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

RAFAEL BILLIG TONETTO

A Platform to Evaluate the Fault Sensitivity of Superscalar Processors

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

Advisor: Prof. Dr. Antonio Carlos Schneider Beck
Coadvisor: Prof. Dr. Gabriel Nazar

Porto Alegre
October 2017

CIP — CATALOGING-IN-PUBLICATION

Billig Tonetto, Rafael

A Platform to Evaluate the Fault Sensitivity of Superscalar Processors / Rafael Billig Tonetto. – Porto Alegre: PPGC da UFRGS, 2017.

93 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2017. Advisor: Antonio Carlos Schneider Beck; Coadvisor: Gabriel Nazar.

1. Fault injection. 2. Superscalar processor. 3. Register-transfer level. I. Beck, Antonio Carlos Schneider. II. Nazar, Gabriel. III. A Platform to Evaluate the Fault Sensitivity of Superscalar Processors.

ABSTRACT

The aggressive shrinking of transistors, which led to the reductions in the operating voltage, has been providing enormous benefits in terms of computational power while keeping the energy consumption at an acceptable level. However, as feature size and voltage decrease, the susceptibility to soft errors tends to increase, and the importance of fault evaluations grows. Superscalar processors, which nowadays dominate the market, are a significant example of systems that take advantage of these technological improvements and are more susceptible to errors. Along with that, there exist several methods for fault injection, which is an efficient means to evaluate the resiliency of such processors. However, traditional fault injection methods, such as the hardware-based technique, impose that the processor must be physically implemented before the tests can be conducted, while not providing reasonable levels of controllability. On the other hand, techniques based on simulators implemented in Software offer high levels of controllability. However, while high-level SW simulators (which are fast) may lead to an incomplete, or even misguided, evaluation of the system's resiliency since they don't model the hardware internals (such as the pipeline registers), low-level SW simulators are extremely slow and are hardly available at RTL (Register-Transfer Level). Considering this scenario, we propose a platform that bridges the gap between the HW and SW approaches to evaluate faults in superscalar processors: it is fast, with high controllability, available in software, flexible, and, most importantly, it models the processor at RTL. The tool was implemented on top of the framework used to generate the Berkeley Out-of-Order Machine (BOOM) superscalar processor, which is a highly scalable and parameterizable processor. This property allowed us to experiment with three different architectures of the processor: single-, dual-, and quad-issue out-of-order cores, and, by analyzing how the resiliency to faults is influenced by the complexity of different processors, use them to validate our tool.

Keywords: Fault injection. superscalar processor. register-transfer level.

RESUMO

A diminuição agressiva dos transistores, a qual levou a reduções na tensão de operação, vem proporcionando enormes benefícios em termos de poder computacional, mantendo o consumo de energia em um nível aceitável. No entanto, à medida que o tamanho dos recursos e a tensão diminuem, a susceptibilidade a falhas tende a aumentar e a importância das avaliações com falhas cresce. Os processadores superescalares, que hoje dominam o mercado, são um exemplo significativo de sistemas que se beneficiam destas melhorias tecnológicas e são mais suscetíveis a erros. Juntamente com isso, existem vários métodos para injeção de falhas, que é um meio eficiente para avaliar a resiliência desses processadores. No entanto, os métodos tradicionais de injeção de falhas, como a técnica baseada em hardware, impõem que o processador seja implementado fisicamente antes que os testes possam ser conduzidos, sem fornecer níveis razoáveis de controlabilidade. Por outro lado, as técnicas baseadas em simuladores implementados em software oferecem altos níveis de controlabilidade. No entanto, enquanto os simuladores em SW de alto nível (que são rápidos) podem levar a uma avaliação incompleta, ou mesmo equivocada, da resiliência do sistema, uma vez que não modelam os componentes internos do hardware (como os registradores do pipeline), simuladores em SW de baixo nível são extremamente lentos e dificilmente estão disponíveis em RTL (Register-Transfer Level). Considerando este cenário, propomos uma plataforma que *preenche a lacuna* entre as abordagens em HW e SW para avaliar falhas em processadores superescalares: é rápida, tem alta controlabilidade, disponível em software, flexível e, o mais importante, modela o processador em RTL. A ferramenta foi implementada sobre a plataforma usada para gerar o processador superescalar *The Berkeley Out-of-Order Machine* (BOOM), que é um processador altamente escalável e parametrizável. Esta propriedade nos permitiu experimentar três arquiteturas diferentes do processador: single-, dual- e quad-issue, e, ao analisar como a resiliência a falhas é influenciada pela complexidade de diferentes processadores, usamos os processadores para validar nossa ferramenta.

Palavras-chave: Injeção de falhas , processadores superscalares , nível de transferência de registradores.

LIST OF ABBREVIATIONS AND ACRONYMS

ACE	Architecturally Correct Execution
ALU	Arithmetic Logic Unit
API	Application Programming Interface
AVF	Architectural Vulnerability Factor
BHT	Branch History Table
BPD	Backing Predictor
BPU	Branch Prediction Unit
BTB	Branch Target Buffer
CHISEL	Constructing Hardware in an Scala Embedded Language
CMOS	Complementary Metal-Oxide-Semiconductor
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random Access Memory
DUT	Design Under Test
EPC	Exception Program Counter
FIT	Failures in Time
FPU	Floating-Point Unit
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
ILP	Instruction Level Parallelism
IPC	Instructions per Cycle
ISA	Instruction Set Architecture
JVM	Java Virtual Machine
MBU	Multiple Bit Upset
RAS	Return Address Stack

RAW	Read After Write
RF	Register File
RISC	Reduced Instruction Set Computer
ROB	Reorder Buffer
RTL	Register-Transfer Level
SEE	Single Event Effect
SER	Soft Error Rate
SET	Single Event Transient
SEU	Single Event Upset
SIMT	Single Instruction/Multiple Threads
SOC	System on Chip
SRAM	Static Random Access Memory
TMR	Triple Modular Redundancy
VCS	Verilog Compiler Simulator
VHDL	VHSIC Hardware Description Language
WAR	Write After Read
WAW	Write After Write

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

It has been widely reported that the decrease in technology size is allowing for the implementation of progressively dense and complex processors that boost the computational power to a remarkable level. However, achieving even higher performance while keeping energy consumption at an acceptable level comes at a price, and there are trade-offs that should be taken into account.

Much of the improvement in computational performance is a consequence of the technology advances that nowadays allow the multiplication of resources in complex processors, such as the superscalar ones. Nowadays, superscalar processors dominate the market due to their efficiency. Therefore, reliability research should pay special attention to them.

The technology shrinking resulted in the improvements of superscalar processors because it allowed more transistors to be packed in the same area, resulting in more complex structures and in the multiplication of resources that makes it possible to execute more instructions per cycle. This achievement was mainly possible due to the increase of the density of transistors enabled by technology scaling, which is a consequence of the ameliorations in the complementary metal-oxide-semiconductor (CMOS) technology, as transistors tend to be smaller and faster (DODD et al., 2010).

As a consequence of the technology scaling, transistors must now operate at lower voltages since the amount of charge necessary to change a single bit in a state element (the critical charge) has decreased. More precisely, as transistors' shrink, the voltage has to be reduced to prevent them from wearing out or collapsing due to high electric fields because the critical charge decreases linearly with the voltage (KARNIK; HAZUCHA, 2004).

As it seems to be a *natural law*, it is impossible to achieve any sort of improvement while not paying the price for it. Most of the advances in computational power were enabled by the technology shrinking, which led to an increase in the rate of soft errors.

However, due to the growing demand for performance that can mainly be explored by means of instruction-level parallelism (ILP), superscalar processors are of increasing relevance for many of the everyday applications. As this demand for superscalar processors is accompanied by the increasing levels of sensitivity of the transistors, research should pay special attention regarding fault tolerance strategies for such processors. However, the design of efficient fault tolerance mechanisms demand that the system's behav-

ior should be first *evaluated* when it is subject to faults. By *evaluating*, it is meant that the processor should be artificially deployed in harsh conditions where the environment constantly injects faults into it, all the while its behavior is observed. This way, an understanding of the system's behavior when it encounters faults is provided. Therefore, in this work, we present a fault injection platform for the Berkeley Out-of-Order Machine (BOOM) superscalar processor.

1.1 Problem Statement and Context of this Work

One of the most common and efficient strategies adopted for the evaluation of a system's sensitivity to faults is fault injection, which consists in intentionally flipping the bits of the processor in a scale way bigger than it would experience in a real environment. However, often some of the fault injection techniques, such as hardware-based methods and methods based on high-level simulators have inherent deficiencies and trade-offs.

Even though the good level of representativity of the hardware-based fault injection method is attractive, it demands that the system is physically implemented before it can be evaluated, hence its levels of accessibility is not much encouraging, and it is also expensive because it requires special equipment to induce faults. Moreover, there is no easy way, if any, of controlling where and when faults should be injected into the hardware.

When the hardware-based method does not satisfy due to its shortcomings, one could take advantage of more abstract methods, such as the ones that use high-level simulators. In this scenario, the system is represented by means of a software-based simulator that abstracts several of the micro-architectural structures of the system. In many cases, such simulators are so high-level that they model only the functional aspect of the system. However, as the processor is modeled in software, the levels of controllability and accessibility are attractive. Note, however, that this strategy does not model the system at RTL, which may lead to loss of representativity when the reliability is evaluated.

Therefore, the importance of this dissertation is based on three main claims:

1. Superscalar processors are becoming even more important and ubiquitous.
2. Superscalar processors are becoming more sensitive to faults due to technology scaling.
3. There is a growing need for efficient tools to evaluate the resiliency of superscalar

processors.

Items (1) and (2) tell that the resiliency evaluation of superscalar processors is of growing importance. Item (3) is accompanied by the fact that there is a trade-off between hardware-based methods and strategies that use high-level simulators. For this reason, in this work, we proposed and implemented a platform for the evaluation of the susceptibility to soft errors of the BOOM superscalar processor.

The proposed platform trades off the deficiencies imposed by hardware- and simulation-based methods as it is based on an RTL software-implemented model of the processor, hence it inherits levels of representativeness close to the hardware-based method, and the levels of controllability of the techniques based on simulators.

We believe that fault evaluations of BOOM can provide significant understandings about the processor's behavior when it is deployed in harsh environments, mainly because the proposed tool works at RTL. Besides, the proposed platform is based on a highly flexible simulator generator that allows us to parameterize several features of the processor, such as the issue-width. Because it is easy to parameterize the BOOM processor, we evaluated three versions of it: single-, dual-, and quad-issue cores were experimented.

Because the proposed platform is based on a software-implemented simulator that works at RTL (thus every single bit of the processor is modeled), it should be no surprise that the fault injection process is not as fast as the methods based on high-level and functional simulators. Therefore, we implemented a checkpointing mechanism in order to speedup the fault injection campaigns; this trick yielded remarkable gains in terms of number of faults injected per second.

This dissertation is organized as follows:

- **Chapter 2:** Reviews the basics on superscalar processors and gives an overview, to a certain detail, of the BOOM's architecture, its ISA, and the Chisel language.
- **Chapter 3:** Gives a background on soft errors and presents some related fault injection tools.
- **Chapter 4:** Details how the proposed platform was implemented and how it works.
- **Chapter 5:** Evaluates the implemented tool by injecting faults in the BOOM processor.
- **Chapter 6:** Final conclusions and suggestions for future work.

2 BOOM - THE BERKELEY OUT-OF-ORDER MACHINE

2.1 Introduction

As demand for performance increases, the importance of superscalar processors also grows since they deliver significant gains in terms of instructions per cycle (IPC) at an acceptable cost.

Superscalar processors functionality, such as in (PALACHARLA; JOUPPI; SMITH, 1997), consists of issuing and executing multiple instructions per cycle. This organization is possible nowadays due to technology improvements that allow the multiplication of resources of the processor, such as multiple functional units. By benefiting from multiple functional units, speculative execution and instruction scheduling (static or dynamic), superscalar designs achieved such a significant performance that nowadays they lead the market of embedded systems. Basically, superscalar processors deliver better performance by exploring instruction-level parallelism (SMITH; SOHI, 1995).

2.1.1 The Basic Functionality of Superscalar Processors

Generally speaking, superscalar processors may be categorized into two types: in-order and out-of-order execution. For the in-order execution case, the instructions are statically scheduled (at compile time) and are executed in the same order the programmer would expect they should execute. In other words, instructions are executed in the same sequence as they are fetched from the instruction memory.

One shortcoming imposed by in-order execution is related to some aspects of the program flow that can bring the processor performance down. For example, when one instruction depends upon the results of a previous one and this hazard cannot be resolved by any means, then the pipeline has to stall until the first instruction completes. This usually happens when the results from the previous instruction arrive from load operations, which have a high latency when there is a cache miss. Stalling the processor implies that some structures of the processor will be kept idle, which in turn harms the processor performance. This problem can be attenuated by executing instructions out of the program order.

The out-of-order execution allows for the processor to dynamically schedule and execute decoded instructions out of the expected order. This technique can significantly

reduce the impact of some data dependencies, as happens with the in-order example aforementioned. In this scenario, when the next instruction depends upon the results of a previous load instruction, then instead of stalling the pipeline, the processor can execute another instruction, ahead of the next one, that has no dependencies yet to be resolved. In short, out-of-order processors increase performance because they are keen on: 1) finding more instructions that can execute in parallel (i.e., they explore ILP); and 2) tolerating/-masking instruction latencies, such as load operations. Simply put, out-of-order execution increases the processor's performance (ACOSTA; KJELSTRUP; TORNG, 1986).

Authors in (PALACHARLA; JOUPPI; SMITH, 1997) describe the basic functionality of the pipeline of a generic and abstract superscalar processor, which may be summarized as:

1. Multiple instructions are fetched every cycle from the instruction cache, and branch prediction is made for conditional branch instructions.
2. Instructions are decoded and their register operands are renamed to resolve false dependencies. Renamed instructions are dispatched to the instruction window, where they wait for their source operands and the appropriate functional unit, such as ALUs or FPUs, to become available.
3. As soon as the conditions in (2) are satisfied, instructions are issued and executed in the functional units. The operand values of an instruction are either fetched from the register file or are bypassed (through a bypass logic) from earlier instructions in the pipeline.

The next sections present the basic architecture and organization of the BOOM processor, which fits into the out-of-order category.

2.2 An Overview on The Berkeley Out-of-Order Machine

2.2.1 Introduction

The development of a fault injection platform demands, of course, a careful choice of the processor the fault injector is supposed to be built upon.

An initial alternative for such processor could be, for example, the OpenSPARC T1 processor (PARULKAR et al., 2008), which was developed by Sun Microsystems and released as an open-source Verilog project to the public back in 2006. Benefiting

from an RTL description of the processor sounds attractive, however, as this processor is developed as a multithreading/multicore CPU (it contains eight cores), it turned out to be too complicated for our purposes. Moreover, OpenSPARC T1 does not support out-of-order execution and has the disadvantage of not being parameterizable; one cannot just easily configure the processor in a desired way. As a consequence, the fault injection platform would be restricted only to one processor configuration.

As an alternative to OpenSPARC, one could resort to the gem5 simulator (BINKERT et al., 2011), for instance, which is a very popular and low-level open-source simulator in academia. However, even though gem5 may provide some reasonable degree of hardware representativity, it does not model any processor at RTL.

The Berkeley Out-of-Order Machine (BOOM) processor was chosen as an alternative to the aforementioned strategies, its RTL description is open-source and is not as complex as the OpenSPARC one. These properties already justify choosing BOOM as our target processor. However, BOOM has also the advantage of being parameterizable, hence it facilitates the evaluation of a vast range of different processor configurations.

BOOM is a superscalar, out-of-order RISC core *designed to serve as the prototypical baseline processor for future micro-architectural studies of out-of-order processors* (CELIO; PATTERSON; ASANOVIé, 2015). It is inspired by the MIPS R10k (YEAGER, 1996) and in the Alpha 21264 (KESSLER, 1999) out-of-order processors. It implements the RISC-V instruction set architecture (ISA), detailed in (WATERMAN et al., 2014) and (WATERMAN, 2016). This ISA is shortly reviewed in Section 2.4.

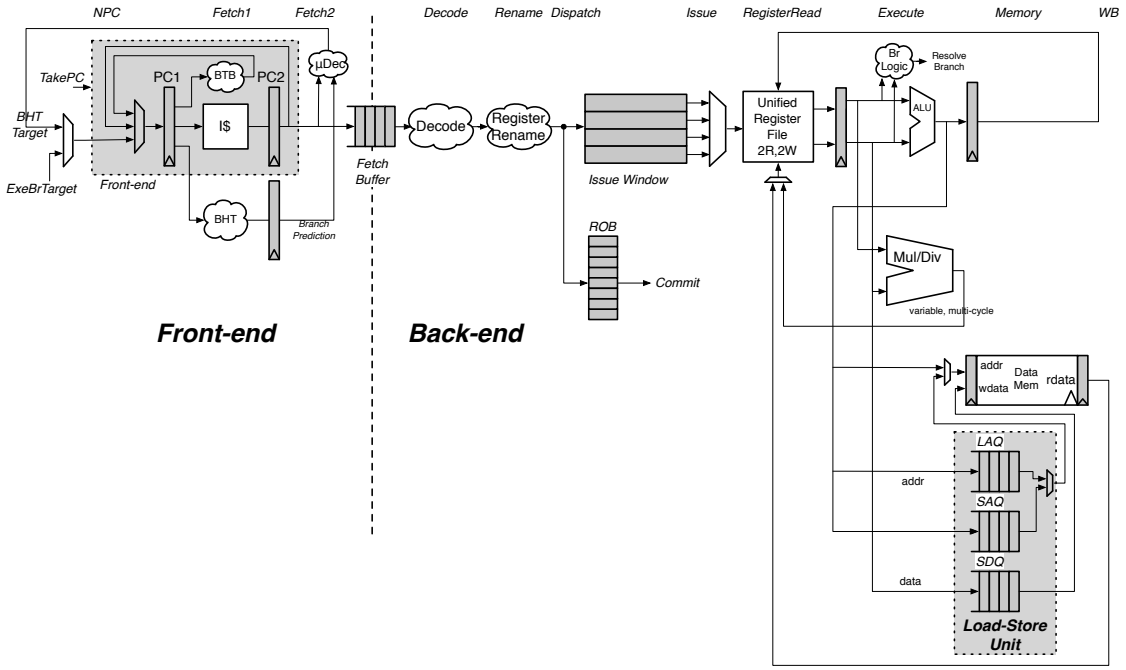
BOOM is currently embedded in the *Rocket Chip* System on Chip (SoC) generator, presented in (ASANOVIć et al., 2016). Most notably, BOOM is developed in the Chisel hardware construction language, which is introduced in Section 2.3.

Since this work is about a fault injection platform for the BOOM processor, it is important to introduce its basic architectural functionalities. A basic knowledge of the hardware structures of the processor is necessary to understand its behavior under the presence of faults. Therefore, the architecture and organization of the processor are described in the next sections.

2.2.2 Architecture and Organization

BOOM supports pipelined execution of multiple instructions in the same cycle. In theory, instructions may execute in up to 10 pipeline stages: *Fetch, Decode, Regis-*

Figure 2.1: A high-level view of the BOOM processor.



Source: (CELIO; PATTERSON; ASANOVIé, 2016)

ter Rename, Dispatch, Issue, Register Read, Execute, Memory, Writeback, and Commit. But, in practice, BOOM combines some of the stages, which reduces the pipeline to six stages: *Fetch, Decode/Rename/Dispatch, Issue/Register Read, Execute, Memory, and Writeback*. The instruction commit stage (sometimes called instruction retirement) occurs asynchronously so it is not considered part of the pipeline.

Figure 2.1 depicts a high-level model of the BOOM processor. The main point of interest of the following sections is to point out some of the hardware components that constitute the processor, which are subject of interest for the BOOM's fault sensitivity analysis and characterization. As Figure 2.1 shows, the processor is constituted of:

- Register file (RF): A unified physical register file constituted of many more registers than the programmer-visible logical registers.
- Reorder buffer (ROB): Hardware component responsible for committing the instructions in order, even though they are executed out of order. The ROB makes sure the out-of-order execution of the instructions does not change the expected program behavior by providing the "illusion" that instructions execute in-order, preserving the meaning of the program. Also, the ROB facilitates the branch prediction of the processor and guarantees precise exception handling.
- Issue window: Instructions are decoded and placed in the instruction window, where they wait until their resources are free so that they can be issued and exe-

cuted.

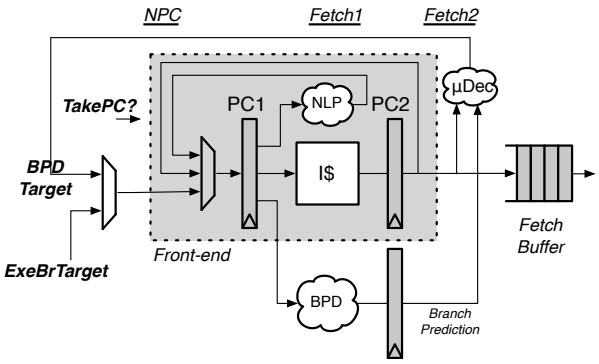
- Branch predictor: Constituted of a branch history table (BHT), a branch target buffer (BTB), a return address stack (RAS), and other components responsible for the management of branch prediction.
- Register rename: Circuitry that handles the register renaming process by mapping logical registers into physical registers, so that some data hazards can be resolved. It enhances the processor performance by increasing the ILP.
- Load/store queue: Constituted of three queues: Load Address Queue (LAQ), Store Address Queue (SAQ), and Store Data Queue (SDQ). These queues are responsible for handling memory operations when load and store instructions execute.
- Execution units and bypass network: Constituted of arithmetic and logic units (ALU), floating point units (FPU), and integer multipliers and dividers. ALU results may be fed back (bypassed) to the ALU operands before results are written to the RF or memory.

As can be observed in Figure 2.1, BOOM is divided into a front-end and a back-end "part". The front-end handles the instruction fetch and a single-cycle branch prediction; in this stage, instructions are fetched from the instruction memory and written into a fetch buffer of instruction. The fetch buffer decouples the instruction fetch stage in the front-end from the subsequent stages in the back-end (Decode/Rename/Dispatch, Issue/Register Read, Execute, Memory, and Writeback).

BOOM's instruction pipeline functionality may be summarized as follows: first, a packet of instructions is *fetched* from the instruction cache and is stored in the fetch buffer of instructions while branch prediction is made if there are branch instructions in the packet. Later, instructions residing in this buffer are pulled out and *decoded* and the logical identifiers of the registers in the instructions are mapped into physical registers residing in the register file (i.e., *register renaming*). In the same cycle, the instructions are *dispatched* to the ROB and to the instruction window.

Instructions residing in the instruction window wait there until all their resources are available, when they can finally be *issued* out of the instruction window and begin *execution*. Once the instruction ends execution, it then sends signals to the reorder buffer, which takes the appropriate actions to make the state of the instruction visible (i.e., results are written in the RF or memory) by *committing* the instruction if it was not miss-speculated and did not cause any exception.

