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

ANDERSON LUIZ SARTOR

Adaptive and Polymorphic VLIW Processor to Dynamically Balance Performance, Energy Consumption, and Fault Tolerance

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

Advisor: Prof. Dr. Antonio Carlos Schneider Beck

Porto Alegre
March 2018

CIP — CATALOGING-IN-PUBLICATION

Sartor, Anderson Luiz

Adaptive and Polymorphic VLIW Processor to Dynamically Balance Performance, Energy Consumption, and Fault Tolerance / Anderson Luiz Sartor. – Porto Alegre: PPGC da UFRGS, 2018.

167 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Advisor: Antonio Carlos Schneider Beck.

1. Adaptive processor. 2. Fault tolerance. 3. Energy consumption. 4. Performance. 5. VLIW. I. Beck, Antonio Carlos Schneider. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof[a]. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof. João Luiz Dihl Comba
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ABSTRACT

Performance is no longer the only optimization goal when designing a new processor. Reducing energy consumption is also mandatory: while most of the embedded devices are heavily dependent on battery power, General-Purpose Processors (GPPs) are being pulled back by the limits of Thermal Design Power (TDP). Moreover, due to technology scaling, soft error rate (i.e., transient faults) has been increasing in modern processors, which affects the reliability of both space and ground-level systems. In addition, most traditional homogeneous and heterogeneous processors have a fixed design, which limits its runtime adaptability. Therefore, they are not able to cope with the changing application behavior when one considers the axes of fault tolerance, performance, and energy consumption altogether.

In this context, we propose two processor designs that are able to trade-off these three axes according to the application at hand and system requirements. Both designs rely on an instruction duplication with rollback mechanism that can detect and correct errors and a power gating module to reduce the energy consumption of the functional units. The former design, called *adaptive processor*, uses thresholds defined at design time to allow runtime adaptation of the application's execution and controls the application's Instruction-Level Parallelism (ILP) to create more slots for duplication or power gating. The latter design (*polymorphic processor*) takes the former one step further by dynamically reconfiguring the hardware and evaluating different processor configurations for each application, and it also exploits the available pipelanes to maximize the number of applications that are executed concurrently.

For the adaptive processor using an energy-oriented configuration, it is possible, on average, to reduce energy consumption by 37.2% with an overhead of only 8.2% in performance, while maintaining low levels of failure rate, when compared to a fault-tolerant design. For the polymorphic processor, results show that the dynamic reconfiguration of the processor is able to efficiently match the hardware to the behavior of the application, according to the requirements of the designer, achieving 94.88% of the result of an oracle processor when the trade-off between the three axes is considered. On the other hand, the best static configuration only achieves 28.24% of the oracle's result.

Keywords: Adaptive processor. fault tolerance. energy consumption. performance. VLIW.

Processador VLIW adaptativo e polimórfico para equilibrar de forma dinâmica o desempenho, o consumo de energia e a tolerância a falhas

RESUMO

Ao se projetar um novo processador, o desempenho não é mais o único objetivo de otimização. Reduzir o consumo de energia também é essencial, pois, enquanto a maior parte dos dispositivos embarcados depende fortemente de bateria, os processadores de propósito geral (GPPs) são restringidos pelos limites da energia térmica de projeto (TDP – thermal design power). Além disso, devido à evolução da tecnologia, a taxa de falhas transientes tem aumentado nos processadores modernos, o que afeta a confiabilidade de sistemas tanto no espaço quanto no nível do mar. Adicionalmente, a maioria dos processadores homogêneos e heterogêneos tem um design fixo, o que limita a adaptação em tempo de execução. Nesse cenário, nós propomos dois designs de processadores que são capazes de realizar o trade-off entre esses eixos de acordo com a aplicação alvo e os requisitos do sistema. Ambos designs baseiam-se em um mecanismo de duplicação de instruções com rollback que detecta e corrige falhas, um módulo de power gating para reduzir o consumo de energia das unidades funcionais. O primeiro é chamado de *processador adaptativo* e usa thresholds, definidos em tempo de projeto, para adaptar a execução da aplicação. Adicionalmente, ele controla o ILP da aplicação para criar mais oportunidade de duplicação e de power gating. O segundo design é chamado *processador polimórfico* e ele avalia (em tempo de execução) a melhor configuração de hardware a ser usada para cada aplicação. Ele também explora o hardware disponível para maximizar o número de aplicações que são executadas em paralelo. Para a versão adaptativa usando uma configuração orientada a otimização de energia, é possível, em média, economizar 37,2% de energia com um overhead de apenas 8,2% em performance, mantendo baixos níveis de defeito, quando comparado a um design tolerante a falhas. Para a versão polimórfica, os resultados mostram que a reconfiguração dinâmica do processador é capaz de adaptar eficientemente o hardware ao comportamento da aplicação, de acordo com os requisitos especificados pelo designer, chegando a 94.88% do resultado de um processador oráculo quando o trade-off entre os três eixos é considerado. Por outro lado, a melhor configuração estática apenas atinge 28.24% do resultado do oráculo.

Palavras-chave: processador adaptativo, tolerância a falhas, consumo de energia, desempenho, VLIW.

LIST OF ABBREVIATIONS AND ACRONYMS

ABFT Algorithm-Based Fault Tolerance.

ACE Architecturally Correct Execution.

ADPCM Adaptive Differential Pulse-Code Modulation.

AFRV Acceptable Failure Rate Variation.

ALU Arithmetic Logic Unit.

ASIC Application-Specific Integrated Circuit.

AVF Architectural Vulnerability Factor.

BB Basic Block.

BBHT Basic Block History Table.

CFG Control Flow Graph.

CMOS Complementary Metal–Oxide–Semiconductor.

CPU Central Processing Unit.

DCT Discrete Cosine Transform.

DFT Discrete Fourier Transform.

DMR Dual Modular Redundancy.

DVFS Dynamic Voltage and Frequency Scaling.

DVS Dynamic Voltage Scaling.

DWC Duplication With Comparison.

ECC Error Correction Code.

EDFP Energy-Delay-Failure Product.

EDP Energy-Delay Product.

FIFO First-In-First-Out.

FPGA Field-Programmable Gate Array.

FU Functional Unit.

GCC GNU Compiler Collection.

GPP General-Purpose Processor.

HPC High-Performance Computing.

ILP Instruction-Level Parallelism.

IPC Instructions Per Cycle.

ISA Instruction Set Architecture.

LANSCE Los Alamos Neutron Science Center.

LUDCMP LU Decomposition.

LUT Look-Up Table.

MCCP Multiple Clustered Core Processor.

MITF Mean Instructions to Failure.

MPI Message Passing Interface.

MWPUETF Mean Work Per Unit of Energy to Failure.

MWTF Mean Work to Failure.

NASA National Aeronautics and Space Administration.

NMR N-modular Redundancy.

NOP No-Operation.

OoO Out-of-Order.

OpenMP Open Multi Processing.

PC Program Counter.

PG Power Gating.

PID Process ID.

PPI Parallel Programming Interface.

Pthreads POSIX threads.

RAW Read After Write.

RMT Redundant Multithreading.

ROB Re-order Buffer.

RTL Register-Transfer Level.

SEE Single-event effect.

SET Single-Event Transient.

SEU Single-Event Upset.

SRAM Static Random Access Memory.

TCL Tool Command Language.

TDP Thermal Design Power.

TID Total Ionizing Dose.

