'Pandora' which is a simplified watchdog co-processor that only listens to the processor bus and it is just activated by a software-level call. Pandora checks the signature associated with the current BB to the one of the next BB as computed by the executed branch. If there's any mismatch, i.e., a wrong branch is computed based on the signature checksum, an error is detected. Bernardi et al. (2005) report an average error detection rate of 99% that the authors only compare with SIHFT techniques. It is reported an average performance overhead of 50% for the micro-benchmark applications used in the paper. However, this overhead tends to be bigger, because, e.g., in the matrix multiplication benchmark, it was used a really small instance of 3×3.

Chaudhari, Abraham and Park (2013) implement the ACCE (Vemu, Gurumurthy and Abraham, 2007) technique in hardware, but they claim to have solved the error recovery issue, i.e., how to rollback architectural state. The signatures of each BB are computed by instrumenting the binary by reconstructing its Control-Flow Graph. These signatures are then stored in memory together with the original application binary, but not interleaved. With this data, a Signature checking hardware module can verify the instructions as ACCE does. To enable error recovery, the architectural state has to be checkpointed. Chaudhari, Abraham and Park (2013) duplicate the register file and require both copies to be synchronized, which is probably an issue in case of timing errors. The other part of the architectural state is the main memory. The authors introduce the 'Write Delay Buffer', which is an 8-entry buffer inserted between the pipeline and the cache storing the value and the address of any data that must be written in memory. If an error is detected in the current BB, the Write Delay Buffer is flushed, and the register file is rolled back. The re-execution would than try to hit the value in the buffer, a miss is generated, and the correct value is filled from memory. In case a BB has more than 8 memory writes, that BB needs to be split. This splitting could introduce register spills, but implementation details are not given in the paper. The authors also assume that the signatures are residing in the L1 cache, otherwise the cost to fill a register with the signature would be higher. Chaudhari, Abraham and Park (2013) report an average error detection coverage of 99.977%, an area overhead of 5.81%, and a performance overhead of 1%, although an overhead of 1% for MiBench given all additional register spills and fills seems overoptimistic.

Some approaches implement the watchdog processor using the available debug machinery of the microprocessors, a debug port for instance. In this way, the watchdog can be implemented without any modification in the microprocessor. Du et al. (2013) evaluate this approach in a miniMIPS core using the debug port as the watchdog. In that work, the software is not modified, but the enhanced debug port called CFC module has to compute the signatures for each BB as the previous watchdog approaches. Du et al. (2013) do not mention how these signatures are computed, but probably the application binary or the source code is analyzed beforehand as in Chaudhari, Abraham and Park (2013).

6.3.2 Checkpointing

ReVive (Prvulovic, Zhang and Torrellas, 2002) is a technique for lightweight checkpoint and rollback in multiprocessor systems which works by protecting the checkpoint data with parity, and a logging scheme to store the processors' state between two checkpoints. ReVive implements all the checkpointing and logging mechanism in software, requiring just few changes in the memory controller. An average performance overhead of 6.3% is reported, with the worst-case overhead of 22% for a Fast Fourier Transform algorithm. The problem with ReVive is that the performance overhead of the error detection mechanism is not considered, while it is known that the error detection overhead is the most important factor of overhead in fault tolerant systems, because the error detection scheme is always executed even in error-free runs. Chapter 1 discussed that his overhead is quite significant in checkpointing and rollback approaches (Chen and Yang, 2013), thus it is expected that ReVive has a much higher overhead than the reported one. Another problem is the high error recovery latency of 400 ms in average. This latency may be acceptable in data-centers and another data-centric application, but it is very high in a real-time computing scenario.

The SoftWare Anomaly Treatment (SWAT) method is a symptom-based error detection approach for multicore systems based on checkpointing and rollback for error recovery (Li et al., 2008), i.e., exactly what ReVive misses. SWAT was designed to detect permanent faults that manifest as a fatal hardware trap, abnormal software termination by the operating system, operating system hangs and abnormal activity. In SWAT, an error is detected by interrupts causing one core to present the mentioned symptoms. It is reported that, for these permanent errors, 100,000 cycles are necessary to detect the error, i.e., receive the interrupt and check if the architectural state is correct. Li et al. (2008) have also performed transient fault injection and have shown that 10 million cycles are necessary until the error can be detected. SWAT was further extended by (Sastry hari et al., 2009) to allow multi-threading applications. These error detection latency results are in pair with the discussion made in Chapter 1 based on Chen and Yang (2013).

FaulTM (Yalcin, Unsal and Cristal, 2013) uses transactional memory to provide fault-tolerance at the thread level through execution checkpointing. In FaulTM, a thread is duplicated and the two versions execute the same set of instructions in two different cores. Right before a thread executes a memory store, the read/write instruction sets are compared, and if they do not match an error is signaled. The problem with this approach is the rollback scheme. The error detection takes place right before a store instruction by comparing the register set of the two threads and their write sets. Because a thread usually executes thousands of instructions interleaved with load and arithmetic ones, the FaulTM would have to re-launch the faulty thread to some unknown region of its code section, which is clearly not feasible, because the authors claim that the rollback is simply to relaunch the thread. In addition, when the thread is relaunched, it is necessary some mechanism not mentioned by Yalcin, Unsal and Cristal (2013) to recover the register file and the memory to its state before the first execution of the erroneous code chunk, otherwise the thread will use invalid memory and register values leading to erroneous computation.

6.3.3 Instruction Replay

Instruction Replay (or Retry) is an error detection and recovery technique where some functional units of the core are duplicated and their results are compared for correction. In Replay schemes, the error detection granularity is the instruction itself, where its register operands and the architectural state are checked for error. In case an error is detected, the recovery mechanism is simply to flush the caches and the pipeline, recover the architectural state, and re-execute the instruction where the error was detected. In Replay schemes, the replicated units have to work in **strict lockstep**, i.e., timing deviations are now allowed, otherwise the detection and recovery mechanism in the instruction granularity would not work.

Kim and Shin (1996) apply Instruction Replay in a reliable version of the S/390 IBM architecture which targeted to main-frame applications during the 90's. In that work, the instruction unit (instruction fetch, decode, address generation, and operand fetch), the fixed point unit, and the floating point unit are duplicated, similarly as in ToBBA. Although not duplicated, the register file needs to be bigger than the one in a not-hardened architecture because all the architectural state is checkpointed and stored in it. In addition, both the register file and the stored data inside it are protected with ECC. The reported S/390 design achieved 400 MHz dissipating 37 Watts of power. The crucial aspect that work is that the S/390 microprocessor has to execute in strict lockstep, which is not a feasible assumption due to aggressive transistor scaling (Bernick et al., 2005).

Intel has recently introduced an Instruction Replay mechanism on its Itanium 9500 processor family which is not based on full comparison of the architectural state (Bostian, 2012). Intel relies on ECC, parity code, and arithmetic residues to perform error detection, plus a duplicated instruction buffer to hold an extra copy of all instructions that go through the pipeline. In the best scenario, where the erroneous instruction is just replayed once, and where not that many instructions are flushed and re-executed, Bostian (2012) reports an error recovery latency of seven cycles. Intel does not disclose power, area, and performance degradation due the error detection scheme.

6.3.4 Pipelining

DIVA (Austin, 1999) is an architecture where its pipeline checks the integrity of executed instructions before they commit. The checking mechanism receives as input the instructions from the reorder buffer of the execute stage and their inputs and outputs. The checking re-executes the instruction, and if there is any mismatch, DIVA signals an error. DIVA assumes the checking mechanism is correct, making it unfeasible for realistic critical systems. Razor (Ernst et al., 2003) extends DIVA to allow dynamic voltage scaling by introducing the shadow latch, but the problem with the rollback checking mechanism is not solved.

Selective replication (Nakka, Pattabiraman and Iyer, 2007) is an error detection technique based on duplicating some portions of the pipeline (fetch, rename and commit) of a superscalar architecture to reduce the overhead associated with full core duplication. The idea is to fetch multiple copies of a reliable instruction, rename their registers in an augmented register alias table, and vote them at the end in the commit stage increased with voting logic. Selective replication has an average overhead 53.1% less than full pipeline duplication, making it 16.5% in average slower than the unreliable baseline architecture. Reported error detection coverage is 97% for data-errors and 62.5% of the manifested control-flow errors.

6.3.5 Comparison with the Stack

In this section and in Section 6.4.1, it is important to emphasize the difference between the two research communities that we have analyzed the literature, as well as the implications on methodology and the evaluation of their research. In short, the on-line test community is interested on high fault coverage, even if it incurs in higher performance overhead to obtain it. On the other hand, the computer architecture community is interested on performance, even if it incurs in lower fault coverage. The on-line test community performs fault injection experiments based on the low-level RTL description, and very often research and prototype microprocessors are used (e.g., miniMIPS) what makes almost impossible to execute full benchmarks such as MiBench on them. The computer architecture community does not even perform fault injection, but uses cycle-accurate performance simulators (e.g., gem5) to evaluate their proposal. The implications are twofold: i) micro-benchmarks hide the actual performance overhead, but do not influence on fault coverage results; ii) cycle-accurate simulators effectively measure performance, but assumptions about error detection and recovery make it impossible to draw fault coverage numbers.

The control-flow monitoring techniques of Section 6.3.1 target the error detection of CFE's with the goal of low-latency and assertive communication of the error occurrence. The (correct) assumption of these techniques is that undetected errors are catastrophic in mission-critical systems, such as space and automotive, thus any erroneous state must be identified. The problem is that some systems allow a reasonable mean time to repair so that the system can be reset after an error is detected. However, hard real-time and reactive applications do not allow it, e.g., an automatic guidance system. The Stack attains a similar overhead of control-flow monitoring when target-dependent optimizations are turned off, but it is important to remind that the Stack is also recovering the detected errors.