Figure 2.2: The fetch unit of the BOOM processor.



Source: (CELIO; PATTERSON; ASANOVIé, 2016)

2.2.2.1 Instruction Fetch

The front-end is responsible for fetching a *fetch packet* of instructions from the instruction memory and storing them in a *fetch buffer* of instructions (a First-in/First-out queue). In the subsequent pipeline stages, instructions residing in the fetch buffer are decoded and renamed before they are stored in the *instruction window* and in the ROB. These phases are handled in-order, and out-of-order execution only starts when instructions are issued out from the instruction window to the execute stage.

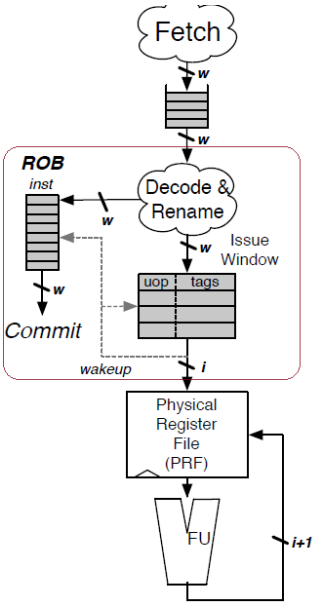
Figure 2.2 depicts the BOOM's fetch unit. As multiple instructions may execute at the same cycle, BOOM fetches a packet of instructions each cycle. Instructions being fetched from the instruction memory are place in a *First-in First-out* (FIFO) *fetch packet* of instructions.

In parallel with the instruction fetch, BOOM predicts every cycle (and in a single cycle) where the next instruction should be fetched from by using the Next-line predictor (NLP) branch prediction strategy. The BOOM's branch predictor will be detailed in Section 2.2.2.9.

2.2.2.2 Instruction Decode

The decode stage takes instructions from the fetch buffer (in-order), decodes them, and allocates the necessary resources as needed by each instruction. This stage will stall if not all resources are available. Otherwise, after instructions are decoded, they are renamed and dispatched to the instruction window and to the reorder buffer, as shown in Figure 2.3.

Figure 2.3: The instruction decode unit.



Source: Adapted from (CELIO; PATTERSON; ASANOVIé, 2016)

2.2.2.3 Register Renaming

As stated in Section 2.1, superscalar processors make extensive use of ILP. But due to the limited number of logical registers in most processors, the ILP may be drastically reduced by data hazards, such as true dependencies, or read after write (RAW), output dependencies, or write after write (WAW), and anti-dependences, or writer after read (WAR).

Register renaming is a technique used to overcome the WAW and WAR hazards. It consists in mapping the *logical registers* (those visible by the programmer) into *physical registers* in the register file.

Even though the RISC-V ISA defines 32 user-visible (logical) registers, BOOM implements a physical register file with many more registers that hold both the committed architectural register state and speculative register state. Also, BOOM implements a unified physical register file: both integer and floating point registers reside in a single register file.

BOOM relies on an *explicit renaming* technique, which means the logical registers are explicitly mapped to physical register in a register file by means of *rename map tables*, which maps logical registers to physical ones by using the logical register identifiers as indexes.

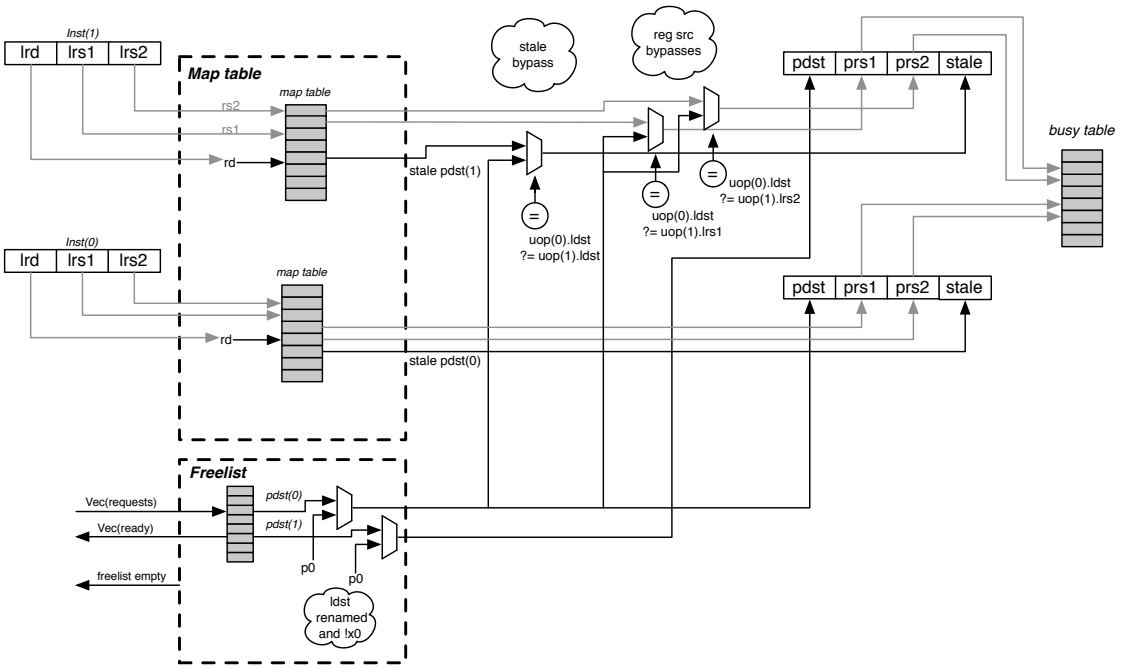
Register renaming also requires circuitry to detect dependencies between registers being renamed. For instance, when a register being renamed depends upon an earlier

instruction, the dependency check logic consists in setting the appropriate control signals in the multiplexers depicted in Figure 2.4 which select the appropriate physical register designator.

The state of the physical registers has to be tracked every cycle, so that appropriate actions may be taken during the register renaming process. The main hardware structures that handle the renaming are:

- Map table: The Rename Map Table contains the assignments from logical to physical registers. More precisely, it contains the speculative mappings from ISA registers to physical registers. Each branch will have its own copy of the rename map table, if there is a branch mispredict, the map table can be reset instantly from the mispredicting branch's copy of the map table.
- Free list: The free-list is a bit-vector that tracks the physical registers that are currently unused and is used to allocate new physical registers to instructions passing through the Rename stage. It contains as many bits as there are registers in the physical register file. A value '1' in the bit 5, for example, means the physical register of index 5 is free. Multiple registers may be allocated in a single cycle.
- Busy table: This structure holds the status and readiness of the physical registers. When an instruction is to be issued, the busy table is consulted to check if its operands are ready and contain valid values. When a register leaves the free list,

Figure 2.4: The rename stage - logical registers are mapped into physical ones.



Source: (CELIO; PATTERSON; ASANOVIé, 2016)

its corresponding bit in the busy table is set to the *busy* state. When a register is written by an execution unit, its corresponding bit in the busy table is reset.

- Stale destination specifiers: For instructions that will write a register, the map table is read to get the stale physical destination specifier (*stale pdst*). Once the instruction commits, the stale pdst is returned to the free list, as no future instructions will read it.

Figure 2.4 depicts a high-level view of the register renaming circuitry. Up to two instructions go through the rename stage at the same cycle and their respective source and destination registers are compared in order to detect possibly dependencies. Any renamed register has to be passed to dependent instructions. When the logical registers read the map table, physical registers are provided for the instruction and the free list and busy table are updated.

2.2.2.4 The Instruction Issue Unit

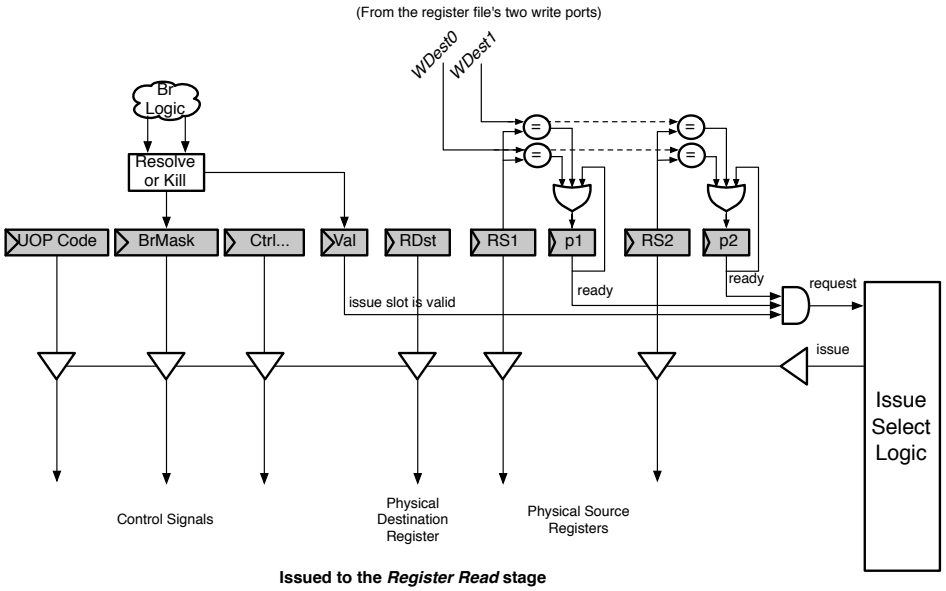
Once instructions are decoded and renamed, micro-operations are generated and then dispatched to the instruction window where they wait until all of their operands are ready and resources such as execution units are free. The 'p' bit depicted in Figure 2.5 stands for *presence*, and indicates when the source registers are available in the register file. Once this criterion is met, the instruction can be issued by setting a *request* bit high. The issue select logic selects a slot with a *request* bit high. Issued micro-operations can then read their operands in the register file and their issue slot is freed to make room for another dispatched instruction.

BOOM's instruction window is unified, thus both integer and floating point instructions are placed in a single issue window. An issue slot is depicted in Figure 2.5.

There are two issue policies available in BOOM:

- *Age-ordered Issue*: dispatched instructions are placed into the bottom of the issue window (at lowest priority). Every cycle, every instruction is shifted upwards. Thus, the oldest instructions will have the highest issue priority. For out-of-order superscalars, branches and load instructions should be resolved as soon as possible, hence this policy can increase the performance of the processor since instructions' priorities tend to increase, hence branches and loads would not be "stuck" at low priorities. However, this policy may result in poor energetic performance since each slot may have to be read and write at each cycle.

Figure 2.5: The issue slot.



Source: (CELIO; PATTERSON; ASANOVIé, 2015)

- *Un-ordered Issue*: dispatched instructions are placed into the first available issue window slot and remain there until they are issued. This policy may harm the performance if branches or loads are inserted in the lowest priority slots and are not able to be issued as soon as possible. However, there is no energy penalty such as in the *Age-ordered Issue*.

Each issue select logic port uses a static-priority encoder that selects the first available instruction in the issue window. Each port will only schedule an operation that its port can handle (e.g., floating point micro-ops will only be scheduled onto the port governing the Floating Point Unit).

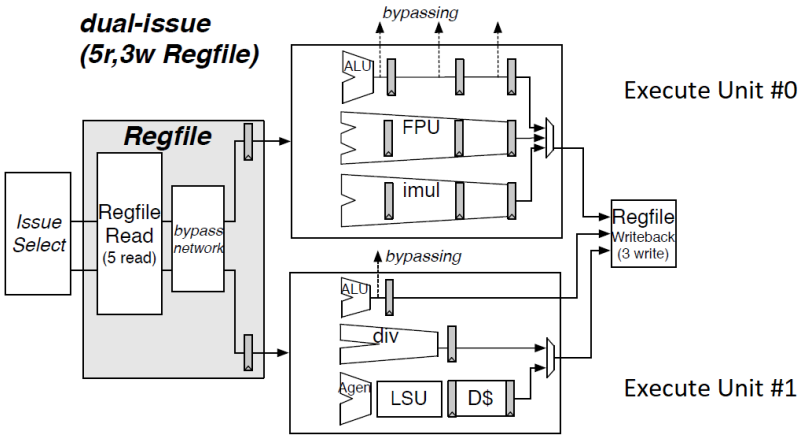
2.2.2.5 The Execute Stage

BOOM makes use of *Execute Units* (EUs) in order to operate on data. Each EU will hold different units of execution, as depicted in the dual-issue version of BOOM in Figure 2.6.

Different EUs are composed of different functional units to execute specific instructions. As an example, in Figure 2.6 the *Execution Unit #0* handles ALU, FPU and integer multiplication operations, while the *Execution Unit #1* handles ALU, integer division and load/store operations.

Each EU is connected to a single issue port in the Issue Window, so for each issue

Figure 2.6: A dual-issue BOOM has two *Execute Units*.



Source: Adapted from (CELIO; PATTERSON; ASANOVIé, 2016)

port, there will be only one associated EU. Therefore, the number of EUs in BOOM corresponds to the processor's issue-width, so for a dual-issue, for example, there are two EUs. Instructions are issued to the appropriate EU by an *Issue scheduler*. The issue scheduler will only schedule instructions that the Execution Unit supports.

2.2.2.6 The Register File and Bypass Network

BOOM has a unified physical register file (PRF), which dictates that the register file holds both the integer, floating point, committed and speculative instructions.

As the BOOM's floating point functional units work with 65-bit operands, the register file is comprised of 65-bit registers. ALU operations can be issued back-to-back by having the write-back values forwarded through the bypass network. Bypassing occurs at the end of the Register Read stage.

The number of read and write ports in the register file depends on the issue width. For a single-issue processor, for instance, the register file needs 3 read ports to satisfy fused multiply–add (FMA) [1] operations and 2 write ports.

For the issued instructions, the register file statically provides the register read ports as in the following example: if for a dual-issue processor, for example, *issue port #0* provides access to the Execute Unit that holds an ALU, and *issue port#1* provides access to the Execute Unit that handles FPU operations, then the first two read ports will statically serve the ALU, and the remaining three read ports will operate on the FPU.

[1] FMA is a floating-point multiply–add operation performed in one step. For operations such as $a \times b + c$, the whole expression is first evaluated, then the result is rounded. This contrasts with unfused multiply–add, where the product a x b is first calculated and rounded, then the result is added to c and, finally, it is rounded again.

2.2.2.7 The Load/Store Unit

Load and store instructions are handled by means of the Load/Store Unit (LSU), which consists of three queues: the Load Address Queue (LAQ), the Store Address Queue (SAQ), and the Store Data Queue (SDQ). The LSU decides when memory operations should be fired to the memory.

When a load instruction is issued, its address is first calculated and placed in the LAQ. Stores are inserted into the issue window (dispatched) as a single instruction (i.e., a single instruction handles address and data generation). When a store instruction is issued, two actions may be taken depending on the conditions, it may either fire a Store Address Generation (STA), or a Store Data Generation (STD). The STA micro-op calculates the store address and places its result in the SAQ queue. The STD micro-op moves the store data from the register file to the SDQ. Each of these micro-ops will issue out of the Issue Window as soon their operands are ready. More precisely, load and store instructions are performed as follows:

- Store instructions: Stores are issued out of the instruction window to the LSU. When both operands are ready, the store can be issued to the LSU as a single micro-op which provides both the address and the data. However, often the address will be available before the data, hence only the STA micro-op is issued to the SAQ to allow later loads to avoid any memory ordering failures. Once a store instruction is committed, the corresponding entry in the Store Queue is marked as committed. The store is then free to be fired to the memory system. Stores are fired to the memory in program order.
- Load instructions: For loads, entries in the LAQ are allocated during the *Decode* stage. In Decode, each load entry is also given a store mask, which marks which stores in the Store Queue the given load depends on (if any). Loads are fired to memory as soon as they arrive in the LAQ, because reading memory contents as soon as possible is convenient for out-or-order pipelines. When a load is to be performed, its load address is simultaneously compared to all the store address it depends on, if there is a match, the memory request is killed. If the corresponding store data is present, then the store data is forwarded to the load and the load is considered successful. If the store data is not present in the SDQ, then the load goes to *sleep*, where they wait until they are retried at a later time.

The LSU must also handle load/store dependencies, which happens when a load

instruction depends on the contents of a memory address written by a previous store instruction. Such dependencies may cause memory order failures. If an ordering failure occurs, then the pipeline must be flushed and the rename map tables are reset.

To discover ordering failures, when a store commits, it checks the entire LAQ for any address matches. If there is a match, the store checks to see if the load has executed, and if it got its data from memory or if the data was forwarded from an older store. In either case, a memory ordering failure has occurred.

2.2.2.8 The Reorder Buffer and the Commit Stage

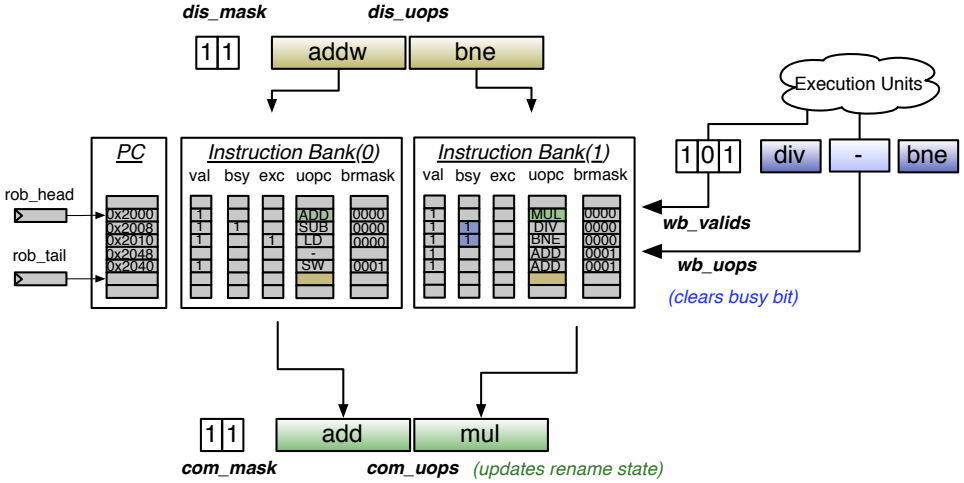
The reorder buffer (ROB) is a circular buffer organized with N banks, where N is the dispatch and commit width of the machine (see Figure 2.7). The ROB tracks the state of all inflight instructions in-order. After instructions are decoded and renamed, they are *dispatched* from the fetch packet and written into the *tail* of the ROB.

Each dispatched instruction is written to a different bank across a row in the ROB. Once instructions are dispatched, they are marked as *busy*, which means the instructions are *not complete*. By doing so, older instructions will always reside in the *head* of the ROB, while the newest ones will reside in the *tail*.

Once instructions end execution, they are marked as *not busy*. When a *not busy* instruction "moves" to the head of the ROB, and it is not miss-speculated and did not cause any exception, it is then *committed* and its results are made architecturally visible.

As an example, Figure 2.7 depicts the ROB of a two-wide version of BOOM. Up

Figure 2.7: A ROB for a two-wide BOOM - up to two instructions can be dispatched and written to a single ROB row.



Source: (CELIO; PATTERSON; ASANOVIé, 2016)

to two instructions can be dispatched to the ROB per cycle, each of them will be written to the head of the two ROB banks, and they will be written in the same *row*. In that case, a ROB *row* consists of two instructions, which is the maximum number of instructions that can be committed in a cycle [2].

Since superscalar commit is supported, the entire ROB row is analyzed for *not busy* instructions, so that the ROB may commit as many instructions as possible per row in a single cycle, which in turn releases as many resources as possible. Be aware that the ROB does not look across multiple rows in order to find instructions that can be committed (i.e., only the ROB head is committed in a cycle).

As can be seen in Figure 2.7, for each bank in the ROB, each row contains a *branch mask* that "tells" the instruction in the bank of a row which branch the instruction is speculated under. If a micro-op in the ROB "belongs" to a miss-speculated branch, it is then flushed and its architectural state is not made visible.

The ROB must know the PC for each inflight instruction since 1) any instruction may cause exceptions, so the exception program counter (EPC) must be known for latter recovery of the program context, 2) branch and jump instructions must know their own PC for the correct target calculation, and 3) jump-register instructions must know both their own PC and the PC of the following instruction to verify if the front-end predicted the correct JR target. Since instructions are fetched, decoded and dispatched to the ROB in-order (hence are located consecutively in memory), keeping a single PC for each ROB row is enough.

Each entry in the ROB is marked with an *exc* flag that indicates whether or not the instruction has caused an exception. An exception will be thrown only if the excepting instruction is in the head of the ROB. When an exception is thrown, the pipeline is flushed and the ROB emptied.

Note that the instruction is still speculative during the register renaming stage, thus the correspondence between physical and logical registers stored in the rename map tables may be miss-speculated. If that is the case, the rename map tables will be invalid. Therefore, if the head of the ROB is marked as *not busy* (i.e., if the instruction ends execution) and it is miss-speculated or caused an exception, then the rename map tables must be reset so that the miss-speculated or excepting mapping of logical to physical registers is eliminated, and the correct architectural state is maintained.

[2] Note that the dispatch and commit width are the same.

2.2.2.9 The Branch Predictor

BOOM supports branch prediction and speculation. The branch prediction occurs at two distinct levels: while one prediction is made combinationally by the *Next-Line Predictor* (NLP) in the BOOM's front-end during the instruction fetch, another prediction is made by the *Backing Predictor* (BPD) in the back-end part.

During the instruction fetch, the NLP takes the current *Fetch-PC* (the current program counter that fetches the fetch packet of instructions) as input and works together with a branch history table (BHT), a branch target buffer (BTB) and a return address stack (RAS) depending on what kind of instruction is being speculated (conditional, unconditional or a return instruction).

First, the Fetch-PC is compared to all entries (PC tags) in the BTB in order to find any tag match. If a match occurs, and if the current instructions is a *ret*, then the return address for the instruction is retrieved from the RAS. If the instruction is a conditional branch, then the BHT is consulted in order to make the prediction, and the predicted branch target will be retrieved from the BTB so that the NLP does not have to wait for the execute stage to calculate the target address. If the instruction is an unconditional branch, then the BHT is not consulted.

Since the NLP has to store PC tags and branch targets in the BTB, it becomes really expensive in terms of area and power, so only a few dozen branch predictions can be stored. To overcome that, BOOM makes use of the BPD as a second level branch predictor. The BPD is not as expensive in terms of power and area as the NLP because it does not retrieve branch targets from the BTB, instead, it actually computes the target address during the execute state.

The BPD does not make any prediction before the *fetch packet* has been decoded and the branch targets are computed directly from the instructions themselves. Therefore, there is no need for the BPD to store predicted target addresses.