TLP Thread-Level Paralellism.

TMR Triple Modular Redundancy.

TSH Tricriteria Scheduling Heuristics.

VHDL VHSIC Hardware Description Language.

VLIW Very Long Instruction Word.

LIST OF FIGURES

Figure 1.1 High-level Overview of Common Optimization Techniques 20

Figure 2.1 Hardware-based Replication with Rollback for Superscalar Processors.......25
Figure 2.2 Triplication of the Combinational Logic with a Stand-by Module 26
Figure 2.3 Traditional Out-of-order Compared to the DIVA Approach 28
Figure 2.4 Aaron Scheduling Example 30
Figure 2.5 Power Gating Header Transistor..33
Figure 2.6 Key Intervals for the Power Gating Technique...34
Figure 2.7 Methodology Flow of the Compiler-directed Power Gating 36
Figure 2.8 Example of Control Flow Graph for a Given Functional Unit 36
Figure 2.9 Multithreaded Application Behavior Regarding Communication.................40
Figure 2.10 Evaluated Cluster Configurations 43

Figure 3.1 Fault Tolerance Implementation of the ρ-VEX Processor............................48
Figure 3.2 Full Duplication Configuration for the 4-issue ρ-VEX Processor................49
Figure 3.3 Rollback Mechanism 49
Figure 3.4 Instruction Memory Access and Instruction Duplication.............................51
Figure 3.5 Data Memory Access and Instruction Duplication..52
Figure 3.6 Code Execution Example 52
Figure 3.7 Temporal and Spatial Duplication Execution Example................................54
Figure 3.8 Temporal and Spatial Duplication Core Overview.......................................54
Figure 3.9 Basic Block Execution Example 56
Figure 3.10 Spatial Duplication with PG and ILP Control 58

Figure 4.1 Static ρ-VEX organization (4-issue)..60
Figure 4.2 Interconnection Between Register Contexts and Processor Pipelanes Controlled by the Reconfiguration Controller...61
Figure 4.3 Dynamic ρ-VEX Schematic 61
Figure 4.4 Example of Bundle that Cannot be Split Using the Generic Binary Scheme 63
Figure 4.5 Instruction Split Example 64
Figure 4.6 Dynamic Issue-width Adaptation 65
Figure 4.7 Adaptive Processor Overview...66
Figure 4.8 Spatial Duplication with ILP Control...67
Figure 4.9 Adaptive Processor's Execution Flow 68
Figure 4.10 Overview of the Processor with Fault Tolerance, Energy Optimization, and Performance Management Mechanisms 69
Figure 4.11 Polymorphic Processor Overview 71
Figure 4.12 Application Scheduling Example 76

Figure 5.1 Fault Injection Flow...84
Figure 5.2 Failure Rate Behavior as more Faults are Injected (Unprotected Processor) 87

Figure 6.1 Performance Degradation when Varying the ILP Reduction Threshold 99
Figure 6.2 Spatial Duplication with ILP Control Normalized to the Unprotected Version 100
Figure 6.3 Dynamic Threshold Adaptation..102
Figure 6.4 Performance Improvement and Failure Rate Variation for the Dynamic Threshold Approach when Compared to the Threshold=1 103

Figure 6.5 Performance, Energy Overhead, Failure Rate, and Energy-Delay-Failure Product (EDFP) Comparison 108
Figure 6.6 MWPUETF Comparison 111
Figure 6.7 Performance Overhead Comparison 115
Figure 6.8 Relative Energy Consumption Comparison 117
Figure 6.9 MWPUETF Comparison - Polymorphic Processor 120
Figure 6.10 System Performance and Energy Consumption Comparison 126

Figure A.1 Issue Utilization and Phase-configurable Duplication 150
Figure A.2 Phase-configurable Duplication for the *ADPCM* Benchmark 151

LIST OF TABLES

Table 2.1 Core Parameters 42

Table 6.1 Accuracy Comparison between RTL and Gate-level Fault Injection....94
Table 6.2 Number of Cycles and Simulation Time Comparison 95
Table 6.3 Number of Transitions and Switching Activity from Gate-level Simulation..97
Table 6.4 Failure Rate and Performance Degradation Comparison....98
Table 6.5 Area and Power Dissipation Comparison 103
Table 6.6 Fault Tolerance Techniques Comparison 104
Table 6.7 Evaluated Designs and Their Techniques for Fault Tolerance and PG on the Static Processor 105
Table 6.8 Performance, Energy Consumption, and Fault Tolerance Comparison 106
Table 6.9 Best Configuration for Each Benchmark 113
Table 6.10 Duplication Ratio - 2-issue Processor 113
Table 6.11 Duplication Ratio - 4-issue Processor 114
Table 6.12 Duplication Ratio - 8-issue Processor 114
Table 6.13 Area Comparison - Temporal Duplication Mechanism 118
Table 6.14 Number of Kernel Executions and Number of Steps to Find the Best Configuration....118
Table 6.15 MWPUETF Normalized to the Oracle Processor 121
Table 6.16 Best Static Configuration When Prioritizing Each of the Axes 122
Table 6.17 Best Configuration - Priority: Energy (0.8)....122
Table 6.18 MWPUETF Normalized to the Oracle Processor - Priority: Energy Consumption 123
Table 6.19 Best Configuration - Priority: Performance (0.8)....124
Table 6.20 MWPUETF Normalized to the Oracle Processor - Priority: Performance.124
Table 6.21 Best Configuration - Priority: Fault tolerance (0.8) 125
Table 6.22 MWPUETF Normalized to the Oracle Processor - Priority: Fault Tolerance 125

Table A.1 Failure Rate and Performance Degradation Comparison....152
Table A.2 Area and Power Dissipation Comparison....153

Table B.1 Best Configuration - Priority: Energy (0.5)....155
Table B.2 Best Configuration - Priority: Energy (0.6)....156
Table B.3 Best Configuration - Priority: Energy (0.7)....156
Table B.4 Best Configuration - Priority: Energy (0.8)....156
Table B.5 Best Configuration - Priority: Energy (0.9)....157
Table B.6 Best Configuration - Priority: Performance (0.5)....157
Table B.7 Best Configuration - Priority: Performance (0.6)....158
Table B.8 Best Configuration - Priority: Performance (0.7)....158
Table B.9 Best Configuration - Priority: Performance (0.8)....158
Table B.10 Best Configuration - Priority: Performance (0.9)....159
Table B.11 Best Configuration - Priority: Fault tolerance (0.5)....159
Table B.12 Best Configuration - Priority: Fault tolerance (0.6)....160
Table B.13 Best Configuration - Priority: Fault tolerance (0.7)....160
Table B.14 Best Configuration - Priority: Fault tolerance (0.8)....160
Table B.15 Best Configuration - Priority: Fault tolerance (0.9)....161