Checkpointing is a technique mostly proposed by the computer architecture community, and all works target servers and high-scale distributed systems. However, in the Stack a sort of lightweight checkpointing mechanism is performed when the rollback destination is stored in the 'TBB Addr' register. In the Stack, the store instructions are checkpointed in the software itself instead of in additional memory areas.

Instruction Replay is similar to ToBBA's error recovery. The (very important) difference is that ToBBA is arranged as a **loose lockstep** DMR, which allows for timing faults and deviations. The Intel implementation of Instruction Replay duplicates internal buffers, which are very sensitive to upsets. In the Intel's scenario this is not much of a problem, because the additional area and power overhead can be incorporated given that the Itanium family targets main-frame systems.

Another difference with Instruction Replay is that the TBB's error recovery mechanism do not actually need the comparison through all the pipeline stages. In fact, a checking mechanism that only verifies if the store instructions have executed correctly would suffice, as SWIFT (Reis et al., 2005) does. In instruction replay, an auxiliary hardware would have to compute and store the regions without write-after-read dependencies (i.e., idempotent regions) as Encore (Feng et al., 2011) does. It would also be necessary to store a snapshot of the register file before the idempotent region starts so the Instruction Replay mechanism could rollback to. Another way to see the TBB, ToBBA, and their error recovery is like an Instruction Replay technique that has a well-defined idempotent region (the TBB), where computation can easily rollback to without any additional effort (the start of the idempotent region is stored in the 'TBB Addr' register).

Finally, pipelining is a technique also by the computer architecture community, which clearly reduces fault coverage for performance, and, in fact, fault coverage is heavily compromised. As we have discussed, DFE coverage is about 97% and CFE coverage is about 62.5%. ToBBA achieves much higher fault coverage, but it pays in a higher overhead compared to pipelining. However, claiming this performance back is the target of the future work discussed in Chapter 7.

6.4 Hybrid HW/SW Techniques

The URISC (Ultra-Reduced Instruction-Set Co-processor) (Rajendiran et al., 2012) is a co-processor that executes only a single instruction, and it was designed to tolerate against hard faults. An application written in standard many instructions is automatically converted into this single instruction format with the LLVM compiler. The URISC was not designed to execute the entire application, but only a subset of the original program that was deemed to have erroneous instructions due to hard faults. Therefore, the decision of which subset of a program contains those erroneous instructions is critical for this work, the bigger this subset is, the higher is the performance overhead to execute it in the URISC. The interesting fact about the URISC is that it does not matter the subset of erroneous instructions: all of them can be executed in the URISC by converting the program into the single instruction format given that an efficient method for finding the subset is at hand, which is not discussed by Rajendiran et al. (2012).

Stochastic Computing (Sartori, Sloan and Kumar, 2011) is a hardware/-software approach for allowing errors in computation. In this approach, the execution of programs produces results within an error margin, instead of pursuing an exact value. In the software side, the programs are transformed into a numerical optimization problem, and the program execution is given by computing the gradient descent method. The authors call the transformation application robustification. In the hardware side, there is an architecture that computes the gradient descent method and that embodies the error margin within it. The current problem of this approach is that the authors do not give an automatic program transformation that takes an imperative program and yields an equivalent one in the gradient descent form. In addition, since linear programming is known to be P-complete, the application robustification transformation is restricted to P-complete programs, as acknowledged in Sloan, Sartori and Kumar (2012). However, this approach is interesting because it is not always the case that a correct result must be always produced, e.g., in fractal computation or in soft real time applications such as video and audio decoding (Sloan, Sartori and Kumar, 2012).

HETA (Azambuja et al., 2013), a technique for control-flow error detection, employs CEDA to monitor the program control flow and use a watchdog processor to detect the errors that CEDA (Vemu and Abraham, 2006) is not capable of detecting. The authors evaluate HETA with two benchmarks, bubble sort and matrix multiplication, for which they report performance overhead of 1.08 and 1.34 for the miniMIPS architecture, respectively. In that paper, HETA is also enhanced with a technique to monitor some special cases of branches that it originally fails to detect, incurring in overhead of 1.55 and 1.43, respectively.

A hybrid technique that also copes with data-flow error detection is proposed by Parra et al. (2014). In that paper, all instructions are duplicated and their produced values are checked in order to protect the data-flow. In the control-flow side, the authors adopt the 'trace interface' as a way to simulate an watchdog processor. The technique was implemented in the PicoBlaze architecture and evaluated with three programs, including a 5×5 matrix multiplication. Performance overhead ranged from 1.96 to 2.68, while the area overhead in terms of gates is 1.4.

6.4.1 Comparison with the Stack

Two hybrid techniques from the computer architecture community and two from the on-line testing community were evaluated, all of them representative of the new ideas in the area.

Both URISC and Stochastic Computing are claimed to be general-purpose solutions that cannot be applied as such, as we have discussed before. In fact, the Stochastic Computing authors have re-positioned it as a more focused method of fault tolerance. The idea of transforming any program from any domain into another problem class for which efficient error detectors and recovery mechanism exist is not feasible, at least so far nobody has achieved it. This dissertation has started aiming at transforming any programs into a chain of matrix multiplications, which could be protected with ABFT (Ferreira, Moreira and Carro, 2010). In a general-purpose setting, that approach found the same problems as Stochastic Computing did. However, for the problems where these techniques are good, the performance overhead is almost non-existent with very high fault coverage.

On the other hand, HETA and the approach of Parra et al. (2014) are realistic on their goals to achieve high fault coverage. HETA's limitation is the applicability to error detection only, while Parra et al. (2014) present a very high performance overhead to provide error recovery. The problem with these two works is that, as we have discussed in Section 6.3.5, they are evaluated using really small data-set instances. HETA uses a 6×6 matrix multiply and a bubble sort over 10 elements. Parra et al. (2014) uses a 5×5 matrix multiply. In fact, the reason for including the 'Small' data-set in the performance evaluation of the Stack in Chapter 5 was to allow the direct comparison with these techniques. In all evaluation scenarios (with and without target-dependent optimizations, atomic and detailed gem5's models), ToBBA offers considerably lower performance overhead than these two state-of-the-art techniques.

7 FINAL REMARKS AND FUTURE WORK

7.1 Summing Up the Transactional Stack and Contributions

This dissertation introduced the Transactional HW/SW Stack for fault tolerant embedded computing. The Stack's software layer introduced the Transactional Basic Block, which creates a software container for soft errors through the forced spill of all live-out values of the original BB's. Chapter 2 discussed in details how a TBB is formed, and how a program with standard BB's can be transformed into an equivalent one with TBB's. We also discussed the implications the TBB has on error recovery latency and register liveness. The Stack's hardware layer was introduced in Chapter 3, the TransactiOnal Basic Block Architecture, a fault tolerant architecture based on loose lockstep DMR. ToBBA uses the rigid TBB's instruction ordering to efficiently do error detection and rollback. We discussed how the TBB definition avoids the register file duplication in ToBBA. Chapter 4 introduced the Spill Register File, an auxiliary hardware mechanism to reduce the performance overhead introduced by the TBB. Chapter 5 evaluated the proposal in terms of area occupation, power dissipation, energy consumption, error coverage, stuck-at sensitiveness, and performance overhead. Finally, Chapter 6 compared the Stack with existing state-of-the-art techniques.

In summary, the contributions of this dissertation are the following.

7.1.1 Compilation Strategy for Error Correction without Checkpointing

The Transactional Basic Block is a container for soft errors, and it creates a small and deterministic unit of checkpointing. With the TBB, it is not necessary to checkpoint architectural state and the memory, because errors are now allowed to propagate from the TBB to the memory.

We have introduced two strategies to generate the TBB, depending if ToBBA is enhanced or not with the Spill Register File. Without the SRF, (1) all the live-out values have to be spilled and filled when they are defined and used, respectively. This dissertation introduced the algorithms to generate the TBB in scenario (1), also considering the case of function calls. If the SRF is used, (2) the live-out values are copied to the SRF, and the additional spill instructions are not necessary. The COPY_SRF instruction is used instead. However, even in scenario (2), the TBB instruction ordering must be respected. We have presented the mean performance overhead for a MiBench subset of 1.54 in scenario (1), and 1.33 in scenario (2). In the best scenario, the measured performance overhead was as low as 1.01.

7.1.2 Reduction of Rollback Data to the Bare Minimum

With the TBB, the only data stored to rollback the execution in case of an error is the start address of the TBB. In ToBBA, this address is stored in the TMR'ed 'TBB Addr' registers. As we have discussed in Chapter 1, the amount of checkpointed data has huge impact on performance, and, in fact, the cycles required to checkpoint and recover the data need to be minimum. A study with Linux has shown that most of the system crashes occur within 10 cycles after the soft error manifested (Gu et al., 2003). ToBBA meets this stringent requirement as we have discussed in Chapter 6. The mean error recovery latency we have measured is 6.17 cycles.

7.1.3 Error Correction Without Duplicating the Register File

In terms of reliability, the register file is one of the most sensitive components of the microprocessor (Blome et al., 2006), thus, it is desirable to avoid its duplication in fault tolerant architectures. As reviewed in Chapter 6, the techniques that also use the basic block as a container of errors, e.g., FASER (Xu et al., 2013), require the register file to contain a copy of the main register file at all times. The solution is either to increase the register file size or to duplicate it. In both cases, this additional area needs to be protected with ECC. In ToBBA, the duplicated pipelines and control-logic share a single register file, reducing the sensitive area to upsets. In the scenario where the Spill Register File is used, the additional sensitive area occupied by the SRF does not need to be protected with ECC, removing the need for additional ECC encoders and decoders.

An important outcome of not duplicating the register file, or to hold ghost copies at all times, is that the energy consumption is halved. Both SWIFT (Reis et al., 2005) and FASER (Xu et al., 2013) double the number of times the register file is accessed, increasing energy consumption. With the SRF, all the live-in registers are copied in bulk only once during the TBB execution. This operation takes just one cycle and can be efficiently implemented in hardware by directly connecting the Working Register File to the SRF. Unfortunately, none of the published works in the literature evaluate the increase on energy consumption, but we can expect it to be higher than in this dissertation.