When a prediction is being performed in the subsequent stages in the back-end, the BPD provides a bit-vector of taken/not-taken predictions, for which there is one bit for each instruction fetched. When the instructions from the fetch packet are decoded and the target branches are calculated, the prediction bits in the bit-vector are consulted in order to decide if the processor's front-end has to be redirected or not. The BPD will be updated in two distinct phases: during the execute stage, if a misprediction is detected, and during the commit stage so that the branch prediction is updated only with non-speculative state.

2.2.3 Parameterization of the BOOM Processor

As will be detailed in Section 2.3, Chisel enhanced the way modern and complex systems are designed by benefiting from facilities that allow the step-by-step and high-level development of individual components that together may form a more complex system, such as a processor.

BOOM was implemented in the Chisel hardware construction language and, by convenience of design, there are "knobs" in its source code that make it easy to parameterize the processor. This way, for BOOM, the main structure sizes of the processor can be chosen by the user before the processor description is compiled, and no further modifications in the source code of the processor are required.

Some of the possible configurations of the BOOM processor that can be easily parameterized are:

1. Fetch/Decode/Commit width: Defines the maximum number of instructions that can be fetched, decoded and committed per cycle. Note that the maximum number of decoded and committed instructions in a cycle, on average, is limited by the fetch width, so actually only the maximum number of fetched instructions has to be parameterized.
2. Issue width: Defines the maximum number of instructions that can be issued out of the instruction window in a cycle.
3. Issue scheduler policy: Can be selected as Un-ordered or Age-ordered.
4. Map table: The commit map table can enable or disabled.
5. ROB size: Defines the number of entries in the ROB.
6. Issue window size: Defines the number of instructions that can reside in the instruction window.
7. LSU size: Defines the number of entries in the load/store queues.
8. RF size: Defines the number of physical registers in the RF.
9. Fetch buffer size: Defines the size of the fetch buffer that holds the fetched instructions.
10. Enable/Disable BPU: The backing branch predictor can be either enabled or disabled.
11. Max in-flight branches: Defines the maximum number of branches that can be issued per cycle.

12. RAS/BTB: Defines the sizes of the RAS and BTB structures.
13. Cache configurations: Defines the number of sets and ways in the L1 instruction and data caches. It is also possible to enable or disable the L2 cache and configure its parameters.
14. Latencies: The FPU and integer and multiplier and dividers latencies can be defined.

Note that because it is easy to parameterize the BOOM processor, and there are plenty of possibilities that can be parameterized, the design space to be explored in terms of fault injection and fault tolerance tends to be high. In our fault injection analysis, for instance, we could easily experiment with three different configurations of the BOOM processor just by setting the appropriate values of the structure sizes.

2.3 Chisel - Constructing Hardware in an Scala Embedded Language

BOOM is developed in the Chisel (Constructing hardware in a Scala embedded language) hardware construction language, which is derived from the Scala programming language (BACHRACH et al., 2012). This language can be understood as a set of predefined classes and libraries embedded in Scala. Chisel leverages Scala and gives it the ability to abstract and to design hardware components with high-level programming features, such as object orientation, functional programming, parameterized types, and type inference.

Moreover, Chisel can generate optimized C++-based cycle-accurate simulators and low-level Verilog that can be mapped either to field-programmable gate array (FPGA) or application-specific integrated circuit (ASIC). The various Chisel backends are shown in Figure 2.8.

The Chisel language was elaborated aiming to solve three common problems faced by traditional hardware description languages (HDLs):

1. Because the most popular hardware-description languages (e.g., Verilog/VHDL) were originally developed as hardware simulation languages, they were only later adopted for hardware synthesis, hence their semantics are based on simulation, which complicates synthesizable designs.
2. These languages did not evolve at the same pace modern programming languages did. So they lack the powerful abstraction facilities that are common in modern

software languages, such as the object-oriented paradigm. This aspect lowers the productivity as it is difficult to reuse components, for example.

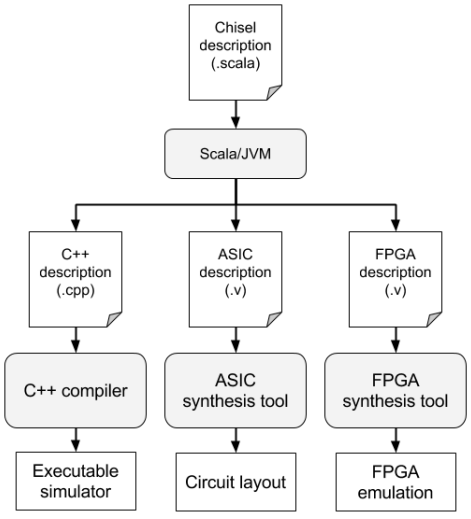
3. These languages make it difficult to explore the design-space of complex structures, as they have limited module generation facilities, which makes it difficult to produce and compose highly parameterized modules required to support thorough design-space exploration.

To overcome the deficiencies in (1)-(3), Chisel was designed as a language that *leverages great ideas in software engineering in hardware design*. In other words, Chisel takes the modern facilities and aspects of a modern programing language (Scala) and enhances the design of complex, parameterizable, flexible and synthesizable hardware structures. Therefore, the language improves the productivity as its modern features provide facilities to design hardware by composing smaller and reusable structures with way better semantics than Verilog/VHDL.

Differently from the traditional concept of HDLs, where the hardware description is translated into a netlist, Chisel works by a transformation of its source code into equivalent C++/Verilog source codes. Since Scala is compiled to Java bytecodes, the process of code translation from Scala to C++/Verilog works by compiling the Scala and then running such bytecodes in the Java Virtual Machine (JVM). The JVM then constructs an abstract syntax tree (AST) representing the hardware description, which in turn is translated into C++/Verilog equivalent source codes.

The C++ Chisel-exported code defines an “RTL” object-oriented simulator that

Figure 2.8: The Chisel code transformation flow.



Source: Author

Figure 2.9: Chisel description of the multiplexer and its equivalent transformed C++.

(a) Chisel description of the multiplexer.

```
/*Chisel description of a 2-input multiplexer
(.scala)*/

class Mux2 extends Module
{
  val io = new Bundle
  {
    val sel = UInt(INPUT,  1)     // 1-bit input
    val in0 = UInt(INPUT,  1)
    val in1 = UInt(INPUT,  1)
    val out = UInt(OUTPUT, 1)  // 1-bit output
  }

   val reg = Reg (UInt(1))   // 1-bit register

  /* reg: stores mux output */
  reg := (io.sel & io.in1) | (~io.sel & io.in0)
  io.out := reg;
}
```

(b) C++-transformed multiplexer module.

```
/* Transformed Mux2 description (.cpp) */

class Mux2_t : public mod_t
{
  public:
    dat_t<1> Mux2__io_in0;
    dat_t<1> Mux2__io_sel;
    dat_t<1> Mux2__io_in1;
    dat_t<1> Mux2__reg_;
    dat_t<1> Mux2__io_out;
    dat_t<1> T3;
    clk_t clk;

    /* Combinational updates */
     void clock_lo ( dat_t<1> reset );

     /* Sequential updates */
     void clock_hi ( dat_t<1> reset );
};
```

(c) C++ behavioral description of the multiplexer.

```
/* clock_lo(): Updates combinational logic */
void Mux2_t::clock_lo ( dat_t<1> reset)
{
  val_t T0;
  { T0 = ~Mux2__io_sel.values[0];}
  T0 = T0 & 0x1L;
  val_t T1;
  { T1 = T0 & Mux2__io_in0.values[0];}
  val_t T2;
  { T2 = Mux2__io_sel.values[0] &
    Mux2__io_in1.values[0];}
  { T3.values[0] = T2 | T1;}
  { Mux2__io_out.values[0] =
Mux2__reg_.values[0];}
}

/* clock_hi(): Updates sequential logic */
void Mux2_t::clock_hi ( dat_t<1> reset )
{
  dat_t<1> Mux2__reg___shadow = T3;
  Mux2__reg_ = T3;  // Update register
}
```

(d) An instance of the multiplexer.

```
#include "Mux2.h"

void main (int argc, char* argv[])
{
  /* An instance of a Mux2 module */
  Mux2_t module;

  /* Assign Mux2 inputs */
  module.Mux2__io_in0 = 0;
  module.Mux2__io_in1 = 1;
  module.Mux2__io_sel  = 1;

  dat_t<1> reset = 0;

  /* Updates combinational logic */
  module.clock_lo (reset);

   /* Updates sequential logic*/
  module.clock_hi (reset);
}
```

Source: Author

honors the behavior of the system in a low-level mode. One could consider this exported simulator as being, somehow, a C++ RTL description of the system, since it is as detailed as its corresponding Verilog representation.

Figure 2.9 depicts an example of a two-input multiplexer written in Chisel and its C++-transformed code. After this model is described in Chisel, the Chisel build process generates the equivalent C++-modeled simulator in Figure 2.9b.

Note how high-level the Chisel model is; even the clock signal is implicit to the designer, so it is not necessary to handle clock events. Conversely, Verilog/VHDL models have to manage clock events explicitly. In Chisel, the clock signal is automatically created in the corresponding C++ model, as can be seen in Figure 2.9b. Also, note how there are high-level classes such as *Reg* that create abstract *concepts* of elements, which is a sophisticated object-orientation semantics applied in hardware descriptions. These object-oriented concepts may increase the designer productivity significantly.

Figure 2.9b depicts the equivalent multiplexer module transformed into C++. The C++ simulator is based on a fast multi-word library using C++ template classes, where the signals and registers are represented by the *dat_t* special class definition (memories defined in Chisel will be represented as *mem_t* objects in C++). Note also that the Chisel transformation topologically sorts nodes based on dependencies by preserving the original variable names while prefixing them with their top module name.

Figure 2.9c depicts the behavioral description of the multiplexer. The behavior of the system in the exported code is based on the *clock_lo()*, and *clock_hi()* primitives. The *clock_lo()* method handles all of the combinational updates, while the *clock_hi()* handles the sequential ones. Notice that this model yields a cycle-accurate representation of the system.

Figure 2.9d depicts an example instance and usage of the multiplexer module. In this phase, all the user has to do is to feed the module with the inputs and invoke the *clock_lo()* and *clock_hi()* method appropriately.

Notice in Figure 2.9 that there is a 1:1 correspondence between the Chisel model and the C++-exported code (also valid for the Verilog-exported one). Each signal and register represented in the Chisel will be represented in the C++ simulator with exactly the same number of bits. In other words, there is no single loss of information about the system's model when the code is transformed from Chisel to C++.

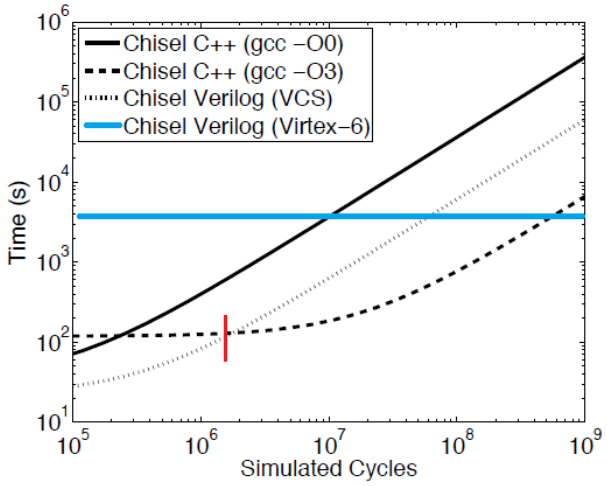
As simulation speed is often an important aspect to be considered, a study in (BACHRACH et al., 2012) compared the performance of the C++-based simulator to

Figure 2.10: Performance of the Chisel-generated Verilog/C++.

(a) Simulation times for the Chisel C++ simulator, Synopsys VCS Verilog simulation, and FPGA emulation when a 64-bit five-stage RISC processor boots an OS.

Simulator	Compile Time (s)	Compile Speedup	Run Time (s)	Run Speedup	Total Time (s)	Total Speedup
VCS RTL simulator	22	1.000	5368	1.00	5390	1.00
Chisel C++ RTL simulator	119	0.184	575	9.33	694	7.77
Virtex-6 FPGA	3660	0.006	76	70.60	3736	1.44

(b) Total time required to compile and simulate a system using various Chisel back-ends.



Source: Adapted from (BACHRACH et al., 2012)

the performance of the equivalent Verilog by using the Synopsys Verilog Compiler Simulator (VCS), where an RISC five-stage pipelined processor was implemented and used to boot an operating system (OS). Figure 2.10 compares the significant difference between performances of various Chisel backends.

Results show that the C++ simulator booted the OS 7.7 times faster than the VCS simulator, as shown in Figure 2.10a. This figure compares the total time taken (compile time + run time) for various Chisel backends in order to boot the OS.

Notice how the compilation time is the bottleneck for the FPGA, so even though the FPGA boots the OS about $575/76 \approx 7.5$ times faster than the C++ simulator, the total time required to compile the processor and to configure the FPGA makes it slower than the C++ backend. Because compilation time is not negligible and it is constant, the fastest method will depend on the number of cycles the application executes. As an example, note that the C++ simulator will only be faster than the VCS when millions or more cycles are simulated, this limit is highlighted by the red dash in Figure 2.10b. Notice also that the performance of the C++ simulator is highly dependent on compiler optimizations.

2.4 The RISC-V ISA

RISC-V is an open source Reduced Instruction Set Computer (RISC) ISA presented in (WATERMAN et al., 2014) and (WATERMAN, 2016). This ISA is divided in a small base integer ISA, useful for education an research, and an optional extension that supports applicable software development. Also, the ISA is subdivided in different categories that may be deployed on 32-, 64-, and even 128-bit architectures, depending on the requirements and performance/energy trade-offs. While the base ISA is simple and suitable for education and research, it is efficient enough to be used in low power devices.

Conceptually, the ISA is subdivided in the subsets RV32I/RV32G, RV32E, and RV64I/RV32G. These categories implement the same instructions and behave similarly, differing only on the width of the physical registers and the size of the memory space. RV32E is a variant of RV32I, it implements the same instructions, but contains half the number of registers, making it suitable for small, power-constrained and energy-efficient applications. BOOM implements the RV64G ISA variant as its register file consists of 65-bit[3] registers.

The RV32I is the most basic and simple variation of RISC-V. It implements 47 instructions, subdivided in system instructions (system calls and performance counters), computation, control flow, and memory access instructions. The ISA is termed a load/-store architecture, as only load/store operations can transfer data between the processor and the memory. The arithmetic and logic instructions operate only on registers.

Both RV32I and RV64I ISAs implement 32 logical registers (x0-x31), 31 of them are of general-purpose use, and register x0 is hard-wired to the constant zero. Instructions are fixed 32-bit long, and are classified in computation, control flow, and memory access instruction, and each of this classes of instruction has a different format.

Each instruction consists at most of one destination and two source registers (i.e., computation). Source and destination register specifiers always occupy the same position in different instructions, regardless of the instruction class. That aspect simplifies the processor implementation and allows for the register fetch and decode to occur in parallel.

As this dissertation does not intend to be a reference manual on the RISC-V instruction set, the following items just shortly illustrate some of the basic aspects of the ISA instructions, which are classified as:

[3]BOOM's physical registers are 65-bit wide because it implements the Berkeley hard float floating point units which use internal 65-bit operands.

- Computational Instructions: Comprised of 21 arithmetic, logic, and comparison instructions. They operate between two integer registers, or between one integer register and one immediate value, and save the results in a destination register. Arithmetic operations are addition, subtraction, and bitwise shifts. The logical operations perform bitwise Boolean operation, such as logical *AND*, *OR*, and *XOR*. Comparison operations perform arithmetic magnitude comparisons between two sources and write either 0 or 1 values into a destination register depending on the comparison result.
- Memory Access Instructions: Comprised of load and store instructions. Load instructions are divided into five different types that can load a single byte (LB or LBU), 16-bits (LH or LHU), or 32-bits (LW) from memory into a destination register. The 'U' stands for unsigned. There are three possible store instructions, SW (for 32-bits), SH (for 16-bits) and SB (for a single byte).
- Control Flow Instructions: Comprised of six conditional branch instructions that work by comparing the contents of two source registers and then changing the program flow in a rage of $\pm$1K instructions. Also, there are two unconditional branch instructions that may set the program counter to a given address while saving the address of the next instruction in a destination register, so that it is possible to return to the address of the instruction fallowing the branch.
- System Instructions: Comprised of eight instructions that can perform system calls, invoke the debugger and operate on control and status registers (CSR). The current CSRs hold the values of the cycle counter, real-time clock and number of instructions retired. These are 64-bit counters, so each of them is implemented as two 32-bit registers and there is a specific instruction to read each of them, yielding six instructions necessary to access the CSR registers.

Beyond the basic functionalities required by any ISA, RISC-V offers additional extensions that makes the ISA flexible. The extensions are termed "MAFD". **M** provides additional instructions that perform integer multiplication and division. **A** stands for atomic memory operations useful for synchronization of parallel applications. **F** stands for single-precision floating point operations, it adds 30 new instructions to operate on data movement (load/store), conversions, comparisons, and arithmetic instructions, all of them operating on floating point data. **D** stands for double-precision floating point operations. This extension is similar to the **F** extension, but operates on 64-bit registers and new instructions are added to operate on double-precision values.

When a base integer ISA RV32I/RV64I is extended to handle the MAFD extension, the extended version of the ISA is termed RV32G/RV64G. Where the abbreviation G is used for the combination of the base integer ISA plus the MAFD extension (i.e., G is an alternative for the term IMAFD).

3 BACKGROUND ON FAULT INJECTION AND RELATED WORK

3.1 Soft Errors and Technology Scaling

Until a few decades ago, most of the concerns with soft errors were related to 1) space applications, where the levels of radiation-induced faults are high due to cosmic rays; and 2) alpha particles emitted by radioactive impurities of uranium and thorium in packaging materials operating at ground level. The latter proved to be the dominant cause of soft errors in DRAM devices during the 1970s (BAUMANN, 2005). Moreover, a study in (MAY; WOODS, 1979) discovered that alpha particles imposed significant contributions to soft errors in dynamic memories, and the soft error rate (SER) was found out to be proportional to the alpha particle flux.

Regarding space applications, a study in (BINDER; SMITH; HOLMAN, 1975), for instance, investigated the influence of cosmic rays that caused anomalies on satellites back in the 70's. However, due to the technology aggressive scaling, transistors' sensitivity to faults tends to increase, and nowadays faults can be experienced even in applications operating at ground level.

Soft errors are radiation-induced, non-permanent faults which happen due to a particle hit, e.g., an alpha particle from radioactive decay of impurities in packaging material or a neutron from cosmic rays (CHANDRA; AITKEN, 2008). When a particle hits the transistor of the sequential logic of a system, it creates a transient current pulse. If the current pulse is large enough, it can flip the value stored in a single element (memory cells, latches, flip-flops, etc...). These upsets are called Single Event Upsets (SEU) and are the main contributors to soft errors observed in many of the current technology operating at ground level (DODD et al., 2010). Furthermore, as the density of transistors in a system has increased, the distance between sensitive regions, such as memory cells, decreased. Hence devices containing multiple elements close to each other can experience Multiple Bit Upsets (MBU), which is defined as any event or series of events that cause more than one bit to be upset during a single measurement (REED et al., 1997).

As a consequence of this shrinking process, transistors become more sensitive to soft errors due to the reduction of the critical charge (KARNIK; HAZUCHA, 2004). Furthermore, even though the soft error susceptibility, mainly SEUs, decreases linearly as area decreases, it increases exponentially as voltage decreases. So it should be expected that the voltage reduction that follows the transistors' size aggressive scaling would cause

growth in SEU rates (DIXIT; WOOD, 2011).

Soft errors can affect static random access memories (SRAM), dynamic random access memory (DRAM), combinational logic and sequential elements in a circuit. Each of these structures may be vulnerable, and they have different sensitivity levels and their fault models are considerably different.

A study in (DODD et al., 2010) delineates some trends in memory cells as features' sizes shrink from 200nm to 40nm CMOS technologies. Two main trends can be identified:

- DRAM's SER per bit decreases because DRAM's storage capacitors have not decreased significantly. Hence an individual DRAM cell of today's technology is not significantly more sensitive than it was generations of technology ago. At the same time, the cell's size has indeed shrunk with technology scaling, hence the SER per bit has decreased since there are more bits per area.
- For SRAMs, the decrease in the *charge collection efficiency*, which is a measure of the magnitude of charge generated by a particle strike, has caused growth in SER per bit with device scaling until the last two or three technology generations. However, SRAM's SER appears to be saturated or even became slightly smaller with 90nm and smaller technologies.

A common method used to report the levels of dependability of a system is based on the rate at which soft errors occur: the failures in time (FIT) metric tells that 1 FIT corresponds to 1 failure in one billion hours. The trends observed in the DRAM and SRAM memories led their soft error sensitivities to become considerably different. At 90nm technology, for instance, SRAM and DRAM's SERs are about 800 and 2 FIT/Mbit, respectively, evidencing that SRAMs are considerably more sensitive than DRAMs (DODD et al., 2010). Note, however, that this metric relates to the number of failures per bit, and does not reflect the overall failures observed in today's memories. As technology shrinks, the number of bits in memory grows significantly as the bit density increases, hence the raw failures experienced in SRAMs increases, while the overall number of DRAM's soft errors is roughly constant over the years.

Soft errors can also arise from faults in the combinational logic of the circuit when sufficient radiation induced charges are created in the transistors, these are known as Single-event transient (SET). If such a glitch propagates to the inputs of a latch or flip-flop during the latching clock signal, the noise will be stored in the input, and possibly

will alter the actual correct value that should be stored (BAUMANN, 2005).

Faults in combinational logic can only be observed if the fault hits the cell in appropriate circumstances, which makes the combinational logic much less susceptible to soft errors than memory cells (SHIVAKUMAR et al., 2002). This low susceptibility is justified by three main phenomena that can naturally mask the faults:

1. Logical masking: Occurs when the fault affects an input of a logic gate that will eventually produce a result that is independent of the faulty input. An example could be a two-input AND gate, where one of the inputs is set low. If a fault affects the other input, the result will not be changed, since it is deterministically zero.
2. Electrical masking: Occurs mainly due to the attenuation of the electrical pulse generated by the particle hit. It is expected that signals propagating from combinational logic to a state element tend to be attenuated as it propagates through the circuits.
3. Latching-window or temporal masking: Occurs when the fault propagates to the latch in a period of time that is not the period where the latch is supposed to be written. Note that this masking effect depends on the frequency of operation of the system.

Even though these three masking factors lead combinational logic to be less sensitive to faults than memory cells, the technology shrinking may eventually attenuate these masking effects, or even prevent them from happening in some exascale technologies. The electrical masking caused by attenuation tends to be less effective as the transistors' size diminishes. Likewise, the voltage reduction allows an increase in the frequency of operation of the system, which reduces the latching-window time, hence the temporal masking may be less effective (SHIVAKUMAR et al., 2002). Additionally, more up-to-date experiments conducted in (MAHATME et al., 2014) and (LI; DRAPER, 2016) concluded that combinational SER is even comparable to the sequential SER with particular technologies.