CONTENTS

1 INTRODUCTION **19**
2 BACKGROUND AND RELATED WORK **23**
2.1 Fault Tolerance **23**
2.1.1 Transient Faults 23
2.1.1.1 Dual Modular Redundancy (DMR) with Rollback 24
2.1.1.2 Duplication with Comparison (DWC) 25
2.1.1.3 Triple Modular Redundancy (TMR) 26
2.1.1.4 Watchdog Processors 27
2.1.1.5 Adaptive Fault Tolerance Approaches 29
2.1.2 Permanent Faults 30
2.2 Energy Consumption **31**
2.2.1 Power Dissipation 31
2.2.1.1 Dynamic Power Dissipation 31
2.2.1.2 Static Power Dissipation 32
2.2.2 Energy Optimization Techniques 32
2.2.2.1 Clock Gating 32
2.2.2.2 Power Gating 33
2.2.2.3 Dynamic Voltage and Frequency Scaling (DVFS) 34
2.2.3 Energy Optimization on the ρ-VEX Processor 35
2.2.4 Energy-aware Fault Tolerance 37
2.3 Performance **37**
2.3.1 ILP and TLP exploitation 37
2.3.1.1 Instruction-Level Parallelism (ILP) 37
2.3.1.2 Multiple Applications and Thread-Level Parallelism (TLP) 39
2.3.2 Adaptive Multi-core Architectures 40
2.4 Fault Tolerance, Energy Consumption, and Performance Trade-off **41**
2.5 Contributions of this Thesis **43**
2.5.1 Fault Tolerance 44
2.5.2 Energy Consumption 44
2.5.3 Performance and Learning 45
2.5.4 Fault Tolerance, Energy Consumption, and Performance Trade-off 45
3 IMPLEMENTED TECHNIQUES **47**
3.1 Fault Tolerance **47**
3.1.1 Basic Duplication with Rollback 47
3.1.1.1 Spatial duplication 50
3.1.1.2 Temporal and Spatial Duplication 53
3.2 Energy Optimization **55**
3.3 Performance **56**
3.3.1 ILP Control 56
3.3.1.1 ILP Control for Fault Tolerance 57
3.3.1.2 ILP Control for Energy Optimization 57
4 PROPOSED ADAPTIVE AND POLYMORPHIC PROCESSORS **59**
4.1 ρ-VEX Processor Background **59**
4.1.1 Static ρ-VEX Processor 59
4.1.2 Dynamic ρ-VEX Processor 60
4.2 Proposed Adaptive Processor **65**
4.2.1 Reliability-oriented Configuration 66
4.2.2 Energy-oriented Configuration 68

4.2.3 Final Remarks Regarding the Threshold Configuration 70
4.3 Proposed Polymorphic Processor 71
4.3.1 Learning Phase 72
4.3.2 Runtime Phase 76
5 PROPOSED HYBRID FAULT INJECTOR 79
5.1 Motivation 79
5.2 Fault Injectors - Related Work 81
5.3 Fault Injection Framework 83
5.3.1 Implementation 83
5.3.2 Fault Model 84
5.3.3 Hybrid Mechanism 84
5.4 Fault Injection Accuracy 85
6 RESULTS 89
6.1 Methodology 89
6.1.1 Benchmarks 90
6.1.2 Temporal Duplication and Optimization Module Simulators 92
6.2 Hybrid Fault Injector Results 93
6.2.1 RTL vs Gate-level: Accuracy Comparison 93
6.2.2 Fault Injection Performance 94
6.3 Adaptive ρ-VEX Processor 96
6.3.1 Spatial Duplication Without Power Gating 97
6.3.1.1 Failure Rate and Performance 97
6.3.1.2 Dynamic Threshold Adaptation 101
6.3.1.3 Area, Power Dissipation, and Energy Consumption 101
6.3.1.4 Comparison With Other Fault Tolerance Techniques 103
6.3.2 Spatial Duplication With Power Gating 105
6.3.2.1 Performance 106
6.3.2.2 Energy and Area 107
6.3.2.3 Failure Rate 107
6.3.2.4 EDFP - Energy-Delay-Failure Product 109
6.4 Polymorphic ρ-VEX Processor 110
6.4.1 Temporal and Spatial Duplication Mechanism 110
6.4.1.1 Mean Work Per Unit of Energy to Failure - MWPUETF 110
6.4.1.2 Duplication ratio 112
6.4.1.3 Performance 114
6.4.1.4 Energy and Area 116
6.4.2 Optimization Algorithm 117
6.4.2.1 Number of Steps to Find the Best Configuration 117
6.4.2.2 MWPUETF Comparison with the Same Weight for All Axes 119
6.4.2.3 MWPUETF Comparison with Different Weights for Each Axis 121
6.4.2.4 System Performance and Energy Consumption 125
7 CONCLUSION AND FUTURE WORK 129
7.1 Summary of Contributions 129
7.2 Future Work 130
7.2.1 Additional Techniques for Energy Optimization 130
7.2.2 Support for Multithreaded Applications 130
7.2.3 Criticality-aware Duplication 130
7.2.4 ILP Control for the Polymorphic Processor 131
7.2.5 Improved Application Dispatcher 131
7.3 Publications 131
7.3.1 Journals 131

7.3.2 Conferences and Workshops 132
REFERENCES 135
APPENDIX A — PHASE-CONFIGURABLE DUPLICATION 149
A.1 Results 151
APPENDIX B — EVALUATION OF DIFFERENT PRIORITIES FOR THE MWPUETF 155
B.1 Priority: Energy Consumption 155
B.2 Priority: Performance 157
B.3 Priority: Fault Tolerance 159
APPENDIX C — RESUMO EM PORTUGUÊS 163
C.1 Introdução 163
C.2 Técnicas propostas 163
C.2.1 Tolerância a falhas 164
C.2.1.1 Duplicação com rollback 164
C.2.1.2 Duplicação baseada em fases 164
C.2.1.3 Duplicação espacial - duplicação quando possível 164
C.2.1.4 Duplicação temporal e espacial 164
C.2.2 Consumo de energia 165
C.2.3 Performance 165
C.3 Processador adaptativo 165
C.4 Processador polimórfico 165
C.5 Injetor de falhas híbrido 166
C.6 Metodologia 166
C.7 Resumo dos resultados 166
C.8 Conclusões 167

1 INTRODUCTION

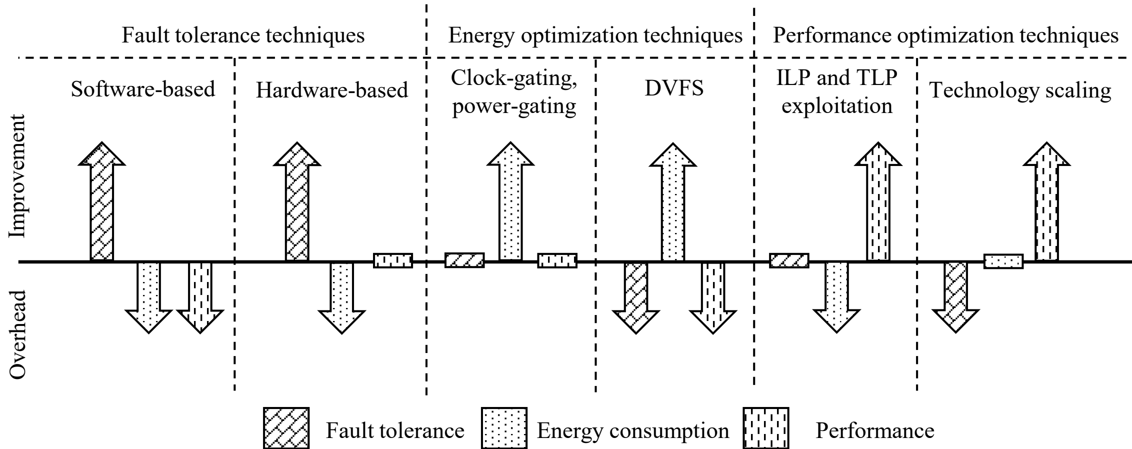
As technology continues to evolve, more attention is paid to energy consumption and fault tolerance when designing new processors. While most of the embedded devices are heavily dependent on battery power, General-Purpose Processors (GPPs) are being held back by the limits of Thermal Design Power (TDP), highlighting the importance of reducing energy consumption. In addition, the need for fault tolerance on both space and ground-level systems is increasingly present in current processor designs. As the feature size of transistors decreases, their reliability is getting compromised as they become more susceptible to soft errors (SHIVAKUMAR et al., 2002). Therefore, energy consumption and fault tolerance, together with performance, should be considered and balanced to address the aforementioned issues according to given design constraints.