7.1.4 Spill Register File as Auxiliary Container of Rollback Data

The SRF was introduced as an auxiliary hardware to hold the live-in registers instead of spilling them to memory. The SRF does not need to be protected with ECC, avoiding additional ECC encoders and decoders. The operation COPY_SRF takes one cycle to execute and can be efficiently implemented in hardware. The SRF is directly connected to the WRF, enabling the bulk copy of the WRF to the SRF. We have shown in Chapter 6 that the SRF introduction reduced the TBB mean performance overhead from 1.54 to 1.33.

7.1.5 Implementation of the TBB Generation in a Production Compiler

The TBB generation was implemented in the LLVM compiler, making it possible to evaluate it thoroughly by using an established benchmark. As we have discussed in Chapter 6, the most important hardware- and hybrid-based techniques are evaluated for performance using micro benchmarks. Microbenchmarks do not show the real behavior and overhead of the technique, being difficult to actually assess them in a real usage scenario.

7.1.6 Comprehensive Evaluation of the Full HW/SW Stack Using an Adequate Fault Model

This dissertation is the first work in the literature, as we are aware of, to evaluate an error correction technique and fault tolerant architecture considering a broad design space. This work evaluated area occupation, power dissipation, energy consumption, performance overhead, stuck-at sensitiveness, and error coverage. We have made no assumption about error detection performance, detection latency, or types of error that might occur within the soft error spectrum.

The full evaluation is an important aspect of this dissertation. Previous works make several assumptions of the other layers in the fault tolerance stack. For instance, Encore (Feng et al., 2011) assumes an error detection latency that is not attainable to provide an adequate error coverage, besides not recovering control-flow errors; SWIFT (Reis et al., 2005) uses a register file with 128 registers; ReVive (Prvulovic, Zhang and Torrellas, 2002) assumes an error detection latency that is also not feasible. The list of inadequate assumptions about the fault model and the error detection latency make it really difficult to assess these techniques in a real usage scenario. As we have discussed, software and hardware are deeply interconnected, and we have shown that even slight changes in the software layer through the modification of the register allocation might lead to a completely different hardware usage.

The assumption that a technique exists for an aspect of fault tolerance does not assure that the full stack will work as it should. The hidden overhead behind these assumptions after combining all these separate techniques will certainly be bigger than evaluating them separately. On the other hand, in this dissertation the entire fault tolerant HW/SW stack was considered, and was comprehensively evaluated. The fault injection campaign of over 600 million injected faults is something never done before in the literature. Hardware- and hybrid-techniques usually inject 10,000 faults. Software techniques even less, around 1,000. Considering that a considerable number of these faults are going to be masked, the actual errors will certainly not tell the complete story about the fault tolerant mechanism.

7.2 Future Works

7.2.1 Implementing the Checking Mechanism in the Commit Stage for Out-Of-Order Execution

Out-of-order architectures are increasingly being used in embedded computing, but the hardware- and hybrid based techniques assume an in-order core. ToBBA could be adapted to make error detection in a similar way as DIVA (Austin, 1999) does, but instead of increasing the pipeline with an additional Check stage, the Commit stage could be compared between ToBBA's duplicated DMR pipelines. Differently from DIVA, this solution does not need to assume that the checking logic is error-free or manufactured with bigger transistors.

The challenge of executing the TBB in an OOO architecture is that the COPY_SRF instruction has to commit before any store instructions. Because the commit is in-order, the register definition and register termination segments are correctly checked and executed in-order. A solution to the COPY_SRF instruction would be to add a very small logic to the Commit stage that checks if the last committed instruction is not a store and the current one in the top of the Reorder Buffer (ROB) is a store. If this condition holds, we know due to the TBB instruction scheduling that the register definition segment has finished and that the register termination segment is about to start. At this moment, ToBBA can automatically copy the WRF to the SRF even without the need for the COPY_SRF instruction.

7.2.2 Supporting Permanent Fault Mitigation Through Selective DMR Activation

Because ToBBA has the DMR RISC cores, it can support permanent fault mitigation and execute in a degraded mode. After a permanent error is identified by reaching a defined threshold based on the number of times the same TBB is executed, the pipeline stage in which the permanent fault was detected can be turned off together with its error checking logic. In this scenario, the system designer would have to assign a level of trust in the produced data, but selectively turning off some pipeline stages may increase ToBBA's lifetime in cases it is desirable to do so. The challenge is to identify which of the RISC cores is affected by the permanent fault. This could be done by executing in the two cores small test programs that exercise specific portions of the pipeline. There are several approaches to identify these errors patterns for specific components of the microprocessor (Psarakis et al., 2010).

7.2.3 Fault Tolerant Multicore Architecture

ToBBA's stateless hardware model is interesting for fault tolerance in multicore because, in case a core fails, the thread can be migrated based only on the live-out register information. Multicore fault tolerance can be efficiently implemented in the Stack because saving the core's state is the power bottleneck in thread migration (Rangan, Wei and Brooks, 2009). In addition, the TBB has a very similar structure of the code organization used in HELIX (Campanoni et al., 2012) to perform automatic parallelization. In fact, the only data HELIX needs to execute a basic block in parallel are the live-out values. In the TBB, these data are explicit. Therefore, HELIX could be used to amortize the performance and energy costs of migrating the TBB from the failed core to another one to boost performance when migrating the threads. HELIX could also be used to improve TBB performance for error-free executions by providing a better utilization of the ToBBA cores.

7.2.4 Developing and Orchestrating Compiler Optimizations

We have discussed that the performance of TBB programs are heavily influenced on how the cache is used, with different patterns of cache footprint leading to different performance overhead. In the compiler level, an optimization that considers the cache usage could be designed to reduce performance overhead. Another approach would be to understand how a different set of optimizations affect the TBB performance. Compiler orchestration (Pan and Eigenmann, 2006) is still an open topic in the literature, but it is an interesting topic for TBB performance.

7.2.5 Performing the Radiation Test Using an FPGA

Performing the ToBBA's radiation test would allow to assess the architecture reliability deployed on its intended operating environment. The challenge behind it is to test a ASIC design using an FPGA, given that ToBBA was not manufactured. The FPGA has a different fault model than an ASIC, and the time to recover from an error in the FPGA is limited by the time required to execute the scrubbing. Another challenge is that ToBBA needs to be stopped while the scrubber executes.

7.3 List of Publications

7.3.1 Currently Under Review

Ferreira, R; Rolt, J.; Nazar, G.; Moreira, Álvaro F.; Carro, Luigi. Live-Out Register Fencing: Interrupt-Triggered Soft Error Correction based on the Elimination of Register-to-Register Communication. Submitted to the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) 2015.

Ferreira, R; Sanchez, E.; Rolt, J.; Nazar, G.; Moreira, A.; Carro, L.; Sonza-Reorda, M. Permanent Fault Detection and Diagnosis in the Lightweight Dual Modular Redundancy Architecture. Submitted to the 16th IEEE Latin-American Test Symposium (LATS) 2015.

7.3.2 Conferences

Ferreira, R; ROLT, J.; NAZAR, G.; Moreira, Álvaro F.; Carro, Luigi. Adaptive Low-Power Architecture for High-Performance and Reliable Embedded Computing. In: 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) 2014. **(Qualis A1)**

FERREIRA, RR; ROLT, J. ; NAZAR, G. L. ; Moreira, AF ; CARRO, L . Lightweight DMR for SEE Hardening in Low Power Embedded Systems. In: IEEE Nuclear and Space Radiation Effects Conference (NSREC) 2014

Ferreira, Ronaldo; KLOTZ, T.; VORTLER, T.; ROLT, J.; NAZAR, G. L.; Moreira, Álvaro F.; Carro, Luigi; EINWICH, K. Reliable Execution of Statechart-Generated Correct Embedded Software under Soft Errors. In: IEEE 17th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS) 2014. **(Qualis B3)**

Parizi, Rafael B.; **FERREIRA, RR**; Carro, Luigi; Moreira, Álvaro F. . Compiler Optimizations Do Impact the Reliability of Control-Flow Radiation Hardened Embedded Software. In: 4th IFIP TC 10 International Embedded Systems Symposium (IESS) 2013. **(Qualis B5)**

RECH, P.; AGUIAR, C. Z.; **FERREIRA, RR**; FROST, C. ; CARRO, L . Neutrons Radiation Test of Graphic Processing Units. In: International On-Line Testing Symposium (IOLTS) 2012. **(Qualis B1)**

ITTURRIET, F.; **FERREIRA, RR**; GIRAO, G.; NAZAR, G.; Moreira, AF; CARRO, L. Resilient Adaptive Algebraic Architecture for Parallel Detection and Correction of Soft-Errors. In: EUROMICRO Conference on Digital System Design (DSD) 2012. **(Qualis B1)**

ITTURRIET, F. ; **FERREIRA, RR**; CARRO, L . Fault-Tolerant Algebraic Architecture for radiation induced soft-errors. In: European Test Symposium (ETS) 2012. **(Qualis B2)**

Parizi, R ; **FERREIRA, RR** ; CARRO, L ; Moreira, AF . Impact on Reliability in the Control-Flow of Programs under Compiler Optimizations. In: Brazilian Symposium on Computing System Engineering (SBESC) 2012. **(Qualis B4)**

FERREIRA, RR; Moreira, AF ; CARRO, L . Matrix Control-Flow Algorithm-Based Fault Tolerance. In: International On-Line Testing Symposium (IOLTS) 2011. **(Qualis B1)**

ARGYRIDES, C. ; **FERREIRA, RR**; LISBOA, C. A. L. ; CARRO, L . Decimal Hamming: A Software-Implemented Technique to Cope with Soft Errors. In: International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT) 2011. **(Qualis B1)**

FERREIRA, RR; Moreira, AF ; CARRO, L . System Level Hardening by Computing with Matrices. In: EUROMICRO Conference on Digital System Design (DSD) 2010. **(Qualis B1)**

7.3.3 Book Chapters

BECK, A. S. ; LISBOA, C. A. L. ; CARRO, L ; NAZAR, G. L. ; PEREIRA, M. M. ; **FERREIRA, RR**. Adaptability: The key for Future Embedded Systems. In: Antonio C S B Filho; Carlos A L Lisboa; Luigi Carro. (Org.). Adaptable Embedded Systems. 1ed.New York: Springer, 2012, v. , p. 1-12.