Despite SET effects tend to become more and more problematic, in this work we focus on SEUs since they are still the main contributors to soft errors (SHIVAKUMAR et al., 2002). More specifically, we focus on SEUs that affect the sequential logic (i.e., the flip-flops) of BOOM.

As systems are more susceptible to soft errors, fault-tolerant circuits are of increasing importance. Therefore it is necessary to understand the behavior of the system under

the presence of faults, so efficient fault-tolerance mechanisms can be devised. One of the most common and efficient techniques to estimate the fault-tolerance levels of a system is by means of fault injection, which can be defined as the intentional perturbation of the elements in a system in order to alter their logical values, while the fault propagation is observed (HSUEH; TSAI; IYER, 1997).

For low-level evaluations, fault injection techniques usually work by changing the state of latches, flip-flops or even individual transistors. For higher-level evaluations, however, it consists in disturbing the state of more abstract elements such as registers, instructions or the contents of memory addresses.

Fault injection can be traditionally categorized in hardware-, emulation-, simulation-, and software-based techniques, and each of them can be traded-off in terms of cost, injection time, precision, and representativeness.

It is important to illustrate both advantages and deficiencies imposed by hardware-based fault injection, hence evidencing the trade-offs between this method and the simulation-based ones. This way, Sections 3.2 and 3.3 briefly review hardware-based and simulation-based fault injection methods, respectively. Some examples of simulation-based fault injection methods are introduced in Section 3.4

3.2 Hardware-based Fault Injection

Hardware-based fault injection consists in injecting faults directly in the physical hardware in a pure analog way. It relies on specialized equipment to disturb the hardware components. This method is traditionally categorized in fault injection *with* and *without* physical contact (HSUEH; TSAI; IYER, 1997), as follows:

- With physical contact: There is a direct physical contact with the system and works at pin-level. It may either use probes to produce voltage or current disturbances in the pins or it can work by inserting sockets that force analog signals in the pins that perturb logic values of the state elements.
- Without physical contact: There is no direct contact with the hardware. External tools induce heavy-ion radiation into the system, or the system is placed in an electromagnetic field that causes the fault. This tactic causes spurious currents that disturb the system.

Hardware-based fault injection becomes expensive since it requires setting up the

system with dedicated equipment to perform the fault injection and to analyze the behavior of the system under the presence of faults. Also, the setup time it takes to prepare the system with such equipment should be taken into account, even though it is a one-time process (ZIADE; AYOUBI; VELAZCO,). However, it should be mentioned that after the setup process, running applications and injecting faults in an actual hardware is usually faster than doing it in a simulator that models the hardware.

Beyond the costs imposed by the specialized equipment, one must consider the costs and accessibility of the hardware itself. In that sense, the limitations are:

- The hardware is, of course, not available before it is actually implemented, therefore it is not possible to make early evaluations of the target system. Therefore, in case hardware structures are found to be fault-sensitive, it would have to be rebuilt from scratch.
- The hardware may be damaged in the fault injection process, hence becoming useless and the costs tend to increase.

Another deficiency of such technique is related to its levels of controllability, observability, and repeatability. Usually, there is no way to control where the faults are supposed to be injected with accuracy, mainly when there is no direct contact with the target system. Hence, any sort of failure cannot be repeated, nor can its behavior be easily observed. Moreover, one of the most difficult parts of such technique is related to its fault monitoring and observability because the detection of the fault in the internal structures of the processor may be really difficult and requires the usage of complex monitoring hardware (CARREIRA et al., 1998).

Even though the hardware-implemented technique has many shortcomings, it has some advantages, such as:

- There is a low-level model of the system, so the entire system is exposed to the tester, meaning that the fault coverage is usually the highest possible.
- It is not intrusive, so the design under test (DUT) does not need to be modified by any means to perform the faults.
- Experiments are usually fast after the setup process.
- It puts the equipment to work in a situation that resembles the actual environment the equipment is supposed to be used, so that the evaluation may be realistic.

Since the limitations of the hardware-based fault injection method may be a prohibitive factor, designers can appeal for RTL descriptions (i.e., Verilog or VHDL models)

of the hardware, which can be used to emulate the hardware behavior at low-level and with high controllability. This method may work as an emulation-based technique implemented on FPGAs. Unfortunately, such RTL descriptions are frequently intellectual property and hence are not publicly available. Also, this technique tends to be slower than the hardware-based one, since the system is simulated at low-level.

3.3 Simulation-based Fault Injection

On account of the restrictions imposed by the hardware- and emulation-based fault injection methods, some techniques rely on software-based simulators, mostly described in C/C++ high-level languages, that intend to mimic the behavior of the system in a more high-level perspective where the low-level structures are not modeled. Good examples of such tools are the instruction set simulators (ISS). An ISS may be defined as a layer of software that resides in a host system that enables instructions compiled to one target architecture that is different from the host architecture to execute in the host (LV et al., 2008).

An ISS can either work by means of *binary translation*, such as QEMU (BELLARD, 2005), where the target instructions are translated to the host instructions and then are executed natively, or they can work by means of *interpretation* and simulated execution of the target instructions without explicitly translating them, such as the gem5 simulator (BINKERT et al., 2011).

The main advantages of the simulation-based over the hardware-based strategy are:

- It is flexible, so it is easy to modify the hardware-models, or even the fault models, for instance.
- No need for the hardware to be physically implemented, so it is possible to early-evaluate the system's reliability with certain accuracy.
- It is highly controllable, so fault location and time can be deterministic.
- It is does not rely on RTL descriptions of the system, which may not be of public domain.

One of the worst limitations of some high-level simulation-based techniques are related to their degree of representativeness. Most of these tools only model the hardware at high-level, which brings difficulties in evaluating the hardware structures with the same

accuracy provided by RTL models.

The next sections introduce some tools related to simulation-based fault injection.

3.4 Related Work on Fault Injection Tools

Several fault injection platforms for General-purpose Processors (GPPs) and Graphic Processing Units (GPUs) have been developed on top of different simulators and using different strategies. This section briefly discusses some of them, pointing out their functionalities, benefits and deficiencies.

3.4.1 DrSEUS - A Dynamic Robust Single-Event Upset Simulator

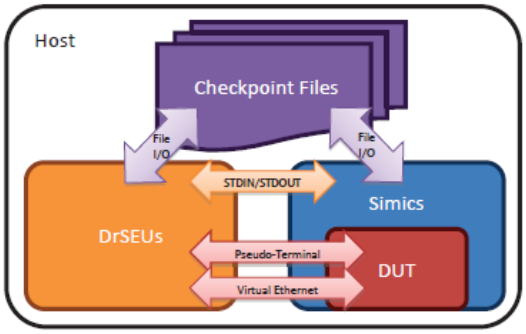
In (CARLISLE et al., 2016), the authors propose DrSEUs, a fault injector capable of simulating two types of single event effects (SEE): single event upsets (SEU) and single event functional interrupts (SEFI).

The tool works coupled with the Simics simulator, which provides functionalities to save and restore processor checkpoints (the state of the processor in a given cycle).

DrSEUs injects faults in a Freescale's PowerPC-based processor by modifying application checkpoint files and loading it into the Simics simulator. Put another way, DrSEUs does not require any modifications of the design under test (DUT) - neither the Simics simulator nor the application needs to be modified since the fault injection process consists basically of modifying checkpoint files by an external tool (the fault injector itself).

Figure 3.1 depicts architecture of the fault injector. Communication between

Figure 3.1: The DrSEUS fault injector architecture.



Source: (CARLISLE et al., 2016)

DrSEUs and Simics is done by means of the standard STDIN and STDOUT ports. Also, a pseudo-terminal that emulates serial communication is used to issue commands and to monitor the Simics execution. A virtual Ethernet port is implemented in order to send and receive files to Simics, which includes applications binaries and input/output files.

As a first step, the fault injection campaign collects golden checkpoints in a fault-free run (the state of the processor in a given cycle where no faults are injected). Then, the fault is injected by picking a copy of a randomly selected checkpoint and bit-flipping a random bit of an arbitrary register of it. After that, the mutated checkpoint is loaded into the Simics simulator, and the simulation starts execution from that checkpoint.

If the application completes execution with no errors, an output file is extracted from Simics in order to compare it with the golden equivalent file in order to detect failures.

The main disadvantage is that the Simics is a functional simulator that works at the instruction level, so it does not provide a low-level perspective of the hardware components that could be targeted in a hardware-based fault injection. Therefore fault injections have to be performed at the architectural level (usually at user-visible registers only) of the processor components.

3.4.2 OVPSim-FIM

In (ROSA et al., 2015), the authors propose the OVPSim-FIM fault injection platform aiming to evaluate the ARM Cortex-A9, Cortex-A15, and Cortex-M4F processors. The tool was implemented by coupling a fault injection library to the OVPSim simulator.

The OVPSim-FIM fault injection process can be summarized in five stages:

1. Golden execution: Applications execute in a fault-free mode. No faults are injected and the final processors state, memory and instructions count is logged.
2. Fault configuration: This phase configures the fault location and time. As the OVPSim simulator is not cycle-accurate, the number of instructions executed is used as a temporal reference, and the fault time actually tells in which instruction the fault is supposed to be injected. The fault locations are any randomly chosen register or memory address, and the fault model is based in a single bit-flip.
3. Error analysis: Compares the results of the fault campaigns with the equivalent golden execution in order to detect failures. Also, failures are classified in this

phase.

4. Error report: Reports results created during phase 4.

Since this simulator is not cycle-accurate, the dynamic instruction count was used as a temporal reference during campaign time, which is considerably coarse-grained. Also, the simulator models the processor behavior at high-level, and faults can be injected only on architectural registers and memory.

3.4.3 F-SEFI - Fine-grained Soft Error Fault Injector

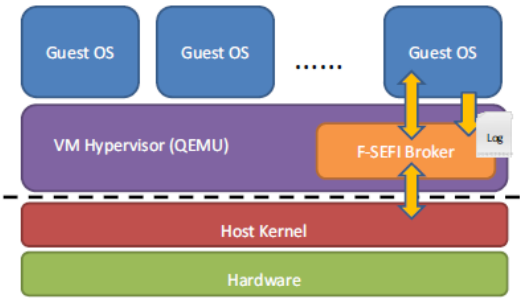
In (GUAN et al., 2014), the authors propose F-SEFI, a fault injection tool built by coupling a fault injection broker (a module) to the QEMU virtual machine. The tool performs binary injection dynamically.

QEMU works by translating instructions from the guest ISA to the host ISA by using the Tiny Code Generator (TCG). After the translation phase is complete, the translated instructions can be executed natively by the host machine. Fault injections are performed by intercepting instructions and replacing them with faulty instructions during the TCG translation.

F-SEFI can target instructions in specified functions by benefiting from a binary symbol table. Also, multiple instances of Guest OS'es can execute in parallel and in an isolated form, so that different architectures can be evaluated at the same time without interference. F-SEFI can also target a specific application, hence multiple applications running in the same Guest OS can be targeted in a controllable manner.

Figure 3.2 depicts the architecture of the platform. The F-SEFI Broker intercepts selected instructions coming from the Guest OS (the OS being emulated). Faults are injected in the intercepted translated instruction, then the instruction is executed by the

Figure 3.2: The F-SEFI fault injector architecture.



Source: (GUAN et al., 2014)

Host Kernel.

The main disadvantage of this strategy is that QEMU works at instruction-level, hence it can only target specific micro-operations to inject faults. In other words, since the fault injections are performed in the instructions (at the assembly level), it does not accurately reflect equivalent hardware fault injections.

3.4.4 MaFIN and GeFIN

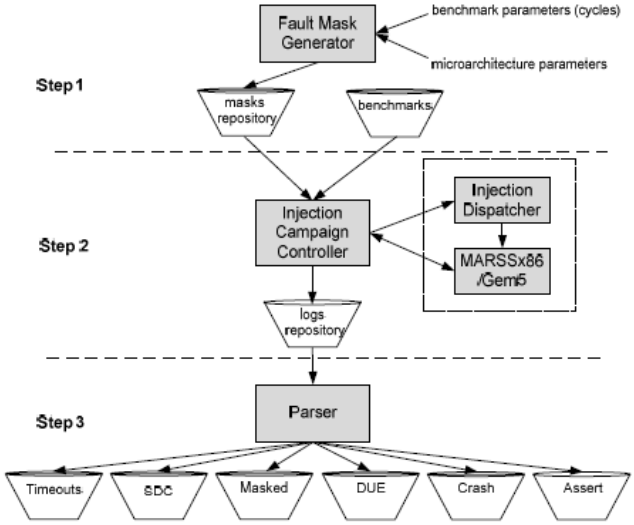
In (KALIORAKIS et al., 2015), two different fault injectors, MaFIN and GeFIN (for MARSS-based and Gem5-based Fault Injector, respectively) are proposed in order to investigate the propagation of faults in the x86 and ARM ISAs at micro-architectural level. The work aimed at studying the reliability on different ISAs and reliability studies on the same ISA on different simulators. Also, both MaFIN and GeFIN support transient, intermittent and permanent fault models.

Both MaFIN and GeFIN functionalities are depicted in Figure 3.3. The process of injecting faults consists in:

1. Generating a list of fault masks: In step 1, a list of fault descriptions is generated in the form of masks. A fault mask consists of the processor core where the fault is going to be injected, the microarchitecture structure on which the fault will be injected, the exact bit position of the injection, the exact simulation cycle or exact instruction on which injection happens (for transient or intermittent), the type of fault, and the population of faults (single or multiple).
2. Running & Injecting the fault: The *Injection Campaign Controller* reads fault masks from the *masks repository* and sends it to a *Injection Dispatcher*, which is the module that communicates directly with the MARSS and Gem5 simulators. Finally, step 2 retrieves files with the results of the fault injection process and logs them in the *logs repository*.
3. Process the results: In step 3, the *Parser* component takes the log files generated in step 2 and processes the files in order to detect and classify the faults.

Even though both MaFIN and GeFIN are micro-architectural and detailed simulators, they do not model the system at RTL (i.e., the C++ description of these simulators is not equivalent to the RTL description of the same system). That implies that these simulators do not model, for instance, all the intermediate flip-flops found in an equivalent

Figure 3.3: The MaFIN and GeFIN fault injector architectures.



Source: (KALIORAKIS et al., 2015)

RTL description of the system. Functional units, for example, are not modeled by these simulators in a way that resembles the actual hardware implementation. Also, these simulators do not model the combinational logic of the processors, meaning that the faults may not propagate through the components in the same way they would when injecting faults in a real RTL description. As a consequence, MaFIN and GeFIN can only inject faults in hardware components that are modeled as arrays, such as the register file, load/store queues, and caches.

3.4.5 GPU-Qin - A GPU Fault Injector

The methodology developed in (FANG et al., 2016) (the GPU-Qin framework) aims to evaluate the resiliency of GPGPU applications written in the NVIDIA Compute Unified Device Architecture (CUDA) and running them on an actual NVIDIA GPU.

CUDA is a standard API (application programming interface) that enables programmers to use a GPU device as a means for processing big amounts of data in parallel in a programming model known as single instruction/multiple threads (SIMT). Basically, this API allows for the programmers to transfer input data from the host CPU to the GPU, the GPU then splits a *kernel* C/C++ function into several distinct threads working on the data in parallel. Once the output is calculated, it is then transferred back to the host CPU. This model extends the usage of GPUs to general purpose applications, so it is commonly termed General Purpose Graphics Processing Unit (GPGPU).

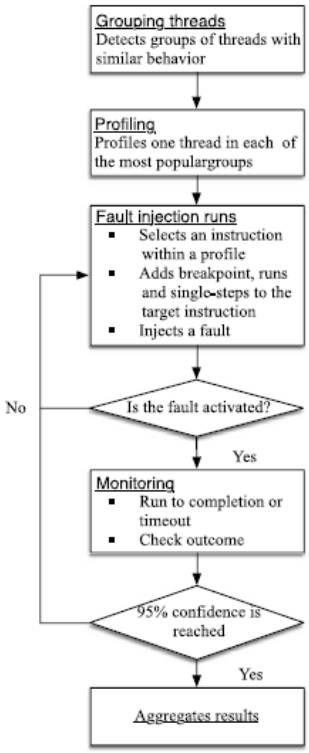
Aiming to evaluate the resiliency of some GPGPU applications, GPU-Qin injects faults at instruction level using a GPU-based debugger (cuda-gdb), which provides methods to trace and modify the application at runtime. The injection process consists in three different phases: *Grouping*, *Profiling* and *Fault injection*. Grouping and Profiling phases are performed in the GPGPU-Sim GPU simulator, but the applications execute natively (i.e., in a real GPU hardware) when faults are injected.

Figure 3.4 depicts the fault injection process, and it can be summarized as:

1. Grouping: As the number of threads in GPGPU applications may be too high (often tens of thousands), it is infeasible to obtain an execution trace for each of the possible threads. To overcome that, the method used consists in analyzing the behavior of the application and in separating similar threads in different groups. Threads are considered similar when they execute a similar number of dynamic instructions. After the threads are grouped, one thread from each group is selected to obtain the execution trace.
2. Profiling: This phase consists in mapping the dynamic instructions executed by the selected threads in the previous phase to their corresponding CUDA source code line. This is important since the fault injector relies on the cuda-gdb tool, which needs the source-code line for setting breakpoints along the program flow. The output of the profiling step is an instruction trace consisting of the program counter values and the source line associated with each instruction (FANG et al., 2016).
3. Fault injection and monitoring: In this phase, a random thread is selected to inject the fault. A conditional breakpoint is triggered when the selected thread reaches a chosen source-code line that corresponds to a randomly selected instruction obtained from the trace from the previous phase. Faults are injected at the instructions (source or destination registers; or memory addresses) when any breakpoint is triggered. This process is repeated until a significant number of faults are injected.

Since this process works at the assembly level, this technique is useful to assess the impact of faults only at the architectural state of the GPGPU. Even though the real hardware is used in the process, cuda-gdb provides access only to a subset of the GPGPU state, so it is not possible to directly target any specific component of the GPGPU.

Figure 3.4: GPU-Qin injection process.



Source: Adapted from (FANG et al., 2016)

3.5 Main Contributions of the Proposed Platform

As it was already mentioned, fault injection tools generally have intrinsic characteristics that should be considered when a system's reliability is to be evaluated. We point out six primitives that should be considered:

1. **Representativeness**: How accurately is the system modeled? Is there a real hardware? If not, how accurately is the hardware modeled?
2. **Controllability**: What is the level of controllability of the fault injection? Can one choose where/when faults should be injected?
3. **Accessibility**: Is there an available model of the system in which faults can be injected? Is there a real hardware? Is there an available simulator or an HDL description of the system?
4. **Cost**: Is there additional costs to inject faults into the system? Or is it for free?
5. **Synthesizability**: Can the system be synthesized?
6. **Performance**: How many faults can be injected per second?

The main problem statement of this work is based on the fact that there is a trade-off between hardware-based techniques and methods based on high-level simulators. For

both techniques, the items (1)-(6) are never provided altogether (ARLAT et al., 2003). However, the main goal of this work aimed to implement a fault injection tool for a superscalar processor that is controllable and cheap, while working at RTL. This aspect requires that such a tool inherits both the levels of representativity of a hardware (to some extent) and the levels of controllability of a simulator implemented in software. Moreover, as some techniques may or may not allow the physical implementation of the system, the *synthesizability* is an important aspect to be considered.

Note that, although our platform works at RTL, it is not as representative as the real hardware because it does model the electrical characteristics, voltages, features' sizes and propagation delays between gates, for example. However, since in this work we are interested only on SEUs, working with a simulator at RTL is arguably accurate and fault injections in this model may be considered *similar* to fault injection in the real hardware. By *similar*, we mean that the SEUs tend to propagate through the simulated processor in an analogous way they would propagate in the real hardware if no electrical events masked the fault. On the other hand, if the phenomenon to be investigated are MBUs, for example, then our tool would lack the physical model of the characteristics of the hardware that influence the propagations of MBUs, because these events depend on the actual distances between gates and latches and the frequency of operation, for example.

Table 3.1 compares the items listed in (1)-(6) for each of the fault injection methods considered. The symbols '+', '++', '-', and '- -' represent the different "intensities" for each of the characteristics listed in the table. For the hardware-based method, the '- -' symbol means it has really poor performance, while the '++' symbol means it is strongly representative, for example. Note how our tool meets the criteria of a really sophisticated fault injection tool, because it fits in one of the best possibilities (i.e., it is accurate, controllable, available and cheap, etc...).

Table 3.1: Comparison between hardware-based, simulation-based, and the proposed fault injection tool.

Feature	Hardware-based	High-level simulator-based	Proposed tool
Representativeness	++	-	+
Controllability	-	+	+
Accessibility	-	+	+
Cost	+	-	-
Synthesizability	+	-	+
Performance	- -	++	+

Source: Author

In order to highlight the main accomplishments of our platform, we explain Table 3.1 as follows:

- **Hardware-based:** Even though it has good representativeness, it is usually expensive due to the infrastructure necessary to inject faults and has poor performance (bad). Also, the acquisition of a hardware is often both difficult and expensive (also bad).
- **High-level simulator-based:** It is highly controllable and has the best performance among all methods (good), but the levels of representativeness may vary considerably as some simulators may be purely functional (bad), while others may model the micro-architectural level of the system, but not at RTL (so-so).
- **Proposed tool:** Models the system at RTL (good), hence its performance tends to suffer (bad). It is based on a software simulator, so it is controllable (good). It is available (good). It is for free (good).

As a final note, it is important to mention that the availability of such a platform is only possible because there is a 1:1 correspondence between the Verilog description of BOOM and its simulator, which is automatically generated by the build tools we benefit from (Chisel itself). Therefore, another notable aspect that boosts our platform one step ahead of the high-level simulators is that, as Chisel generates Verilog, the BOOM processor can actually be physically synthesized or programmed in an FPGA, while other popular high-level simulators, such as gem5, are restricted to work only in software.

4 A PLATFORM TO EVALUATE THE BOOM'S SENSITIVITY TO FAULTS

As it can be noted from the previous sections, there are several methodologies and tools developed in the area of resilience evaluation. However, the most important aspect to be learned is that there is a trade-off between the hardware- and simulation-based techniques. In short, this trade-off can be summarized as:

1. The hardware-based technique, in some cases, is not plausible mainly due to its poor accessibility, high costs and lack of controllability. But since it uses the real hardware, it does not rely on high-level methods that could provide non-realistic evaluations (i.e., it has good representativeness).
2. The techniques based on high-level simulators are not as detailed as the hardware they intend to model, hence it may provide poor representativeness. In other words, injecting faults in such simulators may not resemble fault injections in the real hardware. But it comes at low-cost and experiments are often controllable.

Besides the trade-off between (1) and (2), it is important to mention that there is a lack of open-source tools that evaluate the resilience to faults of superscalar processors at low-level. In fact, none of the tools studied over the course of this work allows for the resilience evaluation of such complex systems at RTL.