On the other hand, current processors are designed to focus on one or, at most, two of these axes. Achieving the ideal balance among them is challenging, due to their conflicting nature. For instance, reducing energy consumption will likely reduce performance; increasing fault tolerance will increase energy consumption and possibly reduce performance; and improving performance will affect the energy consumption and possibly reduce fault tolerance.

Figure 1.1 depicts a high-level overview, comparing the improvement and overhead of applying common optimization techniques to the axes of fault tolerance, energy consumption, and performance (further details about each of these techniques will be discussed in Chapter 3). Let us consider fault tolerance: replication techniques are widely used to detect or mask faults during the execution: while hardware-based techniques increase the power dissipation, software-based ones increase execution time; and both increase the total energy consumption.

Energy optimization techniques usually save energy by shutting down idle hardware or by reducing the supply voltage and frequency, which will very likely reduce performance; and when operating in very low voltage states, the reliability may also be reduced. Performance improvements may be achieved from several different techniques, from core optimizations to better exploit the Instruction-Level Parallelism (ILP); to system-level techniques, which comprise several cores and caches hierarchy to exploit the Thread-Level Paralellism (TLP) from applications. All these techniques focus on a single axis, and the incurred overhead in the other two axes varies depending on the chosen technique.

Figure 1.1: High-level Overview of Common Optimization Techniques



Source: The Author

In this scenario, we propose two processor designs to trade-off fault tolerance, energy consumption, and performance, named *adaptive* and *polymorphic* processors. The adaptive balances these three axes based on design time parameters, while the polymorphic dynamically evaluates different configurations and reconfigures the processor, adapting the hardware to the current application. The proposed processor relies on a set of optimization techniques that work together to provide improvements in each of the aforementioned axes with a low overhead in the other ones.

The ρ-VEX Very Long Instruction Word (VLIW) processor (WONG; Van As; BROWN, 2008) was used as target architecture for this study. VLIW processors are representative examples of current VLIW multiple-issue architectures (e.g., Intel Itanium (SHARANGPANI; ARORA, 2000), Trimedia CPU64 (EIJNDHOVEN et al., 1999), and TMS320C6745 (INSTRUMENTS, 2011)). Notably, a VLIW multi-core was recently used by the National Aeronautics and Space Administration (NASA) in its Mars rover for image processing (BORNSTEIN et al., 2011). These processors are able to deliver high-performance at a low energy consumption cost, as they do not rely on runtime dependency-checking mechanisms for scheduling the instructions that will be executed in parallel (HUCK et al., 2000). Therefore, VLIW processors are suitable options for safety-critical systems in several fields, such as automotive, space, and avionics.

Next, all mechanisms and tools that were developed and evaluated to get to the proposed processors are introduced. Note that several techniques were evaluated, and the ones with the best outcome were chosen to be integrated into each version of the processor (therefore some of them were not included into the final version of the processor).

1. Fault Tolerance Techniques:

  - Spatial redundancy: a modified dual modular redundancy approach with roll-back that exploits idle issue-slots (i.e., functional units that execute No-Operations (NOPs)) and dynamically duplicates instructions to improve the reliability of the system by detecting and correcting errors.
  - Temporal and spatial redundancy: takes the former mechanism one step further by also exploiting idle issue-slots in different bundles, that is, in different cycles, and not only within the same bundle.
  - Hybrid fault injection tool that improves fault injection speed when compared to other simulation fault injection approaches, while maintaining the accuracy of gate-level fault injection.

2. Energy Optimization Techniques:

   - A power gating mechanism that can be applied to the datapath to reduce both static and dynamic power. As idle issue-slots are completely turned off, they are not able to execute duplicated instructions, but they also reduce the sensitive area of the processor, influencing fault tolerance.

3. Performance Optimization Techniques:

   - A dynamic ILP controller that can reduce the parallelism (and therefore performance) at runtime. In this case, issue-slots are artificially freed by automatically moving operations to the next cycle, offering opportunities to duplicate more instructions or maximize the power gating phases (as it will be explained, power gating demands that an issue-slot must be turned off for a minimum period, so the gains in power overcome its costs).
   - Application dispatcher that takes into account the issue-width requirements of each application and the ability of the processor to change its configuration during runtime to maximize the number of applications that are executed concurrently.
   - Optimization algorithm that evaluates and chooses the best configuration for each application that is being executed, considering the trade-off between the three axes.
   - The dynamic version of the ρ-VEX, which is able to change the issue-width of the processor during runtime, was used and extended so the optimization modules could also resize their resources or enable/disable certain features during runtime to exploit the trade-off between the aforementioned axes.

Each version of the proposed processor uses a set of these techniques in addition to a decision module, which controls these techniques during runtime. The adaptive processor uses thresholds to cope with different application behaviors, while the polymorphic processor can dynamically test and choose the best configuration for each benchmark considering fault tolerance, performance, and energy consumption. The polymorphic's learning is done by executing each iteration of a given kernel with a different configuration and evaluating its outcome (a metric that considers the trade-off between the three axes). The learning stops when the best configuration (the one that delivers the best trade-off) is found, resulting in a reduced number of tests. In addition, an application dispatcher module was implemented to dynamically schedule multiple applications concurrently, according to the available hardware. Therefore, it is possible to dynamically and efficiently adapt the processor configuration to the behavior of the application.

The remainder of this work is organized as follows. Background and related work comprising the aforementioned axes are discussed in Chapter 2. Chapter 3 describes the proposed techniques for each of the axes and their implementation. Next, in Chapter 4, the adaptive and polymorphic processor implementations are discussed, and Chapter 5 presents a hybrid simulation-based fault injector that is able to speed up the fault injection campaign while maintaining the accuracy of gate-level fault injection. In Chapter 6 the results are discussed in terms of fault tolerance, energy consumption, performance, area, and the trade-off between these axes. Finally, Chapter 7 concludes this work and discusses future directions.

2 BACKGROUND AND RELATED WORK

In this chapter, the background and related work comprising the axes of fault tolerance, energy consumption, and performance will be discussed. First, the faults will be classified and related work regarding fault tolerance will be discussed. Second, energy optimization and energy-aware fault tolerance techniques will be presented. Third, the axis of performance will be assessed. Finally, the trade-off among these axes will be discussed and related work will be compared to the proposed methodology.

2.1 Fault Tolerance

A fault may be classified into one of the two large groups: transient or permanent. Next, each of these groups is discussed in detail and fault tolerance techniques are presented.