FERREIRA, RR; CARRO, L . Adaptive Software. In: Antonio C S B Filho; Carlos A L Lisboa; Luigi Carro. (Org.). Adaptable Embedded Systems. 1ed.New York: Springer, 2012, v. , p. 279-304.

7.3.4 Journals

ITTURRIET, F. ; NAZAR, G. L. ; **Ferreira, R**; Moreira, Álvaro F. ; Carro, Luigi . Adaptive Parallelism Exploitation under Physical and Real-Time Constraints for Resilient Systems. ACM TRANS RECONFIG TECHN, 2013. **(Qualis B3)**

FERREIRA, RR; Parizi, R ; CARRO, L ; Moreira, AF . Compiler Optimizations Impact the Reliability of the Control-Flow of Radiation-Hardened Software. Journal of Aerospace Technology and Management (Online), v. 5, p. 323-334, 2013. **(Qualis B5)**

7.3.5 Workshops

Ferreira, R; ROLT, J. ; NAZAR, G. ; Moreira, AF ; CARRO, L . A Checkpoint-Deterministic Architecture for Reliable and Low-Power Embedded Computing. In: Workshop on Highly-Reliable Power-Efficient Embedded Designs (HARSH) 2014.

Ferreira, Ronaldo R.; ROLT, J. ; NAZAR, G. L. ; Moreira, AF ; CARRO, L . MoMa: A Radiation Hardened Architecture based on Branch-Free Control-Flow Resolution and on Transactional Data-Path Execution. In: Intel Compiler, Architecture and Tools Conference, 2013.

RECH, P.; AGUIAR, C. Z.; **FERREIRA, RR**; SILVESTRI, M.; GRIFFONI, A.; CARRO, L . Neutron-Induced Soft Errors in Graphic Processing Units. In: Radiation Effects Data Workshop (REDW) 2012.

FERREIRA, RR; AZAMBUJA, J. R. F. ; Moreira, AF ; CARRO, L . Correction of Soft Errors in Control and Data Flow Program Segments. In: HiPEAC Workshop on Design for Reliability (DFR) 2011.

FERREIRA, RR; AGUIAR, C. Z. ; Moreira, AF ; CARRO, L . Generalization of Algorithm-Based Fault-Tolerance by Program Transformation. In: Workshop on Resilient Architectures (WRA) 2011.

FERREIRA, RR; Moreira, AF ; CARRO, L . Deteccao e Correcao de Falhas Transitorias Atraves da Descricao de Programas Usando Matrizes. In: Workshop de Testes e Tolerancia a Falhas (WTF) 2010. **(Qualis B4)**

REFERENCES

ABATE, F.; STERPONE, L.; LISBOA, C.; CARRO, L.; VIOLANTE, M. New Techniques for Improving the Performance of the Lockstep Architecture for SEEs Mitigation in FPGA Embedded Processors. **IEEE Transactions on Nuclear Science**, Washington, DC, USA, v.56, n.4, p.1992–2000, Aug 2009.

ALKHALIFA, Z.; NAIR, V. S. S.; KRISHNAMURTHY, N.; ABRAHAM, J. A. Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. **IEEE Trans. Parallel Distrib. Syst.**, Piscataway, NJ, USA, v.10, n.6, p.627–641, June 1999.

ALLEN, F. E. Control Flow Analysis. In: SYMPOSIUM ON COMPILER OPTIMIZATION, 1970, Urbana-Champaign, Illinois. **Proceedings...** New York, NY, USA: ACM, 1970. p.1–19.

ARGYRIDES, C.; FERREIRA, R.; LISBOA, C.; CARRO, L. Decimal Hamming: a software-implemented technique to cope with soft errors. In: INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI AND NANOTECHNOLOGY SYSTEMS, 2011, Vancouver, BC, Canada. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2011. p.11–17. (DFT '11).

AUSTIN, T. M. DIVA: a reliable substrate for deep submicron microarchitecture design. In: ND ANNUAL ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 32., 1999, Haifa, Israel. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1999. p.196–207. (MICRO 32).

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, Washington, DC, USA, v.1, n.1, p.11–33, Jan 2004.

AZAMBUJA, J.; ALTIERI, M.; BECKER, J.; KASTENSMIDT, F. HETA: hybrid error-detection technique using assertions. **IEEE Transactions on Nuclear Science**, Washington, DC, USA, v.60, n.4, p.2805–2812, Aug 2013.

BAUMANN, R. Soft Errors in Advanced Computer Systems. **IEEE Design and Test of Computers**, Washington, DC, USA, v.22, n.3, p.258–266, May 2005.

BERNARDI, P.; BOLZANI, L.; REBAUDENGO, M.; REORDA, M.; VARGAS, F.; VIOLANTE, M. On-line detection of control-flow errors in SoCs by means of an infrastructure IP core. In: IEEE/IFIP INTERNATIONAL CONFERENCE ON DE-

PENDABLE SYSTEMS AND NETWORKS, 2005, Yokohama, Japan. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2005. p.50–58. (DSN '05).

BERNICK, D. et al. NonStop advanced architecture. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 2005., 2005, Yokohoma, Japan. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2005. p.12–21. (DSN '05).

BINKERT, N.; BECKMANN, B.; BLACK, G.; REINHARDT, S. K.; SAIDI, A.; BASU, A.; HESTNESS, J.; HOWER, D. R.; KRISHNA, T.; SARDASHTI, S.; SEN, R.; SEWELL, K.; SHOAIB, M.; VAISH, N.; HILL, M. D.; WOOD, D. A. The gem5 Simulator. **SIGARCH Comput. Archit. News**, New York, NY, USA, v.39, n.2, p.1–7, Aug. 2011.

BLASS, A.; GUREVICH, Y. Inadequacy of Computable Loop Invariants. **ACM Trans. Comput. Logic**, New York, NY, USA, v.2, n.1, p.1–11, Jan. 2001.

BLOME, J. A. et al. Cost-efficient Soft Error Protection for Embedded Microprocessors. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURE AND SYNTHESIS FOR EMBEDDED SYSTEMS, 2006., 2006, Seoul, Korea. **Proceedings...** New York, NY, USA: ACM, 2006. p.421–431. (CASES '06).

BLUM, M.; KANNAN, S. Designing Programs That Check Their Work. **J. ACM**, New York, NY, USA, v.42, n.1, p.269–291, Jan. 1995.

BOSTIAN, S. **Rachet Up Reliability for Mission-Critical Applications**: intel instruction replay technology. 2012. Technical Report — Intel Corp.

CAMPANONI, S.; JONES, T.; HOLLOWAY, G.; REDDI, V. J.; WEI, G.-Y.; BROOKS, D. HELIX: automatic parallelization of irregular programs for chip multiprocessing. In: TENTH INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION, 2012, San Jose, CA, USA. **Proceedings...** New York, NY, USA: ACM, 2012. p.84–93. (CGO '12).

CHAUDHARI, A.; ABRAHAM, J.; PARK, J. A Framework for Low Overhead Hardware Based Runtime Control Flow Error Detection and Recovery. In: IEEE 31ST VLSI TEST SYMPOSIUM, 2013., 2013, Berkeley, CA, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2013. p.1–6. (VTS '13).

CHEN, H.; YANG, C. Fault Detection and Recovery Efficiency Co-optimization through Compile-time Analysis and Runtime Adaptation. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURES AND SYNTHESIS FOR EMBEDDED SYSTEMS, 2013., 2013, Montreal, Quebec, Canada. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2013. p.22:1–22:10. (CASES '13).

DENNARD, R. et al. Design of ion-implanted MOSFET's with very small physical dimensions. **IEEE Journal of Solid-State Circuits**, Washington, DC, USA, v.9, n.5, p.256–268, Oct 1974.

DING, C.; KARLSSON, C.; LIU, H.; DAVIES, T.; CHEN, Z. Matrix Multiplication on GPUs with On-Line Fault Tolerance. In: INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING WITH APPLICATIONS, 9., 2011, Busan, South Korea. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2011. p.311–317. (ISPA '11).

DU, B.; SONZA REORDA, M.; STERPONE, L.; PARRA, L.; PORTELA-GARCIA, M.; LINDOSO, A.; ENTRENA, L. Exploiting the debug interface to support on-line test of control flow errors. In: INTERNATIONAL ON-LINE TESTING SYMPOSIUM, 19., 2013, Chania, Greece. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2013. p.98–103.

ERNST, D.; KIM, N. S.; DAS, S.; PANT, S.; RAO, R.; PHAM, T.; ZIESLER, C.; BLAAUW, D.; AUSTIN, T.; FLAUTNER, K.; MUDGE, T. Razor: a low-power pipeline based on circuit-level timing speculation. In: ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 36., 2003, San Diego, CA, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2003. p.7–18. (MICRO 36).

ERNST, M. D.; PERKINS, J. H.; GUO, P. J.; MCCAMANT, S.; PACHECO, C.; TSCHANTZ, M. S.; XIAO, C. The Daikon System for Dynamic Detection of Likely Invariants. **Science of Computer Programming**, Amsterdam, Netherlands, v.69, n.1-3, p.35–45, Dec. 2007.