Therefore, we developed a fault injection platform that trades off the shortcomings imposed by the fault injection techniques discussed so far. The development of the tool works by leveraging the BOOM's simulator infrastructure in a way that faults can be injected in the registers of the processor.

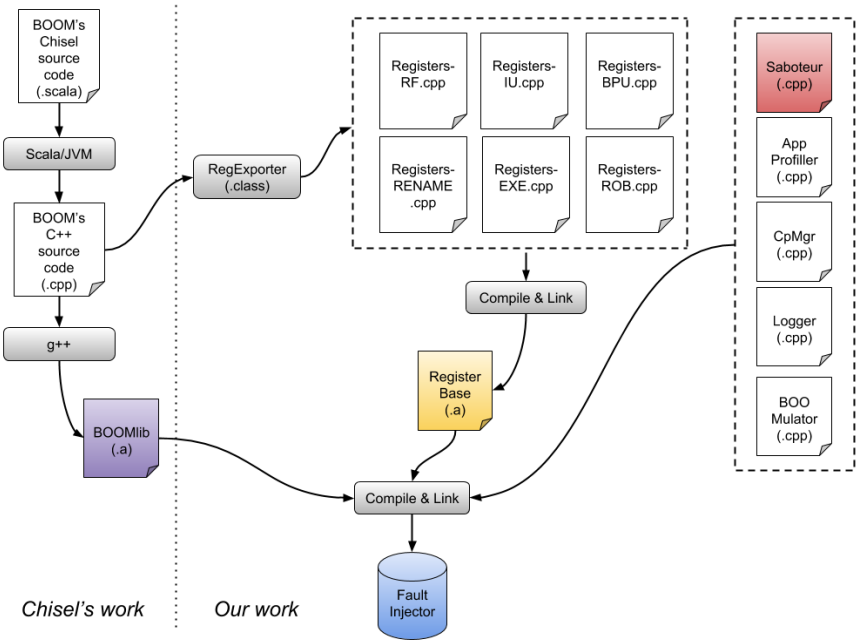
The next sections describe how the platform to evaluate the sensitivity of the BOOM processor was implemented and how the process of injecting faults is performed.

4.1 Platform Overview

The basic foundation of the proposed fault injector is the BOOM's simulator RTL infrastructure. Recall that, since BOOM is developed in Chisel, we benefit from an automatically generated C++ simulator that packs all of the processor state and behavior. The adopted approach to implement the platform was to couple a fault injection module to the BOOM's C++-exported simulator. Figure 4.1 shows the build process of the platform.

The C++ exported simulator is highly detailed, meaning that the complete proces-

Figure 4.1: Build process of the fault injection platform.



Source: Author

sor state is accessible in the simulator's internal behavior, i.e., for each bit in BOOM's Verilog representation, there is an equivalent bit in the C++ simulator. This feature is the core and fundamental idea behind our tool because it bridges the gap between the hardware and simulation-based traditional fault injection tools by providing access to the processor at RTL by means of a cycle-accurate simulator.

First, the BOOM's Chisel source code is compiled, and the equivalent C++ simulator is generated, as shown in the *Chisel's work* part on the left side of the figure. After the BOOM's C++ source code is available, an external tool (the *RegExporter*) scans this source code looking for variables that represent each of the registers of the processor (i.e., all sequential components). The registers are filtered by their respective variable names and are grouped according to their functionalities in the processor (e.g., registers in the RF or ROB). As an outcome, several other C++ source files that enable the access to any of the registers of the processor are grouped into *bundles* of registers. Once these new automatically generated source files are exported by the *RegExporter* tool, they are then compiled and linked together to form the *RegisterBase* component. This process is illustrated in the *Our work* side of the figure.

The framework can be seen as a tool composed roughly of 6 different components:

- **BOOMlib**: It is the library compiled from the C++ exported source code that models the processor's behavior in a cycle-accurate way. This library consists of the entire processor state and behavior. All of the processor's registers and behavior

will be encapsulated inside a single C++ top class termed *Top_t*, which holds the *clock_lo()* and *clock_hi()* methods for the BOOM's combinational and sequential behavior.

- **AppProfiler**: Collects fault-free information about the application, which is the first step in the fault campaign. It works by running the benchmark and storing some information such as the final virtual memory state (namely, the golden memory) and the number of cycles executed by the application (golden cycles). The golden memory is the *correct result* provided by the processor.
- **RegisterBase**: Consisted of an array of pointers to the registers of the processor. During the framework's build process, the C++ BOOM's source code is scanned by an external tool (the RegExporter), and the registers are exported from it, forming new source codes with methods to push the registers' addresses to an array. At runtime, registers are grouped according to their functionalities (see Figure 4.1). For instance, there is a method responsible for pushing all of the address of the RF's registers to the array in a proper file, while the addresses of the ROB's registers are pushed by a method in another file. At runtime, the fault injection module can target any of these registers. This component is also used to save and restore the processor state so that our tool can afford a checkpointing mechanism.
- **CheckpointingMgr**: The checkpointing manager accelerates the fault injection campaigns by fast forwarding and early stopping the application execution.
- **Saboteur**: Responsible for injecting faults in the processor. It basically consists of a structure that defines the fault model and a pointer to the *RegisterBase* component. As transient faults are supported, this module works by injecting a single bit-flip in a random cycle in any arbitrary register during the application execution.
- **Logger**: Compares the final memory state to the golden memory in order to detect failures and logs the results in a status file.

Beyond the fault injection modules, the platform was also enhanced with an instruction and performance monitor module. This component allows the analysis of the instruction trace and provides a profile of the instruction mix executed by the workload. Also, this module provides the IPC metric of the application. This is useful since it provides the characterization of the workload, which may reflect in the fault sensitivity of the hardware structures.

More implementation details for the modules will be discussed in Section 4.3, but

first, it is important to have a basic understating of the overall fault injection process, discussed in the next section.

4.2 Fault Injection Process

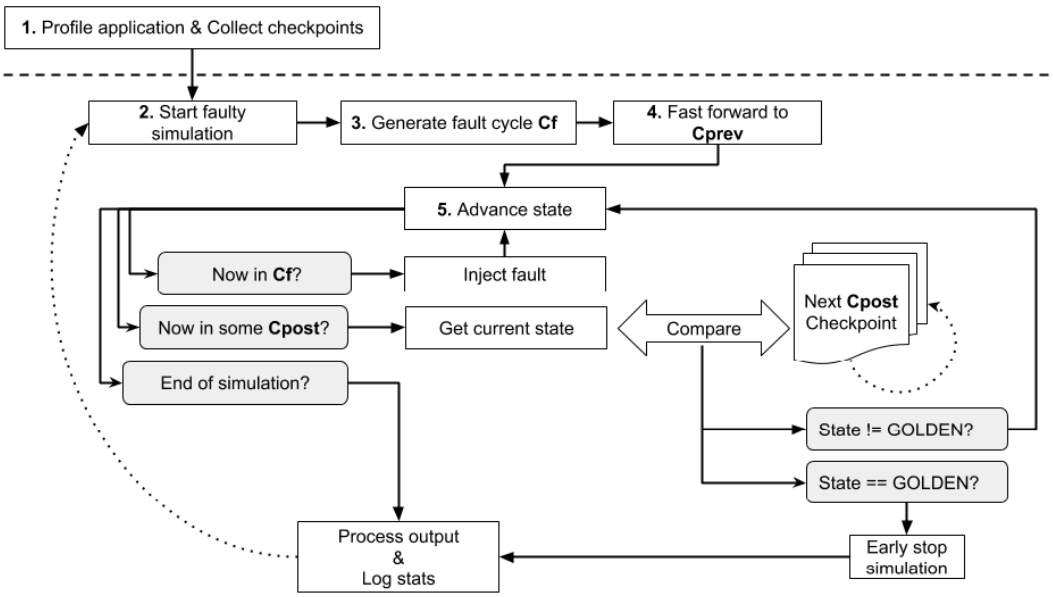
4.2.1 The Fault Injection Life Cycle

Our framework supports two different modes of operation: *fault-free* and *faulty* mode. In the *fault-free* mode, no faults are injected, and it is used to collect golden checkpoints and to profile the application. Application profiling works by collecting the contents of the main memory and the number of cycles taken by the application to be executed. That must be the first step of the fault injection campaigns, as shown in Figure 4.2. The checkpointing mechanism works by saving and restoring the state of the registers, caches, and memory considering a given interval between them. Note that the step *Profile application & Collect checkpoints* is a one-time process, hence can be skipped for future campaigns.

Once the *fault-free* mode is complete and all the necessary checkpoints are collected, the *faulty* mode starts, when the actual faults are injected in the processor.

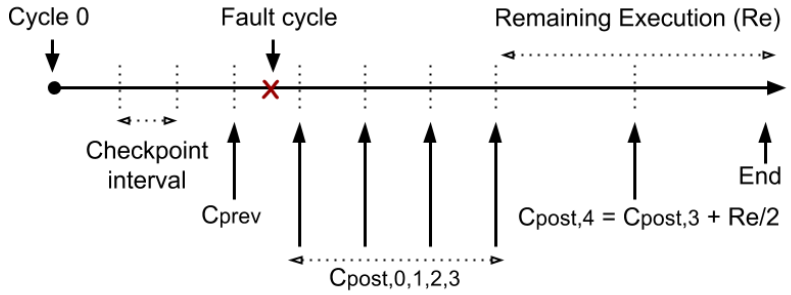
The *faulty* mode is the framework's bottleneck in terms of execution time since the simulator is supposed to run for several times in this mode, as thousands of faults have to

Figure 4.2: The fault injection life cycle.



Source: Author

Figure 4.3: The checkpointing mechanism - the application is first fast-forwarded to *Cprev*, and it *may* be halted in any *Cpost* in case the fault is masked.



Source: Author

be injected. Therefore, performance becomes of crucial importance, and the checkpointing mechanism is used to achieve better performance.

As shown in Figures 4.2 and 4.3, the checkpointing mechanism works in two distinct ways: it either fast-forwards the application and, lately, it early-stops the execution in case the fault is masked. This idea is similar to the strategies adopted in (CHATZIDIMITRIOU; GIZOPOULOS, 2016) and (CARLISLE et al., 2016). In short, this mechanism works as follow:

1. Since the processor's execution and state before the fault is injected is of no importance, it can be skipped without any loss of information. The checkpointing manager fast forwards the application execution to the nearest checkpoint available right before the fault injection cycle (the *Cprev* checkpoint).
2. The checkpointing manager early stops the application execution: After the fault is injected, it is possible that the faulty register is overwritten before it is read, preventing the fault from propagating to other hardware components. This masking effect can be detected in any post-fault checkpoint by comparing the state of the processor to its corresponding golden state. In case the fault is masked, the application can be halted since we know it will not lead to any failure. For empirical reasons, the early stop is handled by comparing a maximum of five checkpoints, which are the next four available checkpoints right after the fault cycle and, lastly, the checkpoint located right in half of the remaining application execution. These are called the *Cpost* checkpoints, depicted in Figure 4.3.

More details of the checkpointing mechanism will be given in Section 4.3.3.

4.2.2 Fault Classification

Once the simulator starts running the *faulty* mode and the fault is injected, the behavior of the simulation becomes unpredictable (e.g., it may crash). When the simulation ends execution, and if it was not early-stopped, and no timeouts or crash occurred, then the contents of the final memory state are compared with the contents of the golden memory in order to detect failures. The fault can be either considered masked, or it can manifest as a failure.

In short, the simulation outcomes may be summarized as:

- **Masked fault:** The faulty register is never used, or it is overwritten before it is read and hence the fault cannot propagate. In this scenario, the final contents of the memory are identical to the contents of the golden memory, hence no failures can be detected.
- **Silent Data Corruption (SDC):** The injected fault is not masked and propagates through the processor causing the final memory state to be different from the golden memory, and it cannot be detected during the application execution by any means, hence the term *silent*.
- **Timeout:** The faulty register causes the processor to enter into a hang state. To avoid running the application indefinitely, it stops executing when the number of executed cycles reaches twice the golden cycles.
- **Crash:** The injected fault causes *segmentation-fault* in the simulator. It happens mainly when the processor tries to access invalid/corrupted memory addresses.

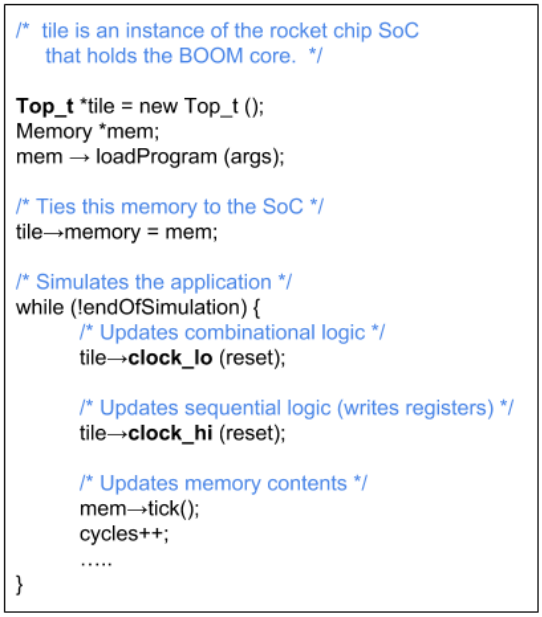
4.3 Platform Infrastructure

4.3.1 The BOOMlib Component

As explained in Section 2.3, the behavior of any C++ simulator exported from Chisel consists in instantiating an object-module of the exported class that models the hardware, then invoking the *clock_lo()* and *clock_hi()* primitives in order to update the combinational and sequential state of the system, respectively.

In that sense, the *BOOMlib* component is a library that holds the necessary code to mimic the BOOM processor. More specifically, when the Chisel description is trans-

Figure 4.4: Example of a C++ simulator exported from Chisel.



```
/*  tile is an instance of the rocket chip SoC
    that holds the BOOM core.  */

Top_t *tile = new Top_t ();
Memory *mem;
mem → loadProgram (args);

/* Ties this memory to the SoC */
tile→memory = mem;

/* Simulates the application */
while (!endOfSimulation) {
        /* Updates combinational logic */
        tile→clock_lo (reset);

        /* Updates sequential logic (writes registers) */
        tile→clock_hi (reset);

        /* Updates memory contents */
        mem→tick();
        cycles++;
        .....
}
```

Source: Author

formed into C++, the processor state will be exported to a *Top_t* class, and all of the registers of the processor will be found as objects declared inside this class along with the processor behavior that will be modeled by the *clock_lo()* and *clock_hi()* methods. In order to simulate the SoC, it is necessary to instantiate an object of the *Top_t* class, and to invoke the *clock_lo()* and*clock_hi()* methods appropriately.

Figure 4.4 shows, in a really high-level perspective, an example usage of the BOOM simulator. The basic functionality is straightforward: an instance of the SoC is instantiated, then the program is loaded to an emulated virtual memory system that is connected to the SoC and the simulation enters a loop that starts invoking the *clock_lo()* and *clock_hi()* methods to update the processor state, while the memory is ticked. For each sequence of execution of *clock_lo()* and *clock_hi()* methods, one clock cycle is advanced. For BOOM, the simulation will stop executing when a trap (an instruction) writes a proper value to a control and status register (CSR) of the processor.

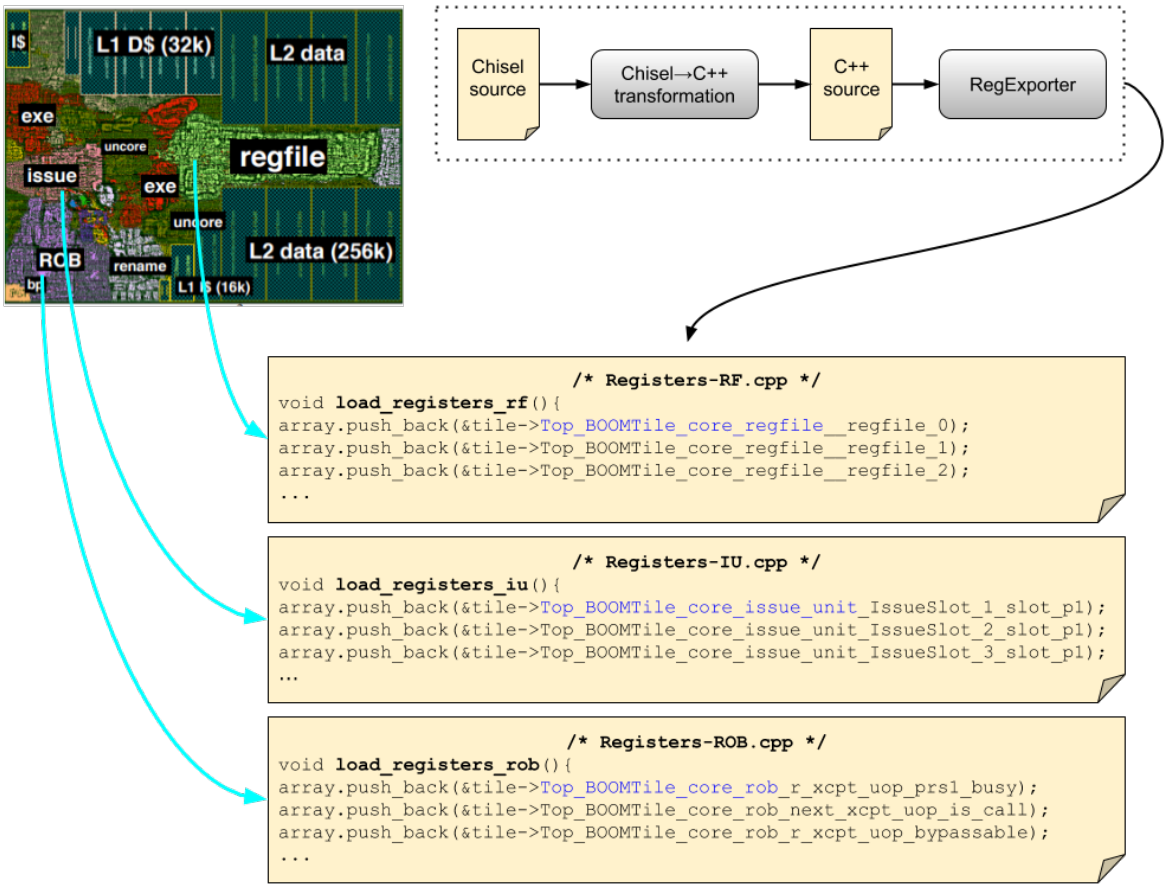
Note that the only code that is exported from Chisel to C++ is the *Top_t* class, which contains the *clock_lo()* and *clock_hi()* methods, and the registers and caches, while the code illustrated in the Figure 4.4 is handwritten, and its role is to instantiate the SoC in order to simulate it.

4.3.2 The RegisterBase Component

The *RegisterBase* component is simply a library composed of an array that holds the addresses of the processor's registers. It is constructed by the *RegExporter* external application tool, which is is a Java-based application that filters each of the registers by taking a substring of their variable names as an input filter. It then emits corresponding source codes for each group of registers, as in Figure 4.5.

As already stated, the BOOM's compilation flow consists in transforming the Chisel source code into an equivalent C++ description. After the C++ source code is generated, the *RegExporter* tool takes such source codes as an input and extracts the registers (represented as variables inside the code) from the source files. *Extraction*, in this context, means that new source codes are automatically generated by the tool. These new generated C++ source codes contain methods that push the addresses of the variables that represent each register to an array, as shown on the bottom part of Figure 4.5.

Figure 4.5: Register extraction and grouping by the *RegExporter* tool - the new generated source files form the *RegisterBase* component.



Source: Author

Fault injections and the checkpointing mechanism consist in operating on this array of registers. Whenever required, the array is loaded with the adequate register group by invoking the proper method (e.g., *load_registers_rf()* will fill the array with the RF registers' addresses). After the array is filled up, the operation on it is performed. After the operation is done, the array is freed to make room for other registers whenever needed.

An implementation-level view of the BOOM processor is depicted in Figure 4.5 (on the top left corner) in order to illustrate the functionality of the *RegExporter*. For simplicity, the figure only illustrates how the groups RF, IU, and ROB are filtered by their corresponding names. The filter is highlighted in blue as a substring of the variable's name.

Recall from section 2.2.1 that the BOOM processor is instantiated inside the *Rocket Chip* SoC generator. Also, recall from section 2.3 that the *Chisel* $\rightarrow$ *C++* code transformation preserves the order of the variable names, while organizing them in a topological manner. For example, in the C++ BOOM's source codes in Figure 4.5, the *Top_BOOMTile_core_regfile__regfile_0* variable is an object that represents a register in the processor. This name was *preserved* when it was transformed from Chisel to C++, and this object holds the actual value of the register.

The name of such variable suggests that the *Top* prefix represents the top module described in Chisel that instantiates the SoC itself. The SoC, in turn, instantiates the BOOM processor core *inside* it, represented by the *BOOMTile_core* substring. The register file is placed, obviously, inside of the core, so that the *regfile__* substring represents the instance of the whole physical register file module, and the attached *regfile_0* substring represents the register of index 0 inside the register file, which holds the actual 65-bit value of a single register in the BOOM's RF. The same analysis may be conducted for all of the registers present in BOOM.

When Chisel is transformed into C++, all of the variables/objects that represent registers are encapsulated inside a single class, and an instance of this class represents the main module of the described hardware. Inside of this class, each register is instantiated as an object. In this scenario, the *tile* object in Figure 4.5 is an instance of the main class that instantiates the SoC, which includes the BOOM processor. More precisely, the entire SoC is *instantiated* by the *tile* object, and all of the operations on the BOOM processor *must* manipulate the objects *inside* the *tile* instance.

4.3.3 The Checkpointing Manager Component

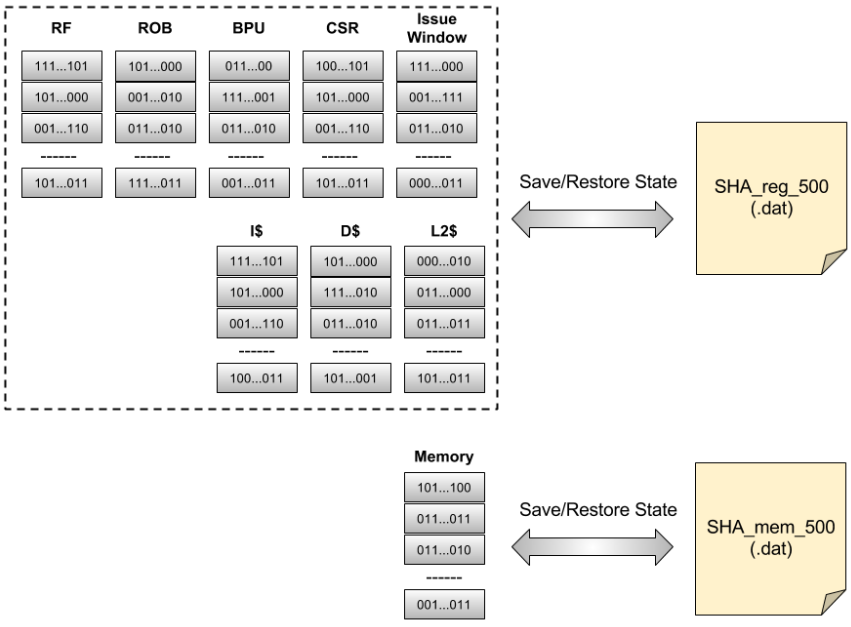
The *CheckpointingMgr* (checkpointing manager) component handles the application checkpointing by saving and restoring the contents of registers, caches, and virtual memory in a given cycle. If the context of the processor is available in any given cycle, the application execution may be latter resumed to that cycle.