2.1.1 Transient Faults

Single-event effects (SEEs) are caused by numerous energetic particles such as protons and heavy ions from space or neutron and alpha particles at ground-level. They are created from the deposition of energy from a single ionizing particle in either sequential or combinational logic in the silicon. The former is called Single-Event Upset (SEU) and characterizes that a stored value in a memory cell had its value inverted (bit flip). The latter is called Single-Event Transient (SET) and it generates a transient current pulse in the combinational circuit, which also may affect memory elements if the inverted signal is captured by a clock edge. SETs may not be captured in a memory element by one of the following reasons (SHIVAKUMAR et al., 2002):

- *Logical masking:* occurs when the particle strike affects a signal that does not affect the output of a subsequent gate, which is determined by its other inputs.
- *Electrical masking:* happens when the pulse from a particle strike is attenuated by subsequent gates, not affecting the result of the circuit.
- *Latching-window masking:* due to a pulse that reaches a latch in a clock transition in which it does not capture its input value, consequently not affecting the result.

For Application-Specific Integrated Circuits (ASICs) and antifuse-based Field-Programmable Gate Arrays (FPGAs), these effects are transient, and only last until the next time a value is loaded to that same element. For Static Random Access Memory (SRAM)-based FPGAs, in addition to the same vulnerabilities of ASICs, the configuration bits are also sensitive, which means that the configuration of the programmed design may change and lead to wrong computation. A fault in the configuration bits is permanent in the sense that it needs reconfiguration in order to restore correct behavior (de Lima Kastensmidt et al., 2004). Approaches have been proposed to perform the reconfiguration: scrubbing (BERG et al., 2008), readback and partial reconfiguration of the faulty parts (CARMICHAEL; CAFFREY; SALAZAR, 2000), etc.

Several works have been proposed for the detection and correction of transient faults in multiple-issue processors (e.g., VLIW and superscalar). These works aim to improve the fault tolerance of the target system, typically based on redundancy, which may be implemented in software, hardware, or both. Next, some of these techniques will be discussed.

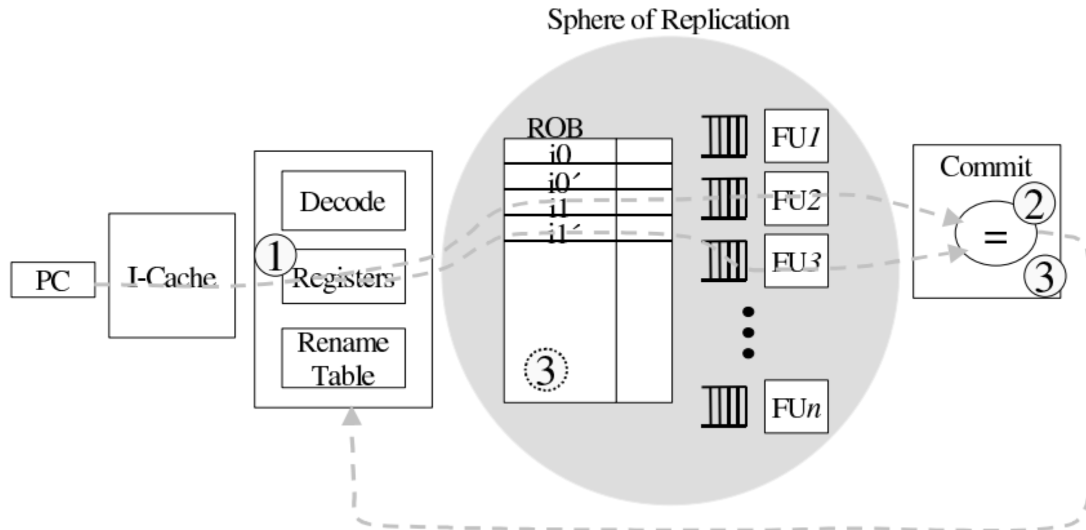
2.1.1.1 Dual Modular Redundancy (DMR) with Rollback

Dual Modular Redundancy (DMR) based on checkpoints with rollback was used by Xiaoguang et al. (2015) and Yang and Kwak (2010) to detect and correct errors. Whenever an error is detected, the state in which the execution was correct is recovered. Therefore, the latency to detect the error on these approaches will vary according to the periodicity of the checkpoints (i.e., when a new checkpoint must be made). These approaches were implemented in software.

In order to reduce the energy consumption of rollback and recovery protocols for High-Performance Computing (HPC), Ibtesham et al. (2014) propose to apply a checkpoint compression scheme. This approach reduces data volume and traffic, incurring higher computational overhead to process the compressed packages. Even though there is a computational cost to compress and decompress the checkpoint information, the energy consumption is reduced due to the reduction in the execution time.

A hardware-based replication with rollback mechanism for superscalar processors is proposed by Ray, Hoe and Falsafi (2001). In this approach, the application is replicated into two or more threads and the computed results are checked before commit. If any inconsistency is found, the rollback is activated, the re-order buffer is flushed, and the execution restarts by fetching once more the next-PC from the last committed instruction.

Figure 2.1: Hardware-based Replication with Rollback for Superscalar Processors



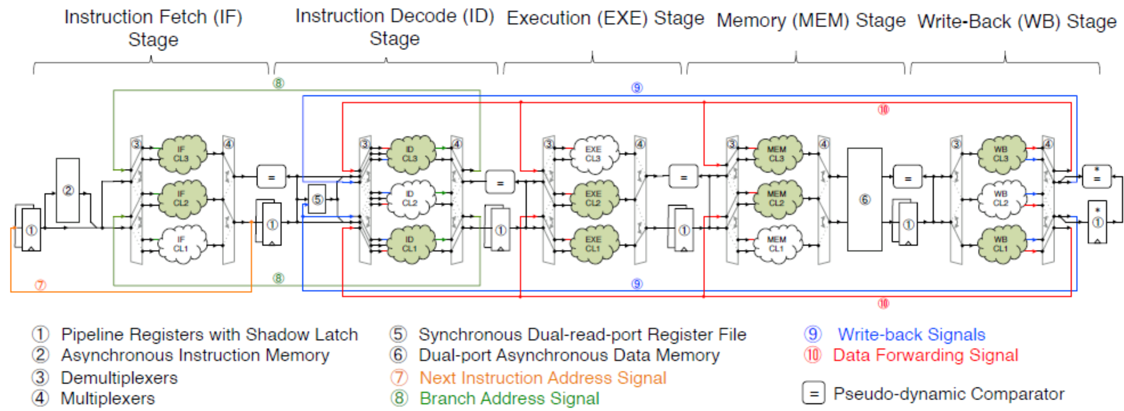
Source: (RAY; HOE; FALSAFI, 2001)

Figure 2.1 depicts this approach: (1) the register renaming capabilities were adapted so the instructions fetched from a single stream can be issued redundantly into two or more data-independent threads; (2) the multiple threads are checked against each other before the committing the instruction; (3) any inconsistency in the redundant results rewinds the execution. These threads have tightly coupled executions (both must execute at the same time) and, consequently, each type of functional unit used by the main thread must be duplicated so that the replicated thread may execute concurrently.

2.1.1.2 Duplication with Comparison (DWC)

Bolchini (2003) and Hu et al. (2009) propose software-based redundancy based on Duplication With Comparison (DWC) for VLIW data paths aiming to reduce the performance overhead by using the idle functional units. However, these techniques still present huge performance degradation and increase code size, as they are implemented in software. Mitropoulou, Porpodas and Cintra (2014) propose an optimization to the DWC's generated code by reducing the impact of the basic block fragmentation caused by the check instructions, having lower, but still not negligible, performance degradation than the previous two techniques.