ESMAEILZADEH, H.; BLEM, E.; ST. AMANT, R.; SANKARALINGAM, K.; BURGER, D. Dark Silicon and the End of Multicore Scaling. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 38., 2011, San Jose, California, USA. **Proceedings...** New York, NY, USA: ACM, 2011. p.365–376. (ISCA '11).

ESMAEILZADEH, H.; SAMPSON, A.; CEZE, L.; BURGER, D. Architecture Support for Disciplined Approximate Programming. In: SEVENTEENTH INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 2012, London, England, UK. **Proceedings...** New York, NY, USA: ACM, 2012. p.301–312. (ASPLOS XVII).

FENG, S.; GUPTA, S.; ANSARI, A.; MAHLKE, S. A.; AUGUST, D. I. Encore: low-cost, fine-grained transient fault recovery. In: ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 44., 2011, Porto Alegre, Brazil. **Proceedings...** New York, NY, USA: ACM, 2011. p.398–409. (MICRO-44).

FERREIRA, R.; MOREIRA, A.; CARRO, L. System Level Hardening by Computing with Matrices. In: EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN: ARCHITECTURES, METHODS AND TOOLS, 13., 2010, Lille, France. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2010. p.373–379. (DSD '10).

FERREIRA, R.; ROLT, J. d.; NAZAR, G.; MOREIRA, A.; CARRO, L. Adaptive Low-Power Architecture for High-Performance and Reliable Embedded Computing. In: IEEE/IFIP INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 44., 2014, Atlanta, GE, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2014. p.1–12. (DSN '14).

GOLOUBEVA, O.; REBAUDENGO, M.; REORDA, M.; VIOLANTE, M. Soft-error detection using control flow assertions. In: INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS, 2003, Boston, MA, USA. **Proceedings. . .** Washington, DC, USA: IEEE Computer Society, 2003. p.581–588. (DFT '03).

GOODMAN, J. R.; HSU, W.-C. Code Scheduling and Register Allocation in Large Basic Blocks. In: ND INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 2., 1988, St. Malo, France. **Proceedings. . .** New York, NY, USA: ACM, 1988. p.442–452. (ICS '88).

GU, W.; KALBARCZYK, Z.; RAVISHANKAR IYER, K.; YANG, Z. Characterization of linux kernel behavior under errors. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 2003, San Francisco, CA, USA. **Proceedings. . .** Washington, DC, USA: IEEE Computer Society Press, 2003. p.459–468. (DSN '03).

GUTHAUS, M. R.; RINGENBERG, J. S.; ERNST, D.; AUSTIN, T. M.; MUDGE, T.; BROWN, R. B. MiBench: a free, commercially representative embedded benchmark suite. In: WORKLOAD CHARACTERIZATION, 2001. WWC-4. 2001 IEEE INTERNATIONAL WORKSHOP, 2001. **Proceedings. . .** Washington, DC, USA: IEEE Computer Society, 2001. p.3–14. (WWC '01).

HAMDIOUI, S.; NICOLAIDIS, M.; GIZOPOULOS, D.; GRASSET, A.; GUIDO, G.; BONNOT, P. Reliability Challenges of Real-time Systems in Forthcoming Technology Nodes. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 2013, Grenoble, France. **Proceedings. . .** San Jose, CA, USA: EDA Consortium, 2013. p.129–134. (DATE '13).

HENTSCHKE, R.; MARQUES, F.; LIMA, F.; CARRO, L.; SUSIN, A.; REIS, R. Analyzing Area and Performance Penalty of Protecting Different Digital Modules with Hamming Code and Triple Modular Redundancy. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 15., 2002. **Proceedings. . .** Washington, DC, USA: IEEE Computer Society, 2002. p.95–100. (SBCCI '02).

HUANG, K.-H.; ABRAHAM, J. A. Algorithm-Based Fault Tolerance for Matrix Operations. **IEEE Transactions on Computers**, Washington, DC, USA, v.33, n.6, p.518–528, June 1984.

ITRS. **ITRS 2012 Roadmap**. 2012. Technical Report — International Technology Roadmap for Semiconductors.

ITTURRIET, F.; NAZAR, G.; FERREIRA, R.; MOREIRA, A.; CARRO, L. Adaptive parallelism exploitation under physical and real-time constraints for resilient systems. In: INTERNATIONAL WORKSHOP ON RECONFIGURABLE COMMUNICATION-CENTRIC SYSTEMS-ON-CHIP, 7., 2012, York, UK. **Proceedings. . .** Washington, DC, USA: IEEE Computer Society, 2012. p.1–8. (ReCoSoC '12).

JENN, E.; ARLAT, J.; RIMEN, M.; OHLSSON, J.; KARLSSON, J. Fault injection into VHDL models: the MEFISTO tool. In: INTERNATIONAL SYMPOSIUM ON

FAULT-TOLERANT COMPUTING, 24., 1994, Austin, TX, USA. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 1994. p.66–75. (FCTS-24).

JONES, T. M.; O'BOYLE, M. F. P. Evaluating the Effects of Compiler Optimisations on AVF. In: WORKSHOP ON INTERACTION BETWEEN COMPILERS AND COMPUTER ARCHITECTURE, 2008, Salt Lake City, UT, USA. **Proceedings...** Pittsburgh, PA, USA: University of Pittsburgh, 2008. p.1–6. (INTERACT-12).

KANAWATI, G.; NAIR, V.; KRISHNAMURTHY, N.; ABRAHAM, J. Evaluation of integrated system-level checks for on-line error detection. In: IEEE INTERNATIONAL COMPUTER PERFORMANCE AND DEPENDABILITY SYMPOSIUM, 1996, Urbana-Champaign, IL, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1996. p.292–301. (IPDS '96).

KEANE, J.; KIM, C. H. Transistor Aging. **IEEE Spectrum**, Washington, DC, USA, Apr 2011.

KEYS, A. et al. High-Performance, Radiation-Hardened Electronics for Space and Lunar Environments. **AIP Conference Proceedings**, College Park, MD, USA, v.969, n.749, p.749–756, 2008.

KIM, H.; SHIN, K. G. Design and Analysis of an Optimal Instruction-Retry Policy for TMR Controller Computers. **IEEE Transactions on Computers**, Washington, DC, USA, v.45, n.11, p.1217–1225, Nov. 1996.

KRANENBURG, T.; LEUKEN, R. van. MB-LITE: a robust, light-weight soft-core implementation of the MicroBlaze architecture. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 2010, Dresden, Germany. **Proceedings...** Leuven, Belgium: European Design and Automation Association, 2010. p.997–1000. (DATE '10).

LATTNER, C.; ADVE, V. LLVM: a compilation framework for lifelong program analysis & transformation. In: CODE GENERATION AND OPTIMIZATION, 2004, Palo Alto, CA. **Proceedings...** Los Alamitos, CA: IEEE, 2004. p.75–86. (CGO '04).

LI, M.-L.; RAMACHANDRAN, P.; SAHOO, S. K.; ADVE, S. V.; ADVE, V. S.; ZHOU, Y. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 13., 2008, Seattle, WA, USA. **Proceedings...** New York, NY, USA: ACM, 2008. p.265–276. (ASPLOS XIII).

LI, S.; AHN, J. H.; STRONG, R. D.; BROCKMAN, J. B.; TULLSEN, D. M.; JOUPPI, N. P. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: ND ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 42., 2009, New York, NY, USA. **Proceedings...** New York, NY, USA: ACM, 2009. p.469–480. (MICRO 42).

LIPASTI, M. H.; WILKERSON, C. B.; SHEN, J. P. Value Locality and Load Value Prediction. In: SEVENTH INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 1996, Cambridge, Massachusetts, USA. **Proceedings...** New York, NY, USA: ACM, 1996. p.138–147. (ASPLOS VII).

MAKAROV, V. N. Fighting register pressure in GCC. In: GCC DEVELOPERS' SUMMIT, 2004., 2004, Ottawa, Canada. **Proceedings...** Raleigh, NC, USA: Fedora Project, 2004. p.85–104. (GCC '04).

MARWEDEL, P. **Embedded System Design**: embedded systems foundations of cyber-physical systems. 1st.ed. New York, NY, USA: Springer, 2011.

MOORE, S. K. An Odometer for Silicon Chips. **IEEE Spectrum**, Washington, DC, USA, Jun 2009.

MORGAN, K. et al. A Comparison of TMR With Alternative Fault-Tolerant Design Techniques for FPGAs. **IEEE Transactions on Nuclear Science**, Washington, DC, USA, v.54, n.6, p.2065–2072, Dec 2007.

MUKHERJEE, S. S.; KONTZ, M.; REINHARDT, S. K. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 29., 2002, Anchorage, AK, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2002. p.99–110. (ISCA '02).

MUKHERJEE, S. S.; WEAVER, C.; EMER, J.; REINHARDT, S. K.; AUSTIN, T. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In: ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 36., 2003. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2003. p.29–41. (MICRO 36).

MURALIMANOHAR, N.; BALASUBRAMONIAN, R.; JOUPPI, N. P. **CACTI 6.0**: a tool to model large caches. 2009. Technical Report — HP Labs.

NAKKA, N.; PATTABIRAMAN, K.; IYER, R. Processor-Level Selective Replication. In: ANNUAL IEEE/IFIP INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 37., 2007, Edinburgh, UK. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2007. p.544–553. (DSN '07).

NASA. Next Generation Space Processor – Solicitation number BAA-RVKV-2013-02. 2013.

OH, N.; MITRA, S.; MCCLUSKEY, E. J. ED4I: error detection by diverse data and duplicated instructions. **IEEE Transactions on Computers**, Washington, DC, USA, v.51, n.2, p.180–199, Feb. 2002.

OH, N.; SHIRVANI, P.; MCCLUSKEY, E. Control-flow checking by software signatures. **IEEE Transactions on Reliability**, Washington, DC, USA, v.51, n.1, p.111–122, Mar 2002.