For the proposed platform, checkpointing is used to speed up the fault injection campaigns by both restoring the application context to the nearest cycle *before* the fault injection cycle and by early-stopping the application, if possible, *after* the fault injection cycle in case the fault is masked.

In the BOOM simulator, each register is represented by means of a unique variable (i.e., there is one variable for each register). Recall that, for the purpose of injecting faults in the processor, all of the registers were first collected and grouped in the *RegisterBase* component according to categories (e.g., RF, ROB, EXE, IU, etc...).

The process of saving the context (or state) of the processor works simply by saving the contents of each register (also memory and caches) in the desired cycle in an appropriate file. For each array of registers (i.e, for each register group), a binary file will hold, sequentially, the contents the registers contained in it. In this architecture, a different file is required to save the state of registers (which also includes the caches) and memory,

Figure 4.6: Saving/restoring the state of the processor for the SHA application in cycle 500.



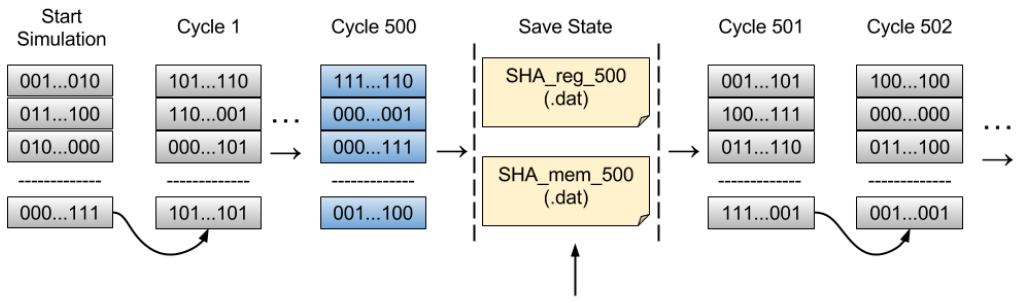
Source: Author

and for the same application, a different file is created for each desired cycle the context is supposed to be saved in. For this reason, the contents of registers and the memory are saved in files whose names are patterned as follows: *ApplicationName*_reg_*Cycle*.dat, for registers and caches; and *ApplicationName*_mem_*Cycle*.dat for the memory.

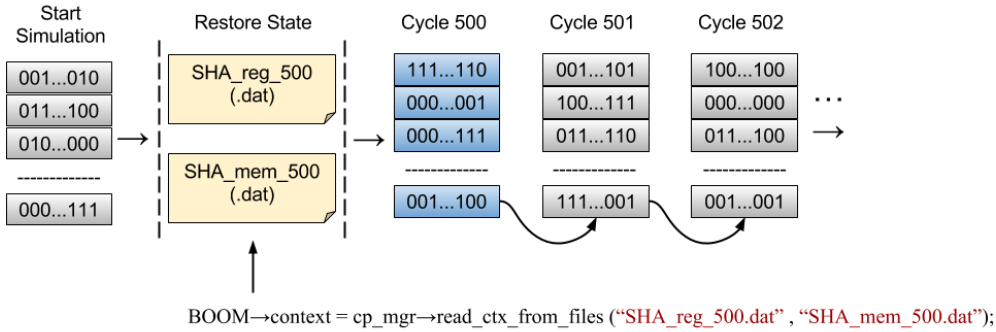
As an example, Figure 4.6 illustrates the process of saving/restoring the processor context in cycle 500 while running the SHA application. Consider that each of the vertical gray "cells" represents a single register; in our framework, the whole "pile" of cells is stored in an array of registers. Note that, for simplicity, the figure only depicts the RF, ROB, BPU, CSR and the Issue Window register groups (and also memory and caches), but the checkpointing mechanism actually saves all of the register groups.

In order to save the processor's context, for each group of registers, an array is first loaded with all of the required addresses of such registers that are pulled from the *RegisterBase* component, which also includes the caches' contents. After that, the contents of the array are written sequentially to the file. For later recovery of the context, the array has just to be read from the file in the same order it was written. As the memory of the

Figure 4.7: Saving/restoring the processor state for the SHA application.
(a) In cycle 500, the checkpoint is created by writing the processor's state to proper files.



(b) In cycle 500, the processor's state is restored by reading it from the proper files.



Source: Author

processor is simply represented by a vector, the process of saving and restoring its context is straightforward.

After processor state is saved, all of the checkpoint files will be found in a folder located in the simulation directory. More precisely, for each application, there will be two different files for each cycle the checkpoint was saved in, as shown in Figure 4.6. The application's context may be lately restored to a required cycle by reading the values of each register and memory content that are read from the associate file.

Another checkpointing example is depicted in Figure 4.7 in order to give a more consistent and generic example of how the application context is saved and restored. However, in this case, consider that the gray vectors represent the whole processor state and memory (again, just for simplicity).

Suppose, for instance, that the processor's context must be saved in cycle 500 for the SHA application. First, the program is loaded into the memory and the simulator starts at cycle 0. When the simulation reaches cycle 500, the application halts, then the *CheckpointingMgr* component saves the state of each of register, caches, and memory. After the files are saved, the application resumes the execution right from the point it stopped. This process is illustrated in Figure 4.7a.

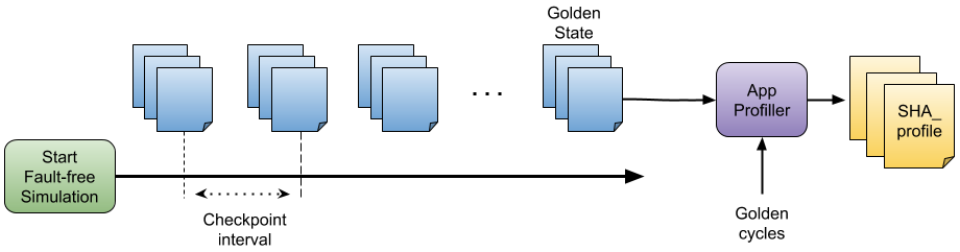
In a future execution of SHA, if the state of the processor before cycle 500 is of no interest, the state of the processor may be restored from the appropriate checkpoint files by passing the cycle of interest as an argument to the simulator. The process of restoring the context of the processor to a given cycle is called *fast-forwarding*, and it is depicted in Figure 4.7b.

The *CheckpointingMgr* provides a considerable speedup for the fault injection campaigns by restoring the processor context to the proper cycle. Moreover, as will be described in the next section, this component is also used to stop the application execution when faults injected in the processor are masked.

4.3.4 The Saboteur Component

The proposed fault injection platform aims to inject a single bit flip (an SEU) in a random bit of any arbitrary register of the BOOM processor. For that matter, the *Saboteur* component was implemented and coupled with the BOOM simulator. This component basically consists of a pointer to the *RegisterBase* component, which enables the access to all of the registers contained in it.

Figure 4.8: Generating checkpoints and profiling the application.



Source: Author

The fault injection process is simple, and it works as follows:

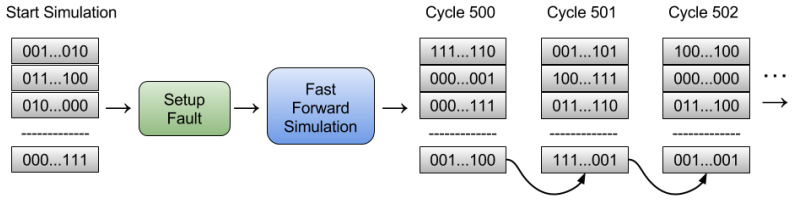
1. Generate a cycle in which the fault is supposed to be injected.
2. Randomly select a register in the processor from the *RegisterBase* component.
3. Generate a random index (*ri*) that represents the bit supposed to be flipped in that register.
4. Using the random bit index, generate a fault mask in which only the *ri*-bit is '1', and all other bits are '0'.
5. When the fault cycle is reached, operate an exclusive-or (XOR) operation between the value of the register and the fault mask.
6. Monitor the state of the processor. If the fault is masked in any *Cpost* cycle, halt the simulation.

Before any fault injection campaign starts, a *golden* run is necessary in order to collect fault-free information about the application, such as the final contents of the memory and the number of executed cycles by the application. Also, it is worth collecting golden checkpoints in a regular interval along the application flow in order to feed the checkpointing mechanism and speedup future fault injection campaigns. The *AppProfiler* component handles such profiling and it is depicted in Figure 4.8. After the profiling is complete, there will be a golden memory file for the application that will be used by the *Logger* component in order to detect failures in future fault injection campaigns.

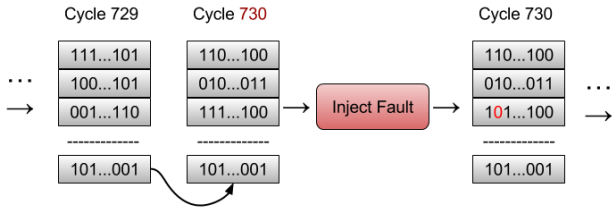
As we benefit from a checkpointing mechanism, it is convenient to generate many checkpoints for an application in a regular cycle interval. Once all the checkpoints are collected and the application profiling is complete, the faulty simulation starts. After the cycle in which the fault will be injected is generated, the state of the processor can be fast-forwarded to the nearest available checkpoint *before* the fault cycle (the *Cprev* checkpoint).

As an example, suppose that the SHA application is executing, and a fault is sup-

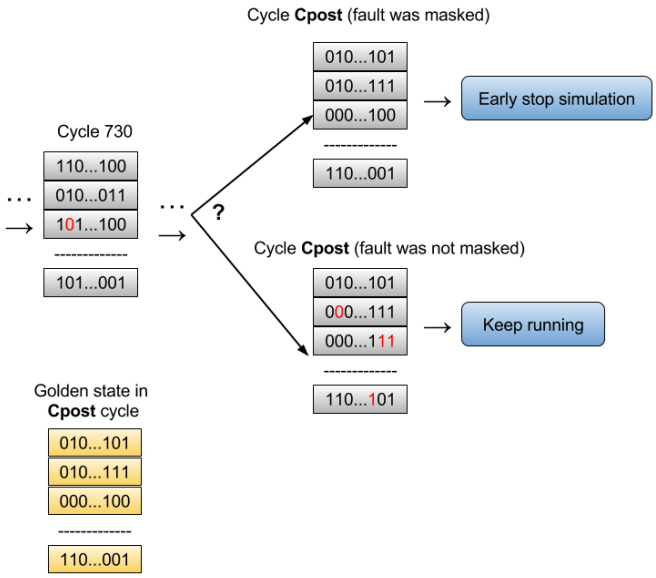
Figure 4.9: Fast-forwarding to the cycle 500 and fault injection in cycle 730.
(a) After fault configuration, the processor is fast-forwarded to cycle 500 and keeps executing.



(b) A single bit is flipped when the simulation reaches the fault cycle.



(c) In a *Cpost* checkpoint, the simulation is early stopped in case the fault is masked, otherwise it keeps running.



Source: Author

posed to be injected in the processor in a random cycle, for instance, cycle 730 (see Figure 4.9). Just in case there is an available checkpoint for this application *before* cycle 730, e.g., cycle 500, then the processor state can be fast-forwarded straight to that cycle, and the time that would be spent to execute 500 cycles is saved.

After the fault is injected, it is naturally possible that the fault becomes masked due to the program flow (i.e., application masking), and since there are checkpoints at regular intervals after the fault injection cycle, the current state of the processor may be compared to the state of such checkpoints (the *Cpost* checkpoints), which are fault-free.

By performing such comparison it is possible to detect whether the fault was masked (if the current state of the processor is identical to the checkpoint state) or not. Recall that the checkpoint files contain the entire state of the processor (registers, caches and memory), hence if the current state of the processor is identical to an equivalent fault-free checkpoint in a given cycle, the fault is *guaranteed* to be masked. Therefore, in this case, the application may be aborted since it is known beforehand the fault will not manifest as a failure, hence even more execution time is saved.

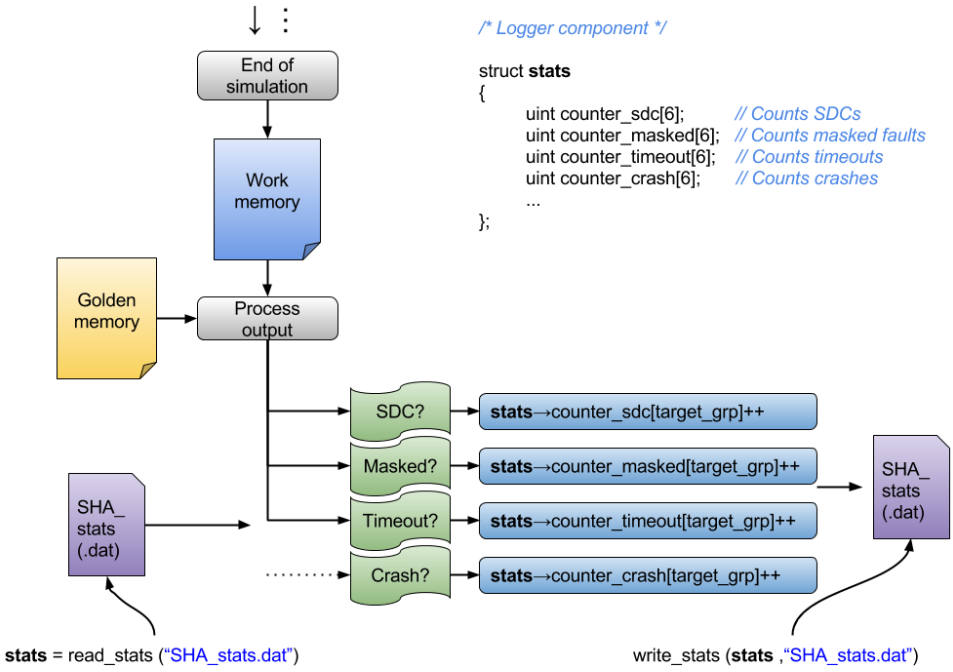
Figure 4.9 depicts the complete scenario of a single fault injection being *helped* by the checkpointing strategy. Each separate figure may be explained as follows:

- Figure 4.9a: First, the simulator starts and the program is loaded into the memory. After that, the *Fault Setup* process randomly generates the faulty cycle and the register the fault is supposed to be injected. As the generated fault cycle is 730, the application is fast forwarded to cycle 500 and then it executes the remaining 230 cycles until it reaches cycle 730.
- Figure 4.9b: When the application reaches the fault cycle (730), an SEU is injected in the randomly selected register, then it resumes the execution.
- Figure 4.9c: When any *Cpost* is reached, the state of the processor is compared to its equivalent golden state. If the fault is masked in this cycle, then the simulation stops executing and campaign time is saved. If the fault is still present, however, the simulation must keep running. Acknowledge that, in our experiments, the maximum number of *Cposts* that will be compared is five. One could consider comparing even more checkpoints in order to check if the simulation can be early-stopped, but have in mind that comparing checkpoints implies in an overhead, hence there is a trade-off between the time to compare a certain number of checkpoints and the number of early-stopped simulations.

4.3.5 The Logger Component

For each fault to be injected, there will be one instance of the simulator process. More specifically, when a fault injection is to be performed, the process of the simulator starts, then it simulates the execution of an application and injects the fault and, when the simulation ends, the *Logger* component compares the final state of the processor's memory (which we call the *Work memory*) to the golden memory state.

Figure 4.10: Workflow for the Logger component.



Source: Author

The *Logger* will then update a status file for the executed application. This file holds information about the fault injection. More specifically, the file contains vectors of counters for each of the possible outcomes of the fault injection campaign.

As several fault injections must be performed, the status files must accumulate the results for each application. Thus, after the simulation ends and the two memories are compared, a status structure is first read from the proper file to retrieve the results from previous fault injections, the structure is then updated (i.e., its counters are incremented) and the results are written back to the same file. This process is depicted in Figure 4.10.

Each position in the vector of counters is bound to a specific hardware component where faults can be injected, hence there are currently six positions (for the RF, ISSUE, RENAME, EXE , BPU, and ROB). As an example, when a fault is injected in the RF, and it leads to an SDC, then the command ***stats***→*counter_sdc[RF_ID]++* will increment the number of SDCs caused by faults injected in the RF component.

After several faults are injected, the status file of an application should be processed by a proper application in order to get the statistics about the whole fault injection campaign. Basically, it only consists of reading the *stats* structure from the binary file.

4.3.6 The BOOMulator Component

Note that we have described the individual functionalities of each module as a separate unit. In order to use the simulator, it is necessary an instance of the simulator that contains, uses and coordinates the usage of such components. This way, the *BOOMulator* component is a class that holds these different components inside of it. In order to execute the simulator, an instance of the *BOOMulator* can be declared inside of the *main()* function, for example, and after that, this object must invoke the routines that actually execute the simulation by triggering the *clock_lo()* and *clock_high()* methods.

This idea is illustrated in Figure 4.11. Note that there can be different instances of the simulator, since both faulty and fault-free simulations are supported. After the application ends a faulty execution, the *Logger* component processes the output generated by BOOM. Also, when the simulation ends a fault-free execution and the user wants to record the golden profile of the application, then the *AppProfiler* will save the BOOM's final memory state and the number of cycles taken by the application.

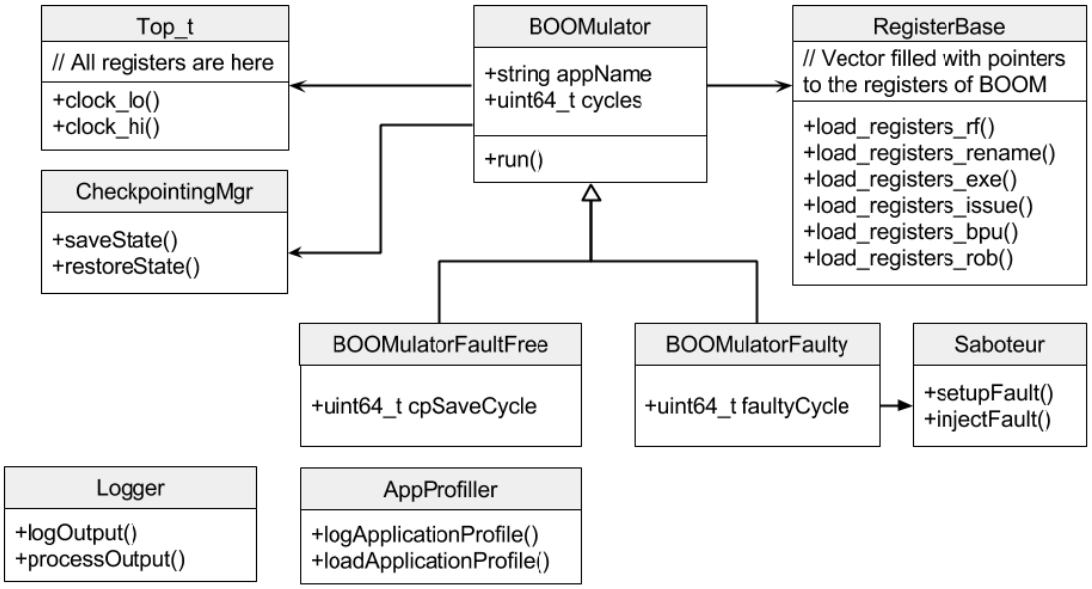
Figure 4.12 shows in a high-level perspective the infrastructure of the simulator. Have in mind, however, that the figure is not the complete UML representation of the platform at all, since its role is merely illustrative. However, some insights about the platform are given, which should be self-explanatory. Note how the BOOMulator "contains" a *Top_t* member object that represents the SoC. This way, the processor is simulated when

Figure 4.11: An example usage of the BOOMulator.

```
/*  An instance of the BOOM simulator */
BOOMulator *boom;

if (args→isFaultySimulation)  // Instantiates the faulty simulator
        boom = new BOOMulatorFaulty (args);

else    // Instantiates the fault-free simulator
        boom  = new BOOMulatorFaultFree (args);

/* Simulates the application:
run() executes the clock_lo() and clock_hi() methods and updates the tile registers*/
boom→run();

/* Logs the application profile, if that's the case */
if (args→isAppProfilerMode)
        appProfiler→logApplicationProfile (boom→workMemory , boom→cycles);

/* Process output if it's a faulty simulation */
else if (args→isFaultySimulation)
        logger→processOutput (boom→workMemory , boom→goldenMemory);
```

Source: Author

Figure 4.12: High-level view of the simulator infrastructure.



Source: Author

a BOOMulator object triggers the *run()* method, which in turn invokes the *clock_lo()* and *clock_hi()* primitives of the Chisel-exported *Top_t* class.

5 RESULTS AND SENSITIVITY ANALYSIS OF BOOM

Authors in (CZECK; SIEWIOREK, 1992) showed that the behavior of a processor when faults are injected in it depends not only on the fault location, but also on the application being executed. Therefore, resilience evaluations should be performed by executing applications commonly used by the general public. One of the most popular workloads to fulfill this requirement is MiBench (GUTHAUS et al., 2001). MiBench encompasses different application domains such as communication, industry, and security. That justifies the use of such suite in the fault injection campaigns.

In order to validate our fault injection tool, the following MiBench workloads were used:

- SHA: The secure hash algorithm takes 512-bit input blocks and produces a 160-bit *message digest* as an output. It is often used in the secure exchange of cryptographic keys and for generating digital signatures. Also, it is used in data integrity by using the message digest of the transmitted data as verification mechanism.
- CRC32: Used in error detection for data transmission. It works by calculating a 32-bit Cyclic Redundancy Check (CRC), or checksum, on the transmitted data. The checksum is calculated based on the remainder of a polynomial division of the data values, and is attached to the transmitted data before the data is transmitted.
- String search: Algorithm that searches a given string in a set of strings or text.
- FFT: Computes the Fast Fourier Transform on a sampled input signal. It delivers the different frequencies with their respective amplitudes contained in the signal. The FFT algorithm has a wide range of applications in signal processing, telecommunication, and several other fields.
- Quicksort: An efficient algorithm used for sorting an array of arbitrary types of data.
- Dijkstra: Computes the shortest path between two nodes in a graph. A graph is implemented as an adjacency matrix, then the algorithm calculates the shortest path between every pair of nodes using repeated executions of Dijkstra's algorithm. This algorithm is mostly used in computer networking, mainly in routing systems.
- Rijndael (encrypt): A symmetric cryptographic algorithm also known as Advanced Encryption Standard (AES). The encrypt algorithm takes a *plaintext* (input data to be encrypted) and a secret key as input and produces a *ciphertext* output (output

encrypted data). This algorithm is highly used in network security.