Figure 2.2: Triplication of the Combinational Logic with a Stand-by Module



Source: (WALI et al., 2015)

2.1.1.3 Triple Modular Redundancy (TMR)

Another common approach is to triplicate hardware components and use a majority voter to mask the faults, Triple Modular Redundancy (TMR), as implemented by Schölzel (2007) and Chen and Leu (2010). In these cases, they only triplicate the functional units of a VLIW processor rather than the entire processor; therefore, it is possible to reduce area and power dissipation costs. Schölzel (2007) proposed the Reduced TMR, in which both hardware and software needed to be changed. If the two instructions (main and duplicated) compute different results, the instruction is executed a third time. Hu et al. (2005) propose a similar approach to Schölzel (2007). However, instruction replication is done in software, so the binary code is changed, even though there is no area overhead. In the same way, replication is done partially to some instructions to amortize the costs in performance.

Wali et al. (2015) propose to triplicate the combinational logic parts of a MIPS processor. However, only two copies run in parallel, while the third one uses time-redundancy in case of an error to vote the correct result. This approach is depicted in Figure 2.2. A state-machine controls which two combinational logic modules will be active, and a control module manages the comparator, pipeline register, demultiplexer and multiplexer. In order to re-compute the operation in case of an error (third execution), shadow latches are used to hold the last fault-free state of the pipeline. Then, the comparator votes the three computed results. Anjam and Wong (2013) propose a TMR approach to be applied on the synchronous flip-flops of a VLIW processor.

Psiakis, Kritikakou and Sentieys (2017) propose to exploit temporal and spatial duplication for the VEX Instruction Set Architecture (ISA) in order to provide more flex-

ibility in the instruction replication, as instructions that cannot be replicated in a given bundle are stored in a buffer so they can be executed in a subsequent cycle. The authors evaluate the duplication (for fault detection only) and the triplication of instructions in order to provide fault tolerance. However, no reliability, area, power, and energy results are presented, only performance ones. In addition, the memory latency is not exploited to execute duplicated instructions (i.e., execute instructions while the processor is waiting for the memory), as the VEX simulator does not provide a cache and memory model.

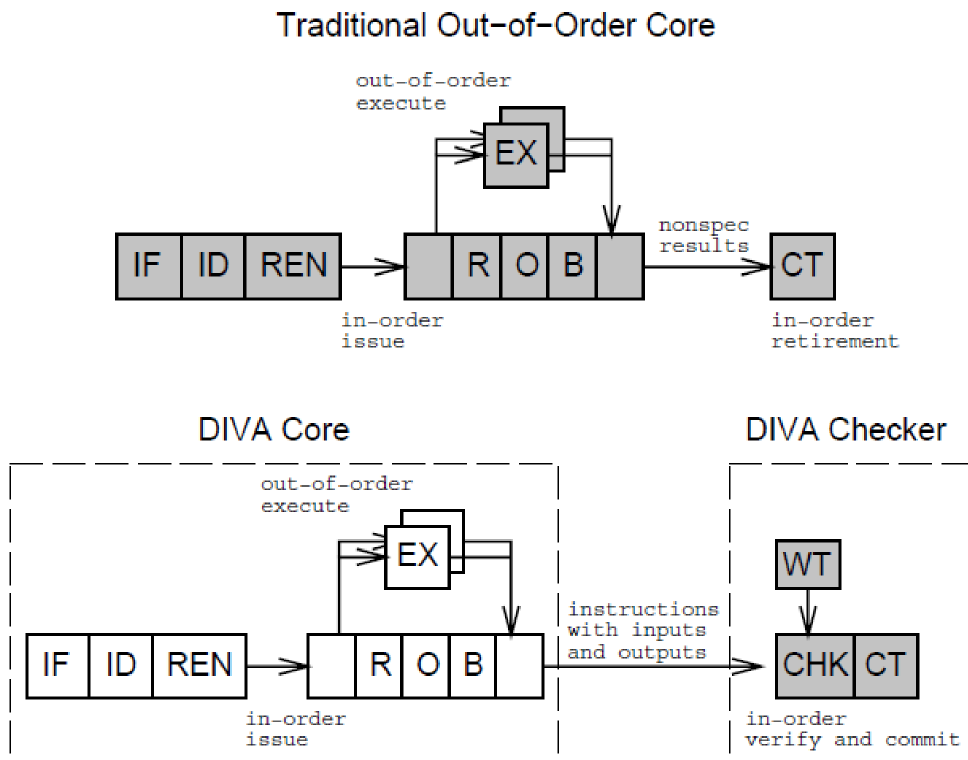
2.1.1.4 Watchdog Processors

Watchdog processors (MAHMOOD; MCCLUSKEY, 1988) execute concurrently to the main processor, and they compare the outputs from the main processor with their own (pre-computed or concurrently computed). Examples of watchdog processors are discussed next. DIVA (AUSTIN, 1999) proposes to increase the reliability of a superscalar processor by augmenting the commit phase of the pipeline with a checker unit (watchdog processor). This checker is a simple in-order processor that does not have any mechanism to speed up computation (e.g., predictors, renamers and dynamic schedulers). The checker will verify and commit the results if the computation is correct, and flush the computation and restart the processor in case of an error, as presented in Figure 2.3. The checker is considered to be more robust than the other parts of the circuit and the verification cost of the checker is lower than verifying traditional processor design due to its simplicity. In addition, the authors discuss the buffering of results in order to deeply pipeline the checker unit, which permits implementations with large time margins and large transistors (more resistant to transient faults and radiation interference).

Other approaches use redundant threads as watchdog processors (ROTENBERG, 1999; REINHARDT; MUKHERJEE, 2000). SHREC (SMOLENS et al., 2004) proposes an approach for asymmetric re-execution, similar to DIVA. In addition, it allows threads to be replicated and staggered. Hence, the difference in the execution progress between the leading and trailing (replicated) thread hides cache-miss latencies and allows the leading thread to provide branch prediction information to the trailing thread. The trailing thread instructions are only checked using input operands produced by the leading thread, avoiding bottlenecks in the issue queue and reorder buffer. In addition, the functional units are shared, unlike DIVA.

Rashid et al. (2005) propose a thread-level redundant execution that consumes less energy than replicating the whole program and running it on identical hardware. This is

Figure 2.3: Traditional Out-of-order Compared to the DIVA Approach



Source: (AUSTIN, 1999)

done by parallelizing the trailing thread and running it in several small cores (i.e., simple cores with reduced frequency). Moreover, the leading thread provides the trailing thread with branch information and L1 cache prefetch hints. Periodic checkpoints are performed by the leading thread to allow the rollback to a fault-free state when an error is detected. Madan and Balasubramonian (2007) propose a similar approach to Rashid et al. (2005), using Dynamic Voltage and Frequency Scaling (DVFS) instead of only Dynamic Voltage Scaling (DVS), and in-order execution on the trailing thread instead of an out-of-order processor.

RECVF (SUBRAMANYAN et al., 2010) proposes to forward critical instruction results from the leading to the trailing thread, so the latter may execute faster and the energy consumption can be reduced by executing the trailing core at a lower voltage and frequency. The results to be forwarded are chosen based on heuristics proposed by Tune et al. (2001). For instance, one approach is to forward instructions on the head of the Re-order Buffer (ROB), as these instructions are likely to be on the critical path (other instructions have to wait for the execution of these ones, so they can be committed); forward instructions in the head of the instruction queue; forward every *Nth* instruction; forward mispredicted branches/jumps; forward all possible values (oracle heuristic given

infinite storage and bandwidth); among other heuristics. In order to recover from an error, the cores are reset to the instruction following the last verified instruction when an error is detected.