PAN, Z.; EIGENMANN, R. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In: INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION, 4., 2006, New York, NY, USA. **Proceedings. . .** Washington, DC, USA: IEEE Computer Society, 2006. p.319–332. (CGO '06).

PARIZI, R. B.; FERREIRA, R. R.; CARRO, L.; MOREIRA, A. F. Compiler Optimizations Do Impact the Reliability of Control-Flow Radiation Hardened Embedded Software. In: SCHIRNER, G., GöTZ, M., RETTBERG, A., ZANELLA, M., RAMMIG, F. (Ed.). **Embedded Systems**: design, analysis and verification. Berlin, Germany: Springer Berlin Heidelberg, 2013. p.49–60. (IFIP Advances in Information and Communication Technology, v.403).

PARRA, L.; LINDOSO, A.; PORTELA, M.; ENTRENA, L.; RESTREPO-CALLE, F.; CUENCA-ASENSI, S.; MARTINEZ-ALVAREZ, A. Efficient Mitigation of Data and Control Flow Errors in Microprocessors. **IEEE Transactions on Nuclear Science**, Washington, DC, USA, v.PP, n.99, p.1–7, 2014.

PATTABIRAMAN, K.; KALBARCZYK, Z.; IYER, R. Automated Derivation of Application-aware Error Detectors using Static Analysis. In: IEEE INTERNATIONAL ON-LINE TESTING SYMPOSIUM, 13., 2007, Crete, Greece. **Proceedings. . .** Washington, DC, USA: IEEE Computer Society, 2007. p.211–216. (IOLTS '07).

PENIX, J.; MEHLITZ, P. **Expecting the Unexpected**: radiation hardened software. 2005. Technical Report — NASA Ames.

PETERSEN, E. **Single Event Effects in Aerospace**. 1st.ed. Piscataway, NJ, USA: Wiley-IEEE Press, 2011.

PRVULOVIC, M.; ZHANG, Z.; TORRELLAS, J. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 29., 2002, Anchorage, AK, USA. **Proceedings. . .** Washington, DC, USA: IEEE Computer Society, 2002. p.111–122. (ISCA '02).

PSARAKIS, M.; GIZOPOULOS, D.; SANCHEZ, E.; REORDA, M. Microprocessor Software-Based Self-Testing. **IEEE Design & Test of Computers**, Washington, DC, USA, v.27, n.3, p.4–19, May 2010.

PYTLIK, B.; RENIERIS, M.; KRISHNAMURTHI, S.; REISS, S. P. Automated fault localization using potential invariants. In: FIFTH INTERNATIONAL WORKSHOP ON AUTOMATED AND ALGORITHMIC DEBUGGING, 2003, Ghent, Belgium. **Proceedings. . .** Ithaca, NY, USA: arXiv CoRR – Cornell University, 2003. p.273–276. (AADEBUG '03).

RAJENDIRAN, A.; ANANTHANARAYANAN, S.; PATEL, H. D.; TRIPUNITARA, M. V.; GARG, S. Reliable Computing with Ultra-reduced Instruction Set Co-processors. In: ANNUAL DESIGN AUTOMATION CONFERENCE, 49., 2012, San Francisco, CA, USA. **Proceedings. . .** New York, NY, USA: ACM, 2012. p.697–702. (DAC '12).

RANGAN, K. K.; WEI, G.-Y.; BROOKS, D. Thread Motion: fine-grained power management for multi-core systems. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 36., 2009, Austin, TX, USA. **Proceedings...** New York, NY, USA: ACM, 2009. p.302–313. (ISCA '09).

REBAUDENGO, M.; REORDA, M.; TORCHIANO, M.; VIOLANTE, M. Soft-error detection through software fault-tolerance techniques. In: INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS, 1999, Albuquerque, NM, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1999. p.210–218. (DFT '99).

REBAUDENGO, M.; REORDA, M.; VIOLANTE, M. A new software-based technique for low-cost fault-tolerant application. In: ANNUAL RELIABILITY AND MAINTAINABILITY SYMPOSIUM, 2003, Tampa, FL, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2003. p.25–28. (RAMS '03).

REED, R. et al. Heavy ion and proton-induced single event multiple upset. **IEEE Transactions on Nuclear Science**, Washington, DC, USA, v.44, n.6, p.2224–2229, Dec 1997.

REHMAN, S.; SHAFIQUE, M.; KRIEBEL, F.; HENKEL, J. Reliable software for unreliable hardware: embedded code generation aiming at reliability. In: INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, 9., 2011, Taipei, China. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2011. p.237–246. (CODES+ISSS '11).

REINHARDT, S. K.; MUKHERJEE, S. S. Transient Fault Detection via Simultaneous Multithreading. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 27., 2000, Vancouver, British Columbia, Canada. **Proceedings...** New York, NY, USA: ACM, 2000. p.25–36. (ISCA '00).

REIS, G. A.; CHANG, J.; VACHHARAJANI, N.; RANGAN, R.; AUGUST, D. I. SWIFT: software implemented fault tolerance. In: INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION, 2005, San Jose, CA, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2005. p.243–254. (CGO '05).

ROSEN, B. K.; WEGMAN, M. N.; ZADECK, F. K. Global Value Numbers and Redundant Computations. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 15., 1988, San Diego, California, USA. **Proceedings...** New York, NY, USA: ACM, 1988. p.12–27. (POPL '88).

ROUF, M. A.; KIM, S. Modeling and Evaluation of Control Flow Vulnerability in the Embedded System. In: IEEE INTERNATIONAL SYMPOSIUM ON MODELING, ANALYSIS AND SIMULATION OF COMPUTER AND TELECOMMUNICATION SYSTEMS, 2010., 2010, Miami Beach, FL, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2010. p.430–433. (MASCOTS '10).

SANTINI, T.; RECH, P.; NAZAR, G.; CARRO, L.; RECH WAGNER, F. Reducing embedded software radiation-induced failures through cache memories. In:

IEEE EUROPEAN TEST SYMPOSIUM, 19., 2014, Paderborn, Germany. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2014. p.1–6. (ETS '14).

SARTORI, J.; SLOAN, J.; KUMAR, R. Stochastic Computing: embracing errors in architectureand design of processors and applications. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURES AND SYNTHESIS FOR EMBEDDED SYSTEMS, 14., 2011, Taipei, Taiwan. **Proceedings...** New York, NY, USA: ACM, 2011. p.135–144. (CASES '11).

SASTRY HARI, S. K.; LI, M.-L.; RAMACHANDRAN, P.; CHOI, B.; ADVE, S. V. mSWAT: low-cost hardware fault detection and diagnosis for multicore systems. In: ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 42., 2009, New York, New York. **Proceedings...** New York, NY, USA: ACM, 2009. p.122–132. (MICRO 42).

SAXENA, N.; MCCLUSKEY, E. Control-flow checking using watchdog assists and extended-precision checksums. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 19., 1989, Chicago, IL, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1989. p.428–435. (FTCS 19).

SHARANGPANI, H.; ARORA, K. Itanium Processor Microarchitecture. **IEEE Micro**, Los Alamitos, CA, USA, v.20, n.5, p.24–43, Sept. 2000.

SLOAN, J.; SARTORI, J.; KUMAR, R. On software design for stochastic processors. In: ANNUAL DESIGN AUTOMATION CONFERENCE, 49., 2012, San Francisco, CA, USA. **Proceedings...** New York, NY, USA: ACM, 2012. p.918–923. (DAC '12).

THIEBAUT, D.; STONE, H. S. Footprints in the Cache. **ACM Trans. Comput. Syst.**, New York, NY, USA, v.5, n.4, p.305–329, Oct. 1987.

VEMU, R.; ABRAHAM, J. CEDA: control-flow error detection through assertions. In: IEEE INTERNATIONAL ON-LINE TESTING SYMPOSIUM, 12., 2006, Lake Como, Italy. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2006. p.1–6. (IOLTS '06).

VEMU, R.; GURUMURTHY, S.; ABRAHAM, J. ACCE: automatic correction of control-flow errors. In: IEEE INTERNATIONAL TEST CONFERENCE, 2007, Santa Clara, CA, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2007. p.1–10. (ITC '07).

VERA, X.; ABELLA, J.; CARRETERO, J.; GONZáLEZ, A. Selective Replication: a lightweight technique for soft errors. **ACM Transactions on Computing Systems**, New York, NY, USA, v.27, n.4, p.8:1–8:30, Jan. 2010.

WANG, N.; PATEL, S. ReStore: symptom based soft error detection in microprocessors. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 2005., 2005, Yokohama, Japan. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2005. p.30–39. (DSN '05).

WILHELM, R.; GRUND, D. Computation Takes Time, but How Much? **Communications of the ACM**, New York, NY, USA, v.57, n.2, p.94–103, Feb. 2014.

XU, J.; TAN, Q.; TAN, L.; ZHOU, H. An Instruction-level Fine-grained Recovery Approach for Soft Errors. In: ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, 28., 2013, Coimbra, Portugal. **Proceedings...** New York, NY, USA: ACM, 2013. p.1511–1516. (SAC '13).

YALCIN, G.; UNSAL, O.; CRISTAL, A. FaulTM: error detection and recovery using hardware transactional memory. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 2013, Grenoble, France. **Proceedings...** San Jose, CA, USA: EDA Consortium, 2013. p.220–225. (DATE '13).

YAN, J.; ZHANG, W. Compiler-guided Register Reliability Improvement Against Soft Errors. In: ACM INTERNATIONAL CONFERENCE ON EMBEDDED SOFTWARE, 5., 2005, Jersey City, NJ, USA. **Proceedings...** New York, NY, USA: ACM, 2005. p.203–209. (EMSOFT '05).

YETIM, Y.; MARTONOSI, M.; MALIK, S. Extracting Useful Computation from Error-prone Processors for Streaming Applications. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 2013, Grenoble, France. **Proceedings...** San Jose, CA, USA: EDA Consortium, 2013. p.202–207. (DATE '13).