Two types of results are presented in the next sections: the speedup the checkpointing mechanism provided, presented in the next Section, and the fault sensitivity and characterization of the BOOM processor, presented in Section 5.2.

5.1 Speeding up Fault Injection Campaigns with Checkpointing

The more faults are injected in a processor, the more reliable its characterization to faults become. Speeding up application execution comes at no cost and has advantages since it necessarily implies in faster fault injection campaigns, which means that more faults can be injected in the same amount of time. Also, fault tolerance techniques such as instruction redundancy imply in a considerable overhead, so speeding up the application execution in this scenario can provide an even more significant performance improvement.

We investigated the speedup our checkpointing mechanism can yield for the single-issue version of the BOOM processor by running some MiBench applications in the following distinct cases:

1. No fast forwarding and no early stopping: the checkpointing mechanism is not used.
2. Early stopping only: It is early stopped, but not fast forwarded.
3. Fast forwarding only: It is fast forwarded, but not early stopped.
4. Fast forwarding & early stopping: The best case - the application is either fast forwarded and early stopped whenever possible.

Table 5.1 depicts the average execution time, in seconds, for the 7 benchmarks executed in the four modes of operation. For each application, 1,000 faults were injected (hence 7,000 executions) for each of the four cases, totalizing 28,000 fault injections.

Table 5.1: Faults per second (FPS) achieved, on average, and speedup for the four cases.

Fast Forward/Early Stop	Seconds Taken	FPS	Speedup
OFF/OFF	14979	0.46	N/A
OFF/ON	9688	0.72	1.54
ON/OFF	7290	0.96	2.0
ON/ON	3539	1.98	4.3

Source: Author

The time for profiling the application and saving the checkpoints is also considered in the table, but it tends to be amortized since this is a one-time process.

Column 4 shows the average speedup achieved when comparing the underlying faults per second to the case 1, where no checkpoints are used (the worst case). As can be noted, fast forwarding the application has more effect than early stopping it. That is due to the fault injection effects that sometimes prevent the application from being early stopped. When the application is both fast forwarded and early stopped, a speedup of up to 4.3 was reached. The faults per second metric depends upon the checkpoint interval (500 cycles in this case) and, of course, the performance of the machine running the fault campaigns. In this case, an Intel I7-860 with four cores @ 2.8 GHz with 16GB of memory.

5.2 Processor Sensitivity Analysis

This section presents the results obtained from the validation tests of the fault injection platform. The platform was explored and validated by injecting faults in three versions of BOOM. Also, the level of controllability of the platform was tested by selectively injecting faults in particular areas of the processors in a cycle-accurate manner.

As reliability is a fundamental concern of today's processors, it may be really important to measure/estimate which of the main components are the most sensitive to faults. In other words, it is important to characterize the processor's sensitivity to faults by analyzing its behavior when faults are injected into it. If such metrics are available during the implementation phase, for example, then protection mechanisms could be devised in order to protect the most sensitive components, thus the final product would be a more error-resilient processor by benefiting from fault detection and fault removal mechanisms.

As the validation of the fault injection platform is essential, fault injections were performed in three different versions of the BOOM processor, and the results are discussed in this chapter. The experiments presented here illustrate how fault injections can be performed in particular chosen areas of the processors. Also, this section illustrates how the processor sensitivity is influenced by some application characteristics.

As discussed in Section 2.2.2, BOOM is constituted of six major hardware structures:

1. RF: The physical register file.
2. RENAME: Register renaming circuitry.

Table 5.2: Number of flip-flops in each structure.

Flip-flops	Single-issue	Dual-issue	Quad-issue
RF	6500	7150	8320
RENAME	3285	5933	6149
IU	4897	8342	11792
EXE	20597	32492	49937
BPU	1557	1638	2257
ROB	11865	22151	56583

Source: Author

3. ISSUE: Instruction issue logic unit, including the instruction window.
4. EXE: Execution units, bypass network, and load/store queues.
5. BPU: Branch prediction unit, including branch target buffer (BTB), branch history table (BHT), and return address stack (RAS).
6. ROB: The reorder buffer.

In order to validate our platform and to estimate the sensitivity to faults of the BOOM processor, fault injections were performed in the registers that compose the hardware structures listed in (1)-(6), which also include all of the sequential logic related to these structures (i.e., it includes all of the interconnections in between these structures). As an outcome, it provided a characterization of the BOOM's sensitivity to faults.

The results provide an estimate of *which* are the most sensitive components and *how much* sensitive the components are when executing a given workload. Moreover, the sensitivities of different processors were estimated and compared. Three processor configurations were evaluated: single-, dual-, and quad-issue. Results show that the levels of sensitivity for the three processors are significantly different. Also, there is a direct correspondence between the components' sensitivities and applications characteristics.

As three different processors are evaluated, it is important to have in mind that the structure sizes for each component of the processors vary. The number of flip-flops for each item in (1)-(6) is depicted in Table 5.2. The main structures' sizes for each of the

Table 5.3: Configured sizes for the main components.

Feature sizes	Single-issue	Dual-issue	Quad-issue
Physical registers in the RF	100	110	128
ROB entries	24	48	128
Issue window entries	12	20	28
LSU entries	8	16	32

Source: Author

Table 5.4: Configured Execute Units.

Core	Execute Unit	ALU	FPU	iMul	iDiv	LSU
Single-issue	EU#0	x	x	x	x	x
Dual-issue	EU#0	x	x	x		
	EU#1	x			x	x
Quad-issue	EU#0	x				
	EU#1	x	x	x		
	EU#2	x			x	
	EU#3					x

Source: Author

experimented cores are depicted in Table 5.3. The most complex core is the quad-issue one, and it can be compared to the ARM Cortex-A15.

Recall from Section 2.2.2.5 that the BOOM processor is constituted of *Execute Units* (EU) , where different execution units operate on different data types and different types of instructions. Different *Execute Units* may be constituted of ALUs, FPUs, load and store units (LSU) and integer multipliers and dividers (iMul and iDiv). Each EU is connected to a single issue port, hence the number of EUs is correlated to the processor's issue width, so there is one EU for the single-issue, two EUs for the dual- and four EUs for the quad-issue processor. For the single-, dual-, and quad-issue processor configurations, the functional units are organized as depicted in Table 5.4.

Note that single-issue core is constituted of a single Execute Unit that provides all of the functional units supported by BOOM, while in the dual- and quad-issue cores the functional units are spread in different Execute Units (the dual-issue configuration can be more easily seen in Figure 2.6).

5.2.1 Hardware Occupancy and Sensitivity

There is a close relationship between the occupancy of a hardware structure and its sensitivity to faults. Occupancy, in this context, reflects the fraction of live bits in the component.

Regarding the relationship between sensitivity and hardware occupancy, authors in (MUKHERJEE et al., 2003) introduced the concept of architectural vulnerability factor (AVF) as the probability that a fault in a particular structure will result in an failure. The AVF is estimated by tracking the fraction of bits in a processor that is necessary for the correct execution of the workload. These bits are termed architecturally correct execution

(ACE) bits and are related to a number of live bits in a given structure of the processor.

Similarly, bits that are not necessary for the correct execution of the workload are termed un-ACE bits, and faults injected in such bits cannot lead to errors.

AVF analysis is an approach used to estimate the sensitivity of a processor (i.e., it is an alternative to fault injection). The more un-ACE bits are detected in a processor, the more error-resilient the processor is.

To a certain degree, our fault injection platform can provide an approximation to the ACE bits in the register file by tracking the number of allocated physical registers in each cycle that contains valid values assigned to them. However, it is possible to improve the functionalities of the simulator so that it would provide a more sophisticated profile of the application that executes on the processor, yielding a precise fraction of the ACE bits.

As our tool affords an approximation to the ACE bits in the RF, experiments were conducted in order to validate the tool and to estimate such metric, and hence we could correlate this metric with the sensitivity of the RF that was estimated by means of fault injection.

5.2.2 The Register File and Register Renaming Circuitry Sensitivities

As a first analysis and as a means of validation of the platform, and also as an example usage of it, we first start this section by demonstrating the influence of fault injections performed particularly in the RF and RENAME components. Note that, due to the high degree of controllability of our tool, we can freely choose *where* and *when* faults are injected at bit-level and with cycle-accurate precision.

As faults were injected in the RF and RENAME components, we will start with a quick reminder about the BOOM's register renaming circuitry; as stated in Section 2.2.2.3, the *Register Renaming* process relies on two of many hardware structures:

1. A *free list*: A list that tracks which physical registers in the register file are currently free. In other words, it tracks the registers that are not allocated, hence they do not represent any logical register.
2. A *busy table*: For instructions residing in the issue window, this table tracks the readiness status of their physical register operands by indicating whether they contain valid values (i.e., it indicates whether the operand registers have been already written).

Fortunately, the implemented platform allows the monitoring of these hardware structures (actually, everything concerning the BOOM pipeline can be monitored).

By monitoring such hardware structures, it is possible to estimate the occupancy of the register file. From the items (1) and (2), it is expected that if a register is currently allocated and contains valid values, than such a register *possibly* contains ACE bits. However, monitoring the free list and the busy table is an approximate method since it only provides the number of currently allocated registers in the physical register file, but does not provide an estimate of the fraction of ACE bits in the registers itself (i.e., it does not detect the fraction of bits in the register that are ACE). Also, note that the allocated registers containing valid values are not necessarily ACE, that is because we do not monitor whether such registers will, in fact, be used in future. These tracked registers may be used by dynamically dead or mispeculated instructions, or maybe they will not even be read by any other instruction, for example, and that would imply that the registers are actually un-ACE. Therefore, we use this metric as an approximation to the ACE registers (A-ACE) in the RF. Said again, we term the average number of currently allocated registers in the RF that hold valid values (per cycle) A-ACE registers, and this metric is taken as an approximation for the ACE registers.

We estimated the A-ACE registers for each application by monitoring the free list and the busy table, considering that only allocated registers containing valid values *may* form ACE-bits. The approximation for the ACE registers is shown in Table 5.5. As an example, for each cycle that the single-issue version of BOOM executed SHA, only about 3.3% of the registers in the RF may contain ACE bits.

All of the previously discussed applications from MiBench were used in order to estimate the sensitivity of the three versions of the processor. Recall that the BOOM pro-

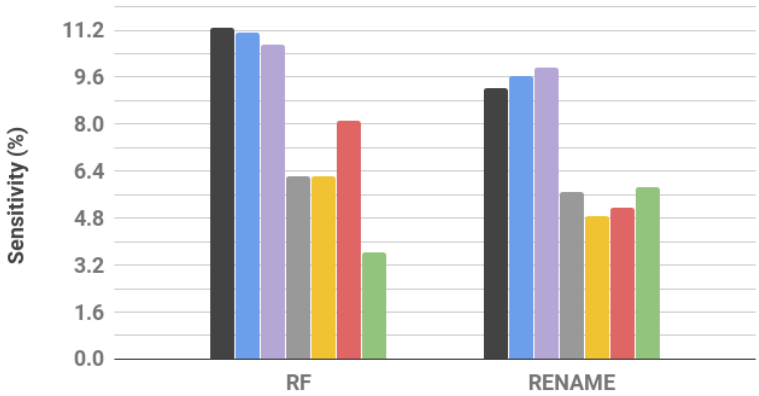
Table 5.5: An approximation to the average ACE registers, per cycle, in the RF (%).

Benchmark	Single-issue	Dual-issue	Quad-issue
SHA	3.3	5.76	12.9
Rijndael (encrypt)	2.8	5.74	7.8
FFT	2.5	3.6	4.3
CRC32	2.2	4.4	17.5
String-search	2.0	3.5	3.7
Qsort	1.4	2.37	3.1
Dijkstra	1.8	2.32	2.7
Average	2.0	4.0	7.4

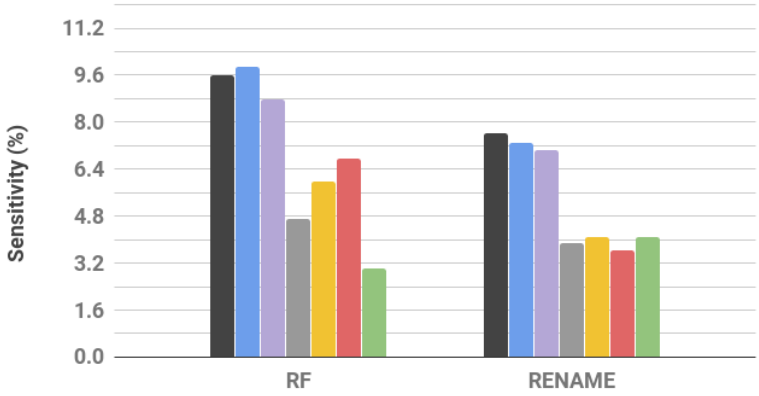
Source: Author

Figure 5.1: RF and RENAME sensitivities for the each benchmark for the three processor configurations.

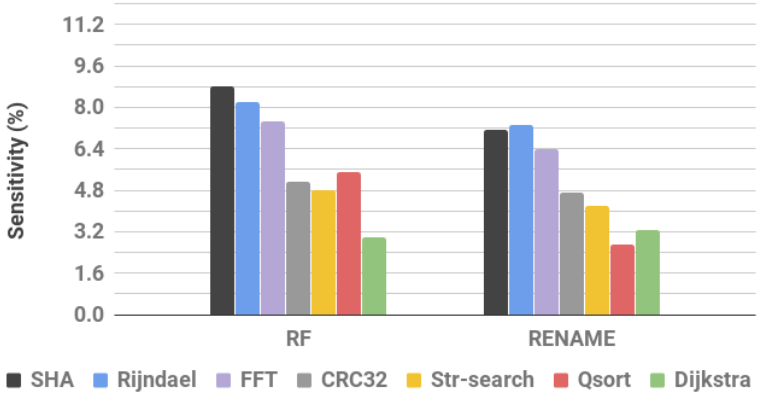
(a) Single-issue: The most sensitive one.



(b) Dual-issue: RF and RENAME are more error-resilient than the single-issue ones.



(c) Quad-issue: The RF and RENAME are the most error-resilient for this core.



Source: Author

cessor may be easily parameterized, so we could experiment with three different versions of it: 1-, 2-, and 4-issue cores were used in order to make an extensive validation of our platform.

In order to estimate the sensitivity of a structure *S*, we divided the number of detected failures due to faults injected in *S* by the number of faults injected in the structure. As an example, if *F* faults were injected in *S*, and *f* of them resulted in failures, then the sensitivity of structure *S* is estimated by the division *f/F*.

The sensitivities of the RF and RENAME components for the single-, dual-, and quad-issue are compared in Figures 5.1a, 5.1b, and 5.1c, respectively.

Recall from Section 2.2 that BOOM makes use of the register renaming strategy. At microarchitectural-level, the RENAME structure maps logical registers to physical ones in RF, which in turn updates the RENAME state during instruction commit. Therefore, faults injected in RENAME tend to propagate to RF, and vice-versa, hence both components appear to have a very similar characterization to failures, as applications that make the RF more sensitive equally make the RENAME more sensitive, as shown in Figure 5.1.

Note that there appears to be, in fact, a relationship between the fraction of A-ACE registers measured in Table 5.5 and the RF and RENAME sensitivities depicted in Figure 5.1. The RF and RENAME components are more sensitive when the processors execute SHA, Rijndael, and FFT, which is in accordance with the values in Table 5.5. The same analysis can be applied to the string-search, Qsort, and Dijkstra applications, which showed to have a smaller fraction of A-ACE registers and presented fewer failures in RF and RENAME. This analysis appears to be consistent with most of the workloads, with exception to Qsort, which makes the RF more sensitive than string-search and CRC32, so there is a slight mismatch between the sensitivity and the data shown in Table 5.5. However, note that the table only suggests an approximation to the useful/live registers.

As a final analysis, note that the average fraction of A-ACE registers residing in the register file for the single-, dual-, and quad-issue are 2.0, 4.0, and 7.4, respectively. That, supposedly, implies that the RF is more error resilient for smaller processors since it has lower occupancy. However, this is not true. Results showed that the single-issue's RF is about 17% more error prone than the dual-issue's one, and about 33% more error prone than the quad-issue's. We suggest that the explanation for that resides in the fact that the dual- and quad-issue's RF have a residency time considerably shorter than the single-issue's one because they issue more instructions per cycle, which means their RFs are

written more frequently, hence fault injections in the RF are more likely to be overwritten. An equivalent analysis may be conducted on the RENAME component. More complex processors fetch and rename more instructions per cycle, which reduces the residency time of the bits in the rename map tables and in the circuitry that handles the register renaming. This characteristic, in turn, makes the RENAME component more resilient to faults for processors that have bigger issue-widths and rename more instructions per cycle.

5.2.3 The Issue Unit and Execution Units Sensitivities

For further experiments with our platform, the fault sensitivities of the issue unit (IU) and execution units (EXE) of the three processors were evaluated.

As stated earlier, the occupancy of a hardware component reflects its sensitivity to faults. A good metric that suffices for the estimation of occupancy of the IU and EXE components is the number of instructions per cycle (IPC) the processor executes. More instructions being issued and executed implies in bigger IU and EXE occupancies, respectively. In other words, the more busier the pipeline is, the more sensitive the processor is.

Table 5.6 depicts the average IPC metrics obtained for each of the workloads for the three processor configurations. Note that the IPC depends on the processor architecture, organization and the application characteristics, as some applications may be *naturally* more parallel than others.

The fault sensitivity for the IU and EXE structures, for the single-, dual-, and quad-issue is depicted in Figures 5.2a, 5.2b, and 5.2c, respectively. The strategy adopted

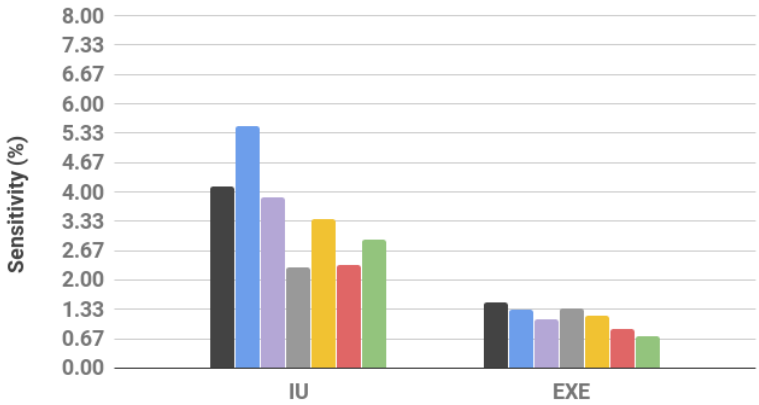
Table 5.6: IPC for each benchmark.

Benchmark	Single-issue	Dual-issue	Quad-issue	σ
SHA	0.914	1.608	2.067	0.47
CRC32	0.905	1.427	1.771	0.36
Dijkstra	0.726	1.011	1.193	0.19
String-search	0.694	0.91	0.907	0.10
Qsort	0.604	0.765	0.789	0.08
Rijndael (encrypt)	0.485	0.607	0.644	0.07
FFT	0.459	0.513	0.525	0.03

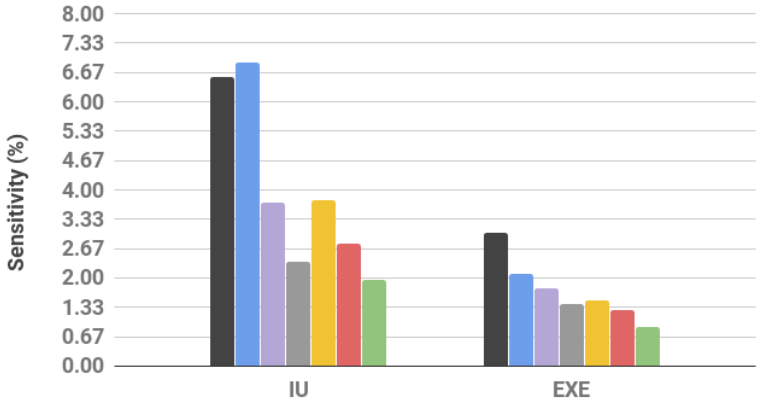
Source: Author

Figure 5.2: IU and EXE sensitivities for the each benchmark for the three processor configurations.

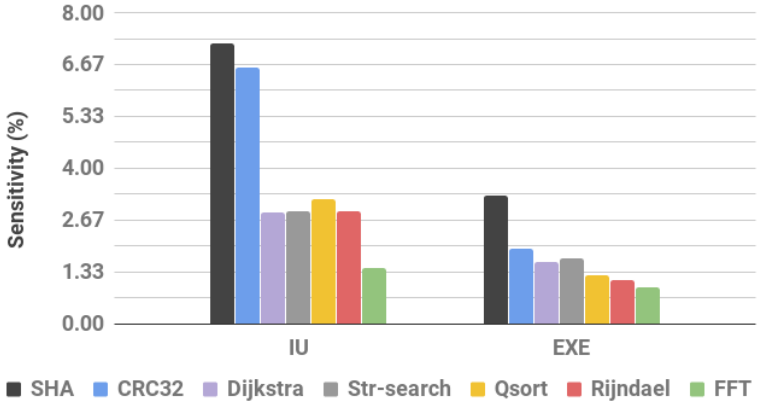
(a) Single-issue: IU and EXE are the most error-resilient ones.



(b) Dual-issue: IU and EXE are the most sensitive ones.



(c) Quad-issue: IU and EXE are slightly more resilient than the dual-issue ones.



Source: Author

to estimate such sensitivities was the same used for the RF and RENAME structures - the sensitivity is the division of the number of failures detected due to faults injected in the component by the number of faults injected in it.

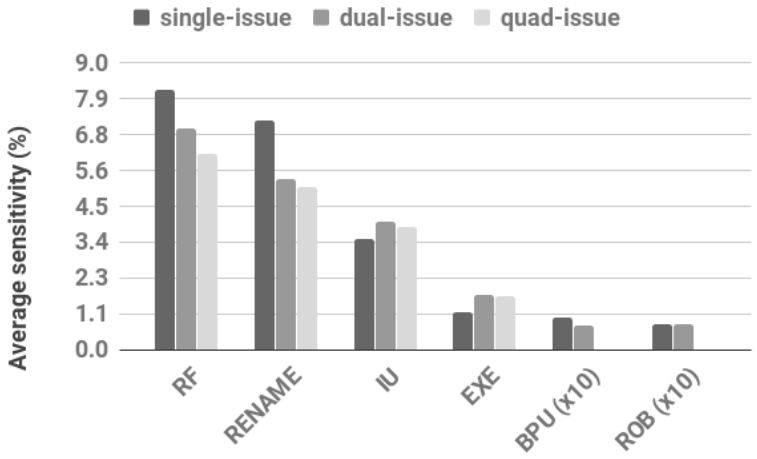
Note how the IU and EXE components are more sensitive to faults when the processor executes applications with bigger IPCs, such as SHA, CRC32, and Dijkstra. On the other hand, such components are more fault tolerant when the processor executes workloads that have lower IPCs, such as Qsort, Rijndael, and FFT.