2.1.1.5 Adaptive Fault Tolerance Approaches

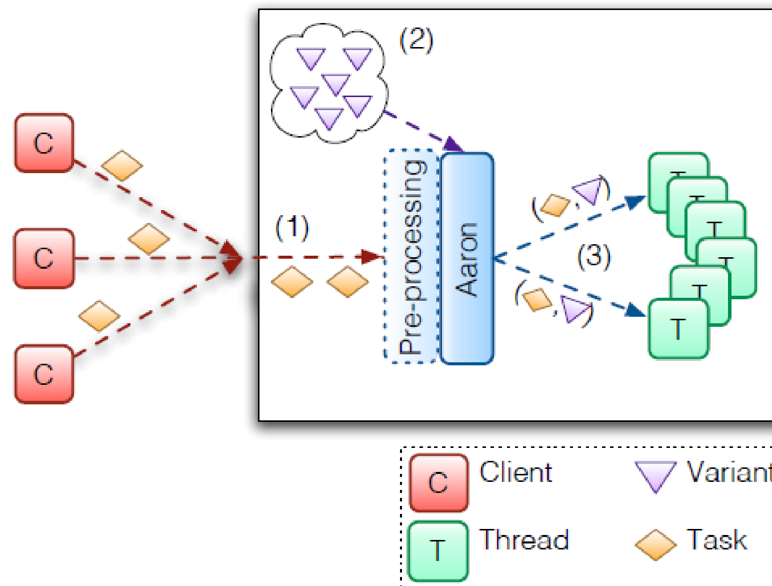
Some works exploit the previous techniques in order to provide an adaptive fault tolerance mechanism. Jacobs et al. (2012) propose an adaptive framework that switches between different fault tolerance techniques depending on a priori knowledge of the environment, external events, or application-triggered events. The supported fault-tolerance modes are triple modular redundancy, duplication with comparison, Algorithm-Based Fault Tolerance (ABFT), internal TMR, and high-performance (no fault tolerance). This approach is for FPGAs only, as the hardware needs to be reconfigured.

An adaptive checkpoint mechanism was proposed by Zhang and Chakrabarty (2004), in which the checkpointing interval is adjusted during the execution based on the occurrence of faults and the available slack. An offline preprocessing based on linear programming is used to determine the parameters that are provided to the online checkpointing procedure. That is, the preprocessing step obtains the slack time for each job according to the deadline constraints and the threshold for the minimum number of checkpoints. Even though the checkpointing is adaptively made, the detection latency is still greater than zero, besides the need for preprocessing.

Works that aim to minimize the performance cost of the software-based fault tolerance mechanisms were also proposed. Nakka, Pattabiraman and Iyer (2007) propose to replicate only critical instructions of the application. This is done by first identifying which are the critical variables, extracting the critical code sections and finally instrumenting the code with check instructions. This approach is only able to detect faults, not to correct them. It detects 87% of instruction errors and 97% of data errors.

Aaron (BRÜNINK et al., 2011) tackles software and hardware errors by using diversified software components in the Central Processing Unit (CPU) spare cycles. Eight methods are used in this diversification, including SWIFT (REIS et al., 2005b), a software-based fault tolerance approach that duplicates instructions and registers. The system load is estimated, and the scheduler chooses the best variant to use the spare cycles (e.g., executing a reliability-oriented variant, which takes longer to execute, but is able to provide fault tolerance), as presented in Figure 2.4. Therefore, whenever load permits, more fault coverage is achieved; even though it is only able to detect errors, not correct them. Gomaa

Figure 2.4: Aaron Scheduling Example



Source: (BRÜNINK et al., 2011)

and Vijaykumar (2005) propose two mechanisms to exploit unused resources on low ILP phases: the first replicates the main thread and compares the results to detect faults; the second applies a reuse technique to detect and compare the results of implicit redundant operations within the main thread. The previous four works discussed in this subsection are implemented in software, having lower performance when compared to a hardware-based mechanism. Most of these approaches are only able to detect, and not to correct, a fault.

2.1.2 Permanent Faults

Single event effects can also be destructive, damaging the device permanently. In this case, reconfiguration of FPGAs will not correct the error and ASICs are also affected. Permanent damage may occur due to several factors, for instance, the energy from a charged particle may lead to excessive supply power. Also, the Total Ionizing Dose (TID), which is a cumulative long-term ionizing damage due to protons and electrons, has the potential to damage the device permanently. Aging effects are also critical, especially for space missions, where system maintenance or replacement is difficult (BOLCHINI; SANDIONIGI, 2010).

As the detection of permanent faults is not in the scope of this thesis, only a few works will be briefly discussed next. One technique is to relocate the affected design to a

part of the FPGA in which there are no errors, via reconfiguration. Fay et al. (2007) use relocation to recover the system in case of a hard fault, whereas Srinivasan et al. (2008) also use this approach to recover from performance degradation due to aging effects.

Other approaches for hardware relocation in order to avoid faulty parts of the circuit are proposed by Bolchini, Miele and Sandionigi (2012), Noji et al. (2010), Mitra et al. (2004). Techniques as TMR or N-modular Redundancy (NMR) naturally mask permanent faults as long as the majority of the results is still correct.

2.2 Energy Consumption

As the energy consumption is defined by the integral of power dissipation over time, next, we discuss the classification of power, which can be divided into two large groups: dynamic or static power. Then, we will discuss optimization mechanisms to reduce the energy consumption of the circuit.

2.2.1 Power Dissipation

2.2.1.1 Dynamic Power Dissipation

Dynamic power ($P = C * V^2 * A * f$) varies according to the load capacitance (*C*), supply voltage (*V*), activity factor (*A*), and operating frequency (*f*), next each of these terms is discussed (KAXIRAS; MARTONOSI, 2008).

- *Capacitance:* mainly depends on the technology of the transistors and wire length of the chip.
- *Supply Voltage:* due to its quadratic influence on power dissipation, when the voltage is reduced, the power is reduced proportionally to the square of that factor.
- *Activity Factor:* refers to how often the transistors transition from '0' to '1' or '1' to '0', which is represented by a fraction that varies between zero and one (i.e., 0% to 100%).
- *Clock Frequency:* frequency affects not only the power dissipation, but also the supply voltage. For instance, for higher clock frequencies, higher supply voltage is required, and vice-versa. Thus, the pair $V^2 * f$ offers the potential of cubic reduction in the power dissipation, which is exploited by energy optimization techniques and will be discussed in detail in the next subsections.

2.2.1.2 Static Power Dissipation

Reducing the feature size of transistors as well as supply and threshold voltages cause an increase in the leakage current, which increase the static power dissipation (leakage) of the circuit (DE; BORKAR, 1999; RAO et al., 2003). Leakage represents 20% to 30% of the total power dissipation of modern processors (BORKAR, 1999; MAIR et al., 2007; RUSU et al., 2007).

Leakage energy can come from sub-threshold leakage and gate leakage, among other sources (KAXIRAS; MARTONOSI, 2008). Sub-threshold leakage represents the power that is dissipated by a gate that is supposed to be off. That is, until the current reaches the threshold voltage (voltage in which the transistor switches on), the transistor charge is leaked to the ground. In addition to the threshold voltage, another important parameter that determines the sub-threshold leakage power is the temperature (it depends exponentially on the temperature), increasing the temperature, increases the leakage. Gate leakage occurs due to the tunneling of electrons through the gate insulator, which separates the gate terminal from the transistor channel. It is also dependent on the temperature, and strongly dependent on the insulator thickness, gate-to-source bias, and gate-to-drain bias.

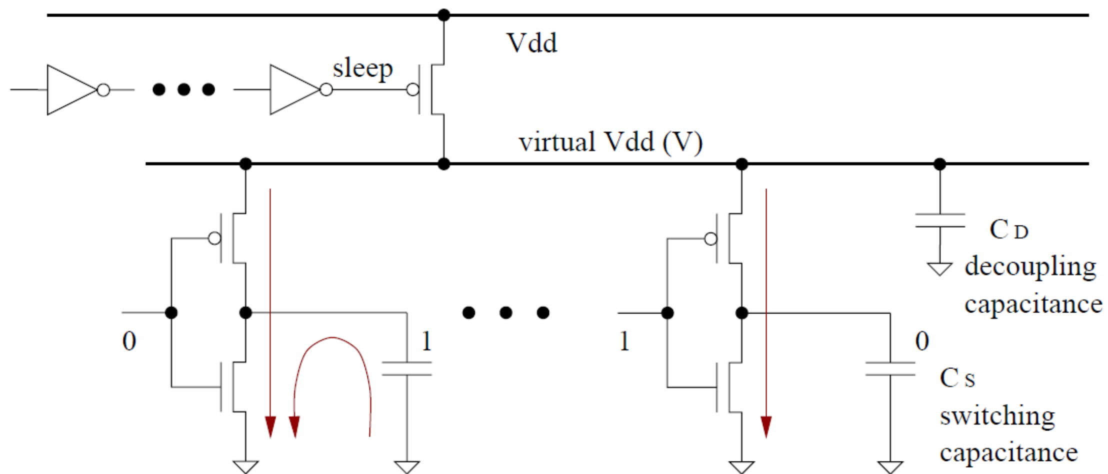
2.2.2 Energy Optimization Techniques

Several techniques may be applied in order to reduce the power dissipation of the circuit and, in most cases, the energy consumption of the applications. These techniques are applied at several and different granularities: from flip-flops to whole functional units or subsystems (e.g., memory and processor). Some of them are discussed next.

2.2.2.1 Clock Gating

Clock gating consists in disabling the clock signal to idle units, therefore, saving dynamic but not static power. For small circuits or individual flip-flops, clock gating reduces the power dissipation by eliminating the switching activity of these units. At this granularity, such transformations are routinely applied by the Register-Transfer Level (RTL) compiler. On the other hand, for coarser granularities, high-level control policies are needed (KAXIRAS; MARTONOSI, 2008).

Figure 2.5: Power Gating Header Transistor



Source: (HU et al., 2004)

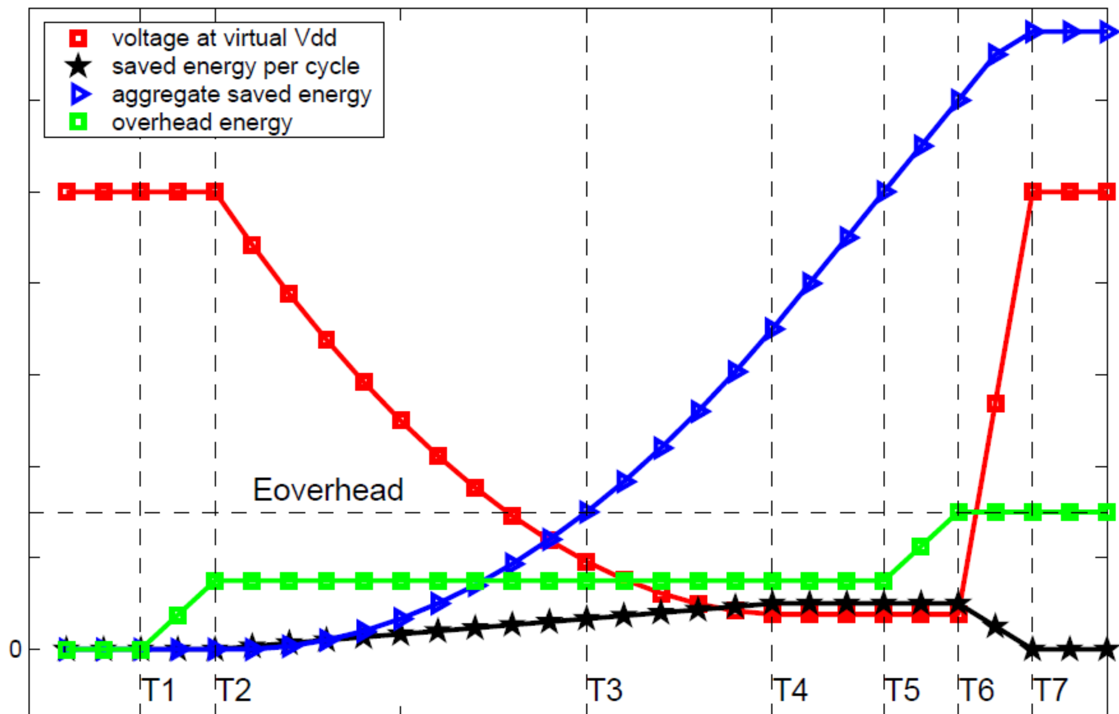
Due to its simplicity, this approach may be applied on a cycle-by-cycle basis with limited penalty in area and timing. This technique is used in both low-power designs (Intel XScale (CLARK et al., 2001)) and high-performance processors (IBM's Power5 (CLABES et al., 2004)).

2.2.2.2 Power Gating

Power gating is used to address leakage power dissipation by shutting off idle blocks of the circuit, thus, reducing both static and dynamic power dissipation. A suitably sized header transistor is used to turn off the supply voltage to the circuit block, which creates a virtual *Vdd*, as presented in Figure 2.5. A sleep signal controls this transistor so the *Vdd* can pass through when the circuit is active (virtual *Vdd* = *Vdd*) and gate the *Vdd* when idle (virtual *Vdd* = 0). The sleep signal is controlled via a global policy that varies according to the implementation. As one more transistor is used, this comes at the cost of performance penalty and power overhead as the power lines are not able to be charged and discharged in one clock cycle. In order to cope with this additional cost of the mechanism, the break-even point must be considered, which is the point when the leakage energy savings equals the energy overhead of switching the circuit on and off (which will be explained in detail next). After reaching this point, the energy consumption of the system is reduced (HU et al., 2004).

Figure 2.6 (HU et al., 2004) depicts the key intervals that must be considered for power gating. The inactivity interval starts at $t = 0$ and goes until $t = T_1$, when the control

Figure 2.6: Key Intervals for the Power Gating Technique



Source: (HU et al., 2004)

circuit makes the decision of power gating the unit. At $t = T_2$, the header device receives the sleep signal and the virtual *Vdd* starts going down. As the voltage decreases, the amount of leakage energy saved per cycle increases. The break-even point is achieved when the execution reaches $t = T_3$, at this point the aggregate leakage energy savings equals the energy overhead of switching ON (T_1 to T_2) and OFF (T_5 to T_6) the header device.

At $t = T_4$ the reduction in voltage at the virtual *Vdd* saturates (not necessarily at zero due to the leakage of the header device). The control logic detects the next busy interval at