APPENDIX A RESUMO EM PORTUGUÊS

A.1 Introdução

A implementação de tolerância a falhas em sistemas embarcados é um desafio devido à restrições físicas de área, potência e desempenho que esses sistemas devem respeitar, além de cobertura de falhas. A solução típica para tolerância a falhas em sistemas de computação em geral é o emprego de redundância modular tripla (TMR), no caso de sistemas tolerante a falhas capazes de recuperar o estado do mesmo após a ocorrência de um erro. O problema dessa solução é o custo de potência e área associados, o que torna proibitivo o emprego de TMR em sistemas embarcados.

Este trabalho propõe uma nova abordagem para tolerância a falhas em sistemas embarcados a qual provê cobertura de falhas equivalente à TMR, mas com custos físicos comparáveis à redundância modular dupla (DMR). A pilha de HW/SW transacional (ou simplesmente, Pilha) introduz uma nova abordagem de compilação de programas onde os blocos básicos originalmente existentes são transformados em Blocos Básicos Transacionais (TBB). Um TBB é uma unidade atômica de contenção de erros, a qual não permite que erros de lógica de hardware se propaguem para a memória principal. Em caso de erros, o TBB pode simplesmente ser re-executado, sem preocurpar-se em salvar e restaurar o estado arquitetural de hardware. Em nível de hardware, esta tese introduz a Arquitetura de Blocos Básicos Transacionais (ToBBA), um novo processador tolerante à falhas capaz de executar a semântica do TBB, provendo correção de erros sem o uso de técnicas de checkpointing arquitetural.

Este breve anexo apresenta de maneira sucinta em Português os principais resultados experimentais obtidos nesta tese, apresentando de maneira quantitativa a técnica proposta. A Seção A.2 discute a metodologia experimental usada para obter os resultados experimentais. A Seção A.3 apresenta e discute os resultados experimentais da Pilha.

A.2 Metodologia

Cobertura de falhas. A campanha de injeção de falhas foi realizada com a descrição VHDL da arquitetura de hardware. Esse VHDL foi executado em um FPGA Xilinx Virtex-5, onde para cada ciclo de cada programa uma falha foi injetada em um sinal da netlist pós-síntese com o uso de um módulo *sabotador* (JENN ET AL., 1994). O injetor de falhas usado nesta tese modifica a netlist, instrumentando-a de maneira a permitir que o valor de cada sinal seja alterado pelo módulo sabotador em qualquer momento da simulação de maneira síncrona ao ciclo de clock. Após o programa terminar sua execução, o injetor de falhas compara a memória dessa execução com uma memória de referência gold. Caso ocorra alguma diferença entre os valores dessas duas memórias, a falha injetada se tornou um erro. Do contrário, a falha foi mascarada pela arquitetura ou o erro foi corrigido. O modelo de falhas usado neste trabalho é o soft-error, onde somente um bit é escolhido aleatoriamente e seu valor é alterado quando a falha é injetada. O bit-flip tem sua duração forçada a um ciclo. Esse modelo de falhas modela Single Event Upsets (SEU) (PETERSEN, 2011, C. 2, P. 14). As falhas foram também injetadas no circuito de recuperação de falhas e seus componentes internos, logo o cenário no qual as falhas ocorrem no mecanismo de tolerância a falhas foi considerado. Foram usados como benchmark seis pequenos programas: bubble sort (bbsort), mñimos quadrados (lsquares), CRC32, computação da árvore espalhadora mínima (kruskal), algoritmo de todos os caminhos entre todas as cidades (floyd) e multiplicação de matrizes (matmul). A campanha de injeção de falhas para esse benchmark é composta de **672.348.891** falhas injetadas. Os algoritmos e os dados de entrada usados nos experimentos de cobertura de falhas são apresentados na Tabela 5.1.

Injeção de falhas permanentes. O VHDL do processador ToBBA foi sintetizado em nível de portas lógicas usando o Synopsys Design Compiler e posteriormente exportado para Verilog. Esse Verilog sintetizado foi usado na ferramenta Synopsis TetraMAX e teve falhas stuck-at injetadas usando as formas de ondas geradas com o ModelSim. A forma de onda foi criada a partir da execução completa do programa bubble sort para a ordernação de 10 elementos inteiros em um vetor. No total, **121.708** falhas permanentes foram injetadas a partir do início da simulação, as quais remanesceram ativas até o final da execução do programa. A injeção de falhas stuck-at compreendeu todas as portas de entrada e saída de todas as portas lógicas do circuito, tanto para stuck-at-0 quanto para stuck-at-1.

Ocupação de área e dissipação de potência. Os resultados de área e potência obtidos para o ToBBA foram extraídos com o programa Cadence RTL Compiler usando uma biblioteca de transitores de 65 nm a partir da descrição VHDL da arquitetura ToBBA. Os resultados de área e potência para o banco de registradores foram obtidos com a ferramenta CACTI 6.5 (Muralimanohar, Balasubramonian and Jouppi, 2009). Para os resultados de área e potência, a arquitetura ToBBA foi sintetizada em 300 MHz. Como arquitetura baseline para calcular o overhead de área e potência, adotou-se a arquitetura MicroBlaze de núcleo único também em 300 MHz com seu banco de registradores original de 32 registradores.

Overhead de desempenho. Os resultados de desempenho neste trabalho foram obtidos de duas maneiras. Durante a campanha de injeção de falhas, o módulo sabotador ao injetar uma falha também capturava a instrução em que a falha foi injetada. Com essa informação, calculou-se a latência média de recuperação de erro a partir do modelo RTL da arquitetura ToBBA tal qual apresentado na Seção 5.4.1. O overhead de desempenho em tempo de execução foi mensurado com a ferramenta gem5 (Binkert et al., 2011) para uma arquitetura in-order usando o conjunto de instruções ARM v7-a. Foi usado o seguinte sub-conjunto de programas do benchmark MiBench (Guthaus et al., 2001): basicmath, bitcount, qsort, susan (corners, edges, smoothing), dijkstra, patricia, crc32, fft, sha, rijndael e adpcm. Esses programas foram selecionados por possuírem disponível seus códigos-fonte C completos, não sendo dependentes de arquivos binários pré-compilados. As versões originais e em TBB dos programas foram executadas para os conjuntos de dados pequeno (small) e grande (large) para ser possível avaliar a escalabilidade da técnica proposta. A Tabela 5.2 lista a configuração utilizada para realizar as simulações no gem5. O LLVM 3.4 foi estendido para incluir o algoritmo de geração de programas TBB apresentado no Capítulo 2 e no Capítulo 4 para o Spill Register File. Os programas foram compilados no LLVM usando a opção de otimização de código O3 para as versões baseline de comparação. Os programas usando TBB foram também compilados com a opção O3, mas desabilitando as otimizações loop-invariant code motion (LICM) e unconditional branch removal.

Consumo de energia. O programa McPAT (Li et al., 2009) foi alimentado com as estatísticas de execução obtidas com o gem5 para as versões orginais e com TBB dos programas com o objetivo de extrair o consumo energético desses programas. A arquitetura simulada tal como descrita na Tabela 5.2 foi modelada no McPAT, o qual invoca o programa CACTI (Muralimanohar, Balasubramonian and Jouppi, 2009) para modelar o consumo energético das memórias da arquitetura.

A.3 Resumo dos Resultados Experimentais

A.3.1 Cobertura de Falhas

Os resultados da campanha de injeção de falhas do ToBBA em termos de cobertura de correção e detecção de SEU são apresentados na Tabela 5.3. Para o benchmark completo, a cobertura de erros do ToBBA, i.e., a soma das coberturas de detecção e correção, é de 99,9%. Considerando somente os erros corrigidos, a cobertura de correção média do ToBBA é 99,3%.

Dado que a campanha de injeção de falhas considerou falhas em *todos* os componentes arquiteturais, houve uma pequena quantidade de erros que não foram detectados, i.e., observaram-se Silent Data Corruption (SDC). Esses SDC's são devidos à falhas injetadas no endereç dos dados que serão escritos na memória logo após esses foram comparados dentro do ToBBA pelos comparadores e logo antes a ocorrência da escrita da memória. Para reduzir a probabilidade de um SDC, poderia-se adicionar mais comparadores até que a probabilidade de um SDC seja aceitável para um dado projeto, pagando os custos adicionais em área, potência e latência. Uma outra solução seria o ToBBA codificar todos os dados a serem escritos em memória com ECC, permitindo que o controlador da memória verifique a correção dos dados a serem escritos. Os erros SDC compreendem **menos que 0.1%** de toda a campanha de injeção de falhas.

Os resultados apresentados nesta seção são interessantes pois até o momento não há na litetura uma metodologia unificada que reduz o consumo de potência e ocupação de área em relação à TMR com a mesma cobertura de falhas que TMR. TMR é bastante difícil de se vencer em termos de desempenho pois o seu overhead é negligenciável e escala linearmente com a complexidade do circuito sendo protegido (Hentschke et al., 2002). Geralmente, para se lidar com a alta ocupação de área e consumo de potência que TMR incorre, as soluções para tolerância a falhas relaxam os requisitos de desempenho ou de coberbura, tal como discutido no Capítulo 6.

A.3.2 Caracterização de Área e Potência

Fig. 5.1 apresenta a ocupação relativa de área de cada unidade arquitetural do ToBBA, evidenciando que o circuito de recuperação de erros compreende uma parte negligenciável da área total. A variação observada da ocupação para frequências distintas deve-se ao trabalho adicional que a ferramenta de síntese faz para se atingir frequências mais altas, apesar de todas as versões serem funcionalmente equivalentes.

Os overheads de ocupação de área e dissipação de potência são apresentados na Figura 5.2. Nessa figura, o baseline é um single-core com um único banco de registradores sem ECC. Na avaliação de área e potência, cinco cenários foram criados para permitir uma melhor comparação com o ToBBA: (1) single-core com ECC (1Core-ECC); (2) DMR sem ECC (DMR); (3) DMR com ECC (DMR-ECC); (4) TMR sem ECC (TMR); (5) TMR com ECC (TMR-ECC). Nessa figura, os resultados do ToBBA são apresentados nas barras verdes próximas à 'TMR-ECC' para as versões com (ToBBA-SRF) e sem (ToBBA) o SRF. O leitor pode referir ao Capítulo 1 para uma discussão sobre as implicações de usar ou não ECC em sistemas DMR e TMR.

Os resultados de área e potência são baseados em duas observações: (1) um banco de registradores protegido com ECC ocupa de maneira grosseira três vezes mais área que um banco sem ECC (Blome et al., 2006); (2) o SRF não necessita ser protegido com ECC. A hipótese (1) é um fato conhecido. A justificativa de (2) é que, caso exista um erro no SRF, a execução do programa está correta, evitando que os dados errôneos do SRF sejam utilizados. Caso exista um erro que corrompa a execução do programa, os dados no SRF estão corretos e podem ser usados para recuperar o TBB. No ToBBA, o banco de registradores principal precisa ser protegido com ECC, pois um erro capturado nele não pode propagar para o SRF.

Em área, o 'ToBBA-SRF' apresenta um overhead de 2,65 em relação ao single-core. 'ToBBA' apresenta overhead de 2,35, ainda melhor que 'DMR-ECC' e 'TMR' padrão. Note que 'DMR-ECC' não provê correção de erros. O uso do SRF ocupa 18% menos área que 'TMR', a primeira configuração capaz de prover correção de erros. 'ToBBA-SRF' aumenta em 12% o overhead de área em comparação ao 'ToBBA'.

Em potência, 'ToBBA-SRF' apresenta overhead de 2,05 em relação ao single-core. Já 'ToBBA' apresenta overhead de 2,02 sendo comparável com 'DMR' e 'DMR-ECC'. Note que para dissipação de potência, ToBBA provê correção de erros tal como 'TMR' com os memsos custos de somente detecção de erros da mesma maneira que 'DMR' e 'DMR-ECC' provêem. 'ToBBA-SRF' aumenta em 2% o overhead de potência em comparação ao 'ToBBA'.

A.3.3 Análise de Desempenho e de Latência de Recuperação de Erros

A.3.3.1 Latência de Recuperação de Erros

Fig. 5.3 apresenta o pior caso e a média da latência de recuperação de erros com desvio padrão em número de ciclos executados para que o circuito de recuperação de erros restaure o TBB. Como discutido anteriormente, a latência de pior caso é sempre o tamanho do TBB. A latência média mensurada foi de 6,17 ciclos com 2,09 de desvio padrão. Esses resultados mostram que, na média, a latência de recuperação é menor que $1/3$ do número total de instruções de um TBB.

A reduzida latência média de recuperação de erros e a computação determinística do pior caso é uma contribuição chave da Pilha Transacional. Recuperação de erros é uma tarefa cara em sistemas tolerantes a falhas, a qual incorre em overhead de 25% a quase 100% da aplicação, tal como foi discutido no Capítulo 1. Esse overhead depende de quantas instruções devem ser re-executadas para se recuperar o erro. Na Pilha, esse overhead é reduzido ao menor possível.

A.3.3.2 Overhead de Desempenho Sem o Spill Register File

As instruções adicionais de spill criadas para eliminar a comunicação entre registradores compartilhados entre os TBB incorrem em overhead de desempenho. Figure 5.4 apresenta o overhead de desempenho mensurado para o subconjunto de programas do MiBench quando o programa original é transformado em uma versão equivalente com TBBs ao invés de blocos básicos tradicionais.

Na Figura 5.4, cinco programas apresentam overhead de desempenho superiores a 2: 'susan corners', 'susan smoothing', 'dijkstra', 'sha' e 'adpcm'. A média geométrica do overhead de desempenho mensurada foi 1,54 ('geomean' na figura). Esse resultado é interessante, pois as instruções adicionais de spill são load e stores que potencialmente poderiam degradar de maneira considerável o desempenho. Como discutido no Capítulo 6, esse overhead de desempenho já é um avanço considerável sobre o estado da arte em técnicas de correção de erros.

A.3.3.3 Overhead de Desempenho Com o Spill Register File

Figura 5.5 apresenta o overhead de desempenho do ToBBA-SRF para o sunconjunto selecionado do MiBench e a Figura 5.6 mostra o ganho de desempenho do ToBBA-SRF com o uso do SRF ao invés das instruções de spill. A média geométrica do overhead de desempenho diminui de 1,54 para 1,33 devido ao SRF e a remoção das instruções adicionais de spill. Essa queda no overhead de desempenho cria uma melhora de 13,81% no overhead de desempenho.

Analizando a Figura 4.1, pode-se observar que o acréscimo no footprint da cache incorre em overhead de desempenho devido à alocação de registradores. Com a análise da melhora de desempenho para o ToBBA-SRF, pode-se observar que os programas com as melhoras mais significativas em desempenho correspondem de maneira grosseira com os programas que apresentam os menores crescimentos no footprint da cache balanceado. Por exemplo, 'rijndael' é o programa com o maior cache footprint e também é o programa que apresentou a pior melhora em performance (-4,64%). A relação melhora em desempenho e cache footprint não é linear, como pode-se observar no 'susan corners' (47,56% de melhora). Susan corners é uma aplicação majoritariamente orientada a dados, mas que possui uma profunda dependência de controle na sua variável n pertencente ao laço for mais externo.

O registrador que armazena n é salvo frequentemente sem o SRF. Ao se remover esses spills, o desempenho aumenta signficativamente como pode-se observar na Figura 5.6.

Programas pouco sensíveis à alocação de registradores incorrem em baixo overhead de desempenho. Por exemplo, 'susan corners', 'qsort' e 'patricia' possuem overhead de desempenho de 1,03, 1,05 e 1,01, respectivamente. Considerando os programas com overhead próximos à média, e.g., 'bitcount' (1,20) e 'rijndael' (1,25), o overhead existente ainda é menor que as técnicas existentes publicadas na literatura para correção de erros. Diferentemente de todas as técnicas publicadas na literatura, dois programas exibem **aumento de desempenho**: 'basicmath' (0,97) e 'fft' (0,92). Para esses dois programas, a nova alocação de registradores favoreceram um melhor uso de registradores, incorrendo em um pequeno mas desejável ganho de desempenho.

Apesar da remoç das instruções spill reduzirem o overhead em ToBBA-SRF, há um overhead inerente à modificação na estrutura do programa. Figura 5.7 apresenta o overhead de desempenho cuulativo de cada modificação realizada no fluxo de compilação do LLVM para dois programas que exibem overhead considerável ('dijkstra' e 'adpcm'). O passo 'Post RA Code Motion' não é relatado pois ele não incorre em overhead de desempenho para esses dois programas. Muito do overhead de desempenho decorre do novo escalonamento introduzido pelo TBB devido às modificações impostas à alocação de registradores. A remoção de saltos incondicionais, (remoção de fallthrough na figura) incorre em pesado overhead para o 'adpcm', pois esse programa possui diversas iterações de laço com um grande número de construções de fluxo de controle.

A.3.4 Consumo de Energia

A figura 5.8 apresenta o overhead de consumo de energia do TBB considerando um Banco de Registradores Principal (WRF) completamente duplicado. Esses resultados superestimam o consumo real de energia, pois o SRF possui uma quantidade consideravelmente menor de acessos que o WRF. Infelizmente, não foi possv́el modelar o SRF no McPAT. Entretanto, esses resultados provêem uma idéia como o TBB se comportaria em termos de overhead de consumo energético. A média geométrica mensurada é 2,48, ainda menor que um arranjo TMR o qual incorreria em overhead de pelo menos 3.

O overhead mais alto de energia observado foi o ‘adpcm’ com 5,30. O menor foi para o ‘basicmath’ com 2,003. Note que ‘basicmath’ foi um dos programas que apresentou ganho de desempenho com a introdução do SRF. Portanto, pode-se esperar que o consumo real de energia também seria menor que 2, incorrendo em redução no consumo energético. Esse também foi o caso com ‘fft’, o qual possui overhead de 1,65. Esse resultado é melhor que um arranjo DMR padrão.

O overhead de consumo energético é uma métrica difícil de se comparar com outras publicadas na literatura pois nenhuma das técnicas publicadas até então relatam o consumo de energia. Entretanto, pode-se esperar que essas outras técnicas incorreriam em overhead mais alto de energia pois elas possuem overhead de desempenho superior a este trabalho.

A.3.5 Comportamento do Circuito de Recuperação sob Stuck-At

A Tabela 5.4 apresenta o comportamento de falhas stuck-at-0 e stuck-at-1 injetadas em todos os sinais do circuito de recuperação de erros do ToBBA.

A injeção de falhas stuck-at no registrador ‘TBB Addr’ e no sinal ‘isRecovering’ não cria um erro observável. Isso se deve ao fato que esses valores são utilizados somente quando um erro transitório é detectado pelo circuito de recuperação de erros. Uma falha stuck-at-0 durante a ocorrência de um erro transitório não influencia o estado do core RISC. Entretanto, falhas stuck-at-1 injetadas nos sinals ‘forceReset’, ‘isFaultyStore’ e ‘faultDetected’ levam o ToBBA a um comportamento errado. Isso se deve a esses três sinais alterarem o comportamento do core RISC. Uma falha stuck-at-1 no sinal ‘isFaultyStore’ pode ser indiretamente detectada pela mudança no modo de operação do banco de registradores, de ‘write permission’ para ‘read-only’. Portanto, o circuito de recuperação de erros do ToBBA funciona da maneira esperada mesmo com a presença de falhas permanentes.