In order to emphasize the relationship between the IPC and the fault sensitivity of the IU and EXE components, consider, for example, the IPC variations the workloads presented when executed on different processors, shown in the standard deviation (σ) column of Table 5.6. Bigger IPC variations reflect in bigger IU and EXE sensitivity variations. Note how SHA, CRC32, and Dijkstra are the benchmarks whose IPC varied the most when executed on different processors, this aspect implied that the sensitivities of IU and EXE also varied the most when executing these applications. On the other hand, applications that have a more stable IPC, such as Qsort, Rijndael and FFT caused the EXE component sensitivity to be similar for the three processors.

A particular analysis may be conducted on the EXE component when FFT executes. Note that this is the case where the EXE component is the most error resilient for the three processor architectures. Among all the applications used in the fault injection tests, FFT is the only one that makes extensive use of floating point operations (GUTHAUS et al., 2001). Also, acknowledge that, in BOOM, there is only one FPU unit regardless the issue-width (i.e., the three cores have 1 FPU). That suggests that, for the FFT, there are structural hazards that bring the IPC down and, as a consequence, the sensitivity in IU and EXE tends to be low because lower IPCs imply in lower sensitivities for these structures.

As a final remark, note that, on average, the IU is more error prone than the EXE structure. This aspect should be expected since the residency time of the bits in the instruction window is significantly longer than in EXE, because instructions must wait in the instruction window until they can be issued and executed. On the other hand, the registers in the EXE unit are written frequently, which increases the probability of a fault to be masked.

Figure 5.3: Sensitivities for all structures averaged for all the 7 benchmarks.



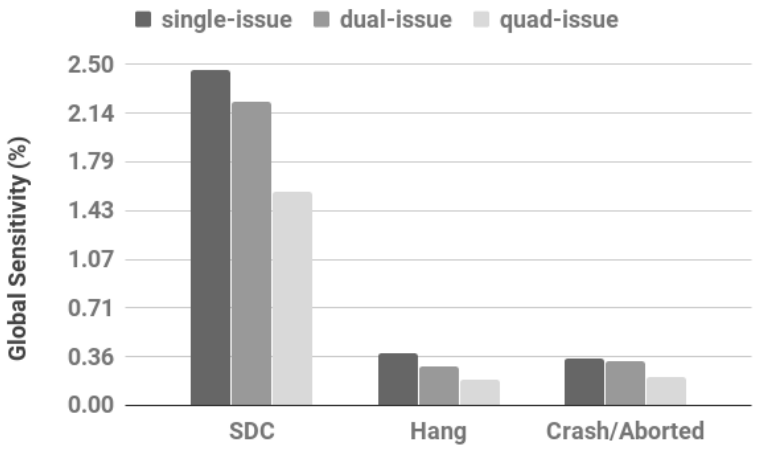
Source: Author

5.2.4 Average and Global Sensitivities of BOOM

The average sensitivity for each individual component is depicted in Figure 5.3. For each processor, this average was estimated by averaging the sensitivities for each executed benchmark. More specifically, the average sensitivity is the average of the values in Figures 5.1 and 5.2.

Figure 5.4 compares the global sensitivity estimated for each processor, averaged for all the benchmarks. In order to estimate the global sensitivity, we weighted the sensitivity of each individual component on the fraction of area occupied by them, which is approximated by number of flip-flops (bits) of the component. We adopted this strategy because the fault rate (i.e., the number of faults per unit of time) experienced by a par-

Figure 5.4: Sensitivity for each processor averaged for all the benchmarks.



Source: Author

Table 5.7: Average sensitivity for each benchmark (%).

Benchmark	Single-issue	Dual-issue	Quad-issue
SHA	4.22	4.58	3.42
Rijndael (encrypt)	3.68	2.95	1.90
FFT	3.90	2.82	1.54
CRC32	3.13	3.30	2.29
Qsort	3.98	3.09	1.51
Dijkstra	2.38	2.59	1.47
String-search	2.67	2.29	1.71

Source: Author

ticular part of the processor is proportional to its area, in which bigger structures tend to be more affected by faults, as opposed to smaller ones. The global sensitivity may be modeled by the equation bellow:

$$GlobalSensitivity = \frac{1}{TotalFF} \sum_{s=1}^{6} (Failures[s]/Faults[s]) \times FF[s]$$

In the formula above, *s* represents the six different structures where faults were injected, *TotalFF* represents the total number of flip-flops calculated by adding up the bits of the six structures, *Failures[s]* is the number of failures caused due to faults injected in structure *s*, *Faults[s]* is the number of faults injected in structure *s*, and *FF[s]* represents the number of flip-flops in the structure. Note that the division *FF[s]/TotalFF* represents the approximate fraction of area occupied by structure *s*[1]. When this fraction is multiplied by the component sensitivity, and all the factors are added together, it yields an approximation to the global sensitivity of the processor. Using this formula, the approximate global sensitivities given for the single-, dual-, and quad-issue cores are 3.18%, 2.84%, and 1.97%, respectively. That means, for example, if a fault hits the single-issue processor, then there is a 3.18% probability that a failure will occur.

Table 5.7 summarizes the data in Figure 5.4 by adding up together SDCs, hangs, and aborted simulations due to crashes for all the workloads. When the single-issue processor executes SHA, for instance, the sensitivity of the processor is about 4.22%, and it can be understood as the probability that a failure will occur if a fault is injected.

Figures 5.4 and 5.3, and Table 5.7 lead to some conclusions that can be summa-

[1] We consider the number of flip-flops as an approximation to the area of each component. However, this metric is not very accurate since we do not consider the gate-level model of the processors. More precise area information may be estimated by making use of synthesis tools.

rized as follows:

1. Figure 5.3:
    - The RF and RENAME are the most sensitive components, and their sensitivities tend to decrease for more complex processors, arguably due to the decrease in their residency times.
    - There is no single trend for the sensitivity in the IU and EXE components, which increases for the dual-issue, but decreases for the quad-issue.
    - The BPU and ROB components are highly fault-tolerant and are multiplied by a factor of ten. For the quad-issue, faults injected in the BPU and ROB never led to failures. Regarding the ROB, have in mind that, in BOOM, this component does not hold temporary values (as happens in many processor architectures, such as the Pentium 4 and the ARM Cortex A57), because the results of both speculative and non-speculative instructions always reside in the RF. We believe that this is one of the factors that makes the ROB considerably error-resilient.
2. Figure 5.4: The single- and quad-issue processors are the most and least error-prone ones mainly because of the influence of the RF and RENAME structures, which are the structures that influence the global sensitivity the most.
3. Table 5.7: The fault sensitivity does not depend only on the processor architecture and its hardware structures, it also highly depends on the application the processor executes since two different algorithms that execute on the same processor may cause the processor to respond differently when faults are injected into it.

As a final remark, note that the results exposed in this chapter were only possible due to the controllable and low-level nature of the fault injection platform. By using another kind of fault injection tool, such as the referenced ones in section 3.4, it would not be possible to inject faults in the same way it was done in this work. More specifically, our platform not only provides bit-level access to the processors but also provides all of the real bits present in them, while affording complete control over where and when faults are supposed to be injected. Note, also, that the proposed platform is highly configurable, hence flexible, so experiments with different processors could be easily be performed, yielding a vast design space to be explored.

6 CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK

This work presented a fault injection platform implemented over an RTL C++-implemented simulator of the BOOM processor.

Due to the existent shortcomings imposed by the hardware-based fault injection strategies and the methods based on high-level simulators, we found out that there is a gap between these two techniques that can only be bridged by means of the development of a tool that inherits both the positive features of the hardware-based method (representative) and the simulation-based one (controllable and cheap). The trade-off between such fault injection techniques was reviewed in Chapter 3.

The development of the fault injection platform is based on the infrastructure of an RTL controllable simulator of BOOM, which was possible due to the compilation strategies of the Chisel language which generates the low-level cycle-accurate C++ simulator automatically. Because it works at RTL and it is controllable, we believe that the deficiencies imposed by the hardware- and simulation-based techniques were traded off. Until a short time ago, and to the best of our knowledge, there was no publicly-available fault injection platform such as the one we developed and presented in this work.

The implementation of the platform was purely based on leveraging the infrastructure of the Chisel→C++ exported simulator: the strategy adopted consisted roughly in coupling separate components to the original exported BOOM's simulator that allowed us to inject faults in each individual bit of the processor. Moreover, we developed a checkpointing mechanism for the simulator that provided a really significant speedup for our fault injection campaigns. All of the strategies adopted to implement the tool is explored in Chapter 4.

As BOOM is a highly parameterizable processor, we could experiment with three different versions of this core: we generated three different fault injectors for a single-, a dual-, and a quad-issue version of the processor. The functionality of each fault injector is exactly the same except that they instantiate different processors.

In order to evaluate our platform, we used some applications from the MiBench benchmark and injected faults in the three versions of BOOM. Our fault injection tests explored in Chapter 5 were purely conducted in order to explore and validate our fault injection platform. However, future evaluations could take a more sophisticated approach by using statistical models concerning fault injection significance for each processor. Nonetheless, the evaluations of our platform provided some relevant results regarding the

sensitivity of each of the experimented processor. The experiments were efficient enough to confirm that there is a significant difference in terms of sensitivity for different processors; more complex processors tend to be more error resilient. Also, we confirmed that there is a close relationship between the characteristics of the application the processor executes and the processor's sensitivity.

For the experiments, we injected faults in six different components of the three versions of BOOM: the register file, the register renaming circuitry, the instruction window, the registers associated with the execute stage of the pipeline, the branch prediction unit and the reorder buffer. The results showed that, for each application, faults injected in each of these six different structures may lead to significantly different sensitivity levels due to the nature of the application and the influence of the component itself.

Future work may use our platform in order to evaluate the efficacy of traditional fault tolerance strategies, especially the software-implemented ones. For example, instruction redundancy techniques may be applied in a given application, then our platform could execute the application while injecting faults in the processor. One could then observe the behavior of the processor while running applications that benefit from instruction redundancy and, as a result, one could estimate how efficient such method is. Note that, since our platform is controllable, we can inject faults in any specific component and estimate what hardware components the fault tolerance system protects better. For example, if faults affect the reorder buffer, how much instruction redundancy is necessary to protect the application? At what cost?

Another example of fault tolerance technique that could be evaluated using our platform is the Triple Modular Redundancy (TMR) implemented both in software and hardware. The hardware-implemented TMR imposes severe overheads in terms area, energy and performance. This way, when such a technique is applied, it is important to know well what parts of the processor could benefit the most from such technique. For example, a TMR deployed in a hardware component that already presents affordable levels of fault tolerance may be inconvenient.

Beyond evaluating the traditional fault tolerance strategies, our platform may be used to guide the development of other sorts of low-cost fault tolerance methods. Once a new fault tolerance technique is implemented, we can then use our fault injector in order to evaluate its efficacy.

We could also use the platform in order to evaluate already existent fault sensitivity methods. For example, we could use the BOOM's simulator to conduct an AVF analysis

of the processor for specific applications, after such analysis is performed, faults could be injected in the processor and the results could be compared to the AVF analysis. Which one is faster? AVF or fault injection? Which one provides the best coverage in terms of characterization of the processor to faults?

Beyond evaluating fault tolerance systems, one could also use the platform in order to evaluate the levels of fault tolerance of different applications. In other words, some applications are better than others in terms of fault masking. One could investigate, in details, what are the characteristics that make an application more or less fault tolerant. For example, what kind of data structures are more fault tolerant? Also, does compiler optimizations have an effect on the levels of fault masking? How much?

For any evaluated fault tolerance system, one could use our platform coupled with the McPAT framework (LI et al., 2009) in order to evaluate the energy consumed by it, hence the relationship between energy spent by the fault tolerance system and fault detection and removal could be explored.

Future work could also use the BOOM's Verilog in order to extract the area of the processor and, if possible, extract the area of each individual component where faults can be injected. As a result, one could estimate more precise levels of sensitivity than the ones we presented in this work.

Finally, even though in this work we only explored the BOOM processor, future work may evaluate the resilience of other systems implemented in Chisel by adopting strategies similar to the ones adopted for BOOM.

REFERENCES

ACOSTA, R. D.; KJELSTRUP, J.; TORNG, H. C. An instruction issuing approach to enhancing performance in multiple functional unit processors. **IEEE Transactions on Computers**, C-35, n. 9, p. 815–828, Sept 1986. ISSN 0018-9340.

ARLAT, J. et al. Comparison of physical and software-implemented fault injection techniques. **IEEE Transactions on Computers**, v. 52, n. 9, p. 1115–1133, Sept 2003. ISSN 0018-9340.

ASANOVIć, K. et al. **The Rocket Chip Generator**. [S.l.], 2016. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>. Acesso em: Ago. 2017.

BACHRACH, J. et al. Chisel: Constructing hardware in a scala embedded language. In: **DAC Design Automation Conference 2012**. [S.l.: s.n.], 2012. p. 1212–1221. ISSN 0738-100X.

BAUMANN, R. Soft errors in advanced computer systems. **IEEE Design Test of Computers**, v. 22, n. 3, p. 258–266, May 2005. ISSN 0740-7475.

BELLARD, F. Qemu, a fast and portable dynamic translator. In: **Proceedings of the Annual Conference on USENIX Annual Technical Conference**. [s.n.], 2005. (ATEC '05), p. 41–41. Disponível em: <http://dl.acm.org/citation.cfm?id=1247360.1247401>. Acesso em: Ago. 2017.

BINDER, D.; SMITH, E. C.; HOLMAN, A. B. Satellite anomalies from galactic cosmic rays. **IEEE Transactions on Nuclear Science**, v. 22, n. 6, p. 2675–2680, Dec 1975. ISSN 0018-9499.

BINKERT, N. et al. The gem5 simulator. **SIGARCH Comput. Archit. News**, v. 39, n. 2, p. 1–7, ago. 2011. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/2024716.2024718>. Acesso em: Ago. 2017.

CARLISLE, E. et al. Drseus: A dynamic robust single-event upset simulator. In: **2016 IEEE Aerospace Conference**. [S.l.: s.n.], 2016. p. 1–11.

CARREIRA, J. et al. Xception: Software fault injection and monitoring in processor functional units. **Dependable Computing and Fault Tolerant Systems**, SPRINGER-VERLAG, v. 10, p. 245–266, 1998.

CELIO, C.; PATTERSON, D. A.; ASANOVIé, K. **The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor**. [S.l.], 2015. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html>. Acesso em: Ago. 2017.

CELIO, C.; PATTERSON, D. A.; ASANOVIé, K. **The Berkeley Out-of-Order Machine (BOOM) Design Specification**. 2016. Disponível em: <https://github.com/ccelio/riscv-boom-doc>.

CHANDRA, V.; AITKEN, R. Impact of technology and voltage scaling on the soft error susceptibility in nanoscale cmos. In: **2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems**. [S.l.: s.n.], 2008. p. 114–122. ISSN 1550-5774.

CHATZIDIMITRIOU, A.; GIZOPOULOS, D. Anatomy of microarchitecture-level reliability assessment: Throughput and accuracy. In: **2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.: s.n.], 2016. p. 69–78.

CZECK, E. W.; SIEWIOREK, D. P. Observations on the effects of fault manifestation as a function of workload. **IEEE Transactions on Computers**, v. 41, n. 5, p. 559–566, May 1992. ISSN 0018-9340.

DIXIT, A.; WOOD, A. The impact of new technology on soft error rates. In: **2011 International Reliability Physics Symposium**. [S.l.: s.n.], 2011. p. 5B.4.1–5B.4.7. ISSN 1541-7026.

DODD, P. E. et al. Current and future challenges in radiation effects on cmos electronics. **IEEE Transactions on Nuclear Science**, v. 57, n. 4, p. 1747–1763, Aug 2010. ISSN 0018-9499.

FANG, B. et al. A systematic methodology for evaluating the error resilience of gpgpu applications. **IEEE Transactions on Parallel and Distributed Systems**, v. 27, n. 12, p. 3397–3411, Dec 2016. ISSN 1045-9219.

GUAN, Q. et al. F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability. In: **2014 IEEE 28th International Parallel and Distributed Processing Symposium**. [S.l.: s.n.], 2014. p. 1245–1254. ISSN 1530-2075.

GUTHAUS, M. R. et al. Mibench: A free, commercially representative embedded benchmark suite. In: **Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)**. [S.l.: s.n.], 2001. p. 3–14.

HSUEH, M.-C.; TSAI, T. K.; IYER, R. K. Fault injection techniques and tools. **Computer**, v. 30, n. 4, p. 75–82, Apr 1997. ISSN 0018-9162.

KALIORAKIS, M. et al. Differential fault injection on microarchitectural simulators. In: **2015 IEEE International Symposium on Workload Characterization**. [S.l.: s.n.], 2015. p. 172–182.

KARNIK, T.; HAZUCHA, P. Characterization of soft errors caused by single event upsets in cmos processes. **IEEE Transactions on Dependable and Secure Computing**, v. 1, n. 2, p. 128–143, April 2004. ISSN 1545-5971.

KESSLER, R. E. The alpha 21264 microprocessor. **IEEE Micro**, v. 19, n. 2, p. 24–36, Mar 1999. ISSN 0272-1732.

LI, J.; DRAPER, J. Joint soft-error-rate (ser) estimation for combinational logic and sequential elements. In: **2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.: s.n.], 2016. p. 737–742.

LI, S. et al. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In: **2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2009. p. 469–480. ISSN 1072-4451.

LV, M. et al. Armiss: An instruction set simulator for the arm architecture. In: **2008 International Conference on Embedded Software and Systems**. [S.l.: s.n.], 2008. p. 548–555.

MAHATME, N. N. et al. Impact of technology scaling on the combinational logic soft error rate. In: **2014 IEEE International Reliability Physics Symposium**. [S.l.: s.n.], 2014. p. 5F.2.1–5F.2.6. ISSN 1541-7026.

MAY, T. C.; WOODS, M. H. Alpha-particle-induced soft errors in dynamic memories. **IEEE Transactions on Electron Devices**, v. 26, n. 1, p. 2–9, Jan 1979. ISSN 0018-9383.

MUKHERJEE, S. S. et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: **Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.** [S.l.: s.n.], 2003. p. 29–40.

PALACHARLA, S.; JOUPPI, N. P.; SMITH, J. E. Complexity-effective superscalar processors. **SIGARCH Comput. Archit. News**, v. 25, n. 2, p. 206–218, maio 1997. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/384286.264201>. Acesso em: Ago. 2017.

PARULKAR, I. et al. **OpenSPARC: An Open Platform for Hardware Reliability Experimentation**. 2008.

REED, R. A. et al. Heavy ion and proton-induced single event multiple upset. **IEEE Transactions on Nuclear Science**, v. 44, n. 6, p. 2224–2229, Dec 1997. ISSN 0018-9499.

ROSA, F. et al. A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability. In: **2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)**. [S.l.: s.n.], 2015. p. 211–214. ISSN 1550-5774.

SHIVAKUMAR, P. et al. Modeling the effect of technology trends on the soft error rate of combinational logic. In: **Proceedings International Conference on Dependable Systems and Networks**. [S.l.: s.n.], 2002. p. 389–398.

SMITH, J. E.; SOHI, G. S. The microarchitecture of superscalar processors. **Proceedings of the IEEE**, v. 83, n. 12, p. 1609–1624, Dec 1995. ISSN 0018-9219.

WATERMAN, A. **Design of the RISC-V Instruction Set Architecture**. Tese (Doutorado) — EECS Department, University of California, Berkeley, Jan 2016. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html>. Acesso em: Ago. 2017.

WATERMAN, A. et al. **The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0**. [S.l.], 2014. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html>. Acesso em: Ago. 2017.

YEAGER, K. C. The mips r10000 superscalar microprocessor. **IEEE Micro**, v. 16, n. 2, p. 28–41, Apr 1996. ISSN 0272-1732.

ZIADE, H.; AYOUBI, R.; VELAZCO, R. A survey on fault injection techniques. Disponível em: <https://hal.archives-ouvertes.fr/hal-00105562>. Acesso em: Ago. 2017.

APPENDIX A — SENSITIVITY TABLES

Table A.1: Sensitivities for the single-issue core (%).

Benchmark	RF	RENAME	IU	EXE	BPU	ROB	Total
SHA	1.94	0.80	0.53	0.81	0.04	0.09	4.22
CRC32	1.07	0.49	0.71	0.73	0.04	0.09	3.13
String-search	1.07	0.43	0.30	0.74	0.04	0.09	2.67
FFT	1.84	0.86	0.38	0.40	0.05	0.38	3.90
Qsort	1.39	0.45	0.44	0.64	0.05	1.01	3.98
Dijkstra	0.62	0.51	0.50	0.61	0.04	0.09	2.38
Rijndael (enc)	1.91	0.84	0.30	0.49	0.04	0.09	3.68
Average	1.41	0.63	0.45	0.63	0.04	0.03	3.18

Source: Author

Table A.2: Sensitivities for the dual-issue core (%).

Benchmark	RF	RENAME	IU	EXE	BPU	ROB	Total
SHA	1.12	0.74	0.89	1.61	0.03	0.18	4.58
CRC32	0.55	0.38	0.94	1.12	0.02	0.29	3.30
String-search	0.70	0.40	0.33	0.74	0.02	0.11	2.29
FFT	1.03	0.68	0.27	0.47	0.02	0.36	2.82
Qsort	0.79	0.35	0.52	0.79	0.02	0.62	3.09
Dijkstra	0.36	0.40	0.51	0.94	0.02	0.36	2.59
Rijndael (enc)	1.16	0.71	0.38	0.68	0.02	0.00	2.95
Average	0.81	0.52	0.55	0.90	0.02	0.03	2.84

Source: Author

Table A.3: Sensitivities for the quad-issue core (%).

Benchmark	RF	RENAME	IU	EXE	BPU	ROB	Total
SHA	0.68	0.41	0.79	1.54	0.00	0.00	3.42
CRC32	0.40	0.27	0.73	0.89	0.00	0.00	2.29
String-search	0.37	0.24	0.32	0.78	0.00	0.00	1.71
FFT	0.58	0.37	0.16	0.44	0.00	0.00	1.54
Qsort	0.43	0.16	0.35	0.57	0.00	0.00	1.51
Dijkstra	0.23	0.19	0.31	0.74	0.00	0.00	1.47
Rijndael (enc)	0.64	0.42	0.32	0.53	0.00	0.00	1.90
Average	0.48	0.29	0.42	0.78	0.00	0.00	1.97

Source: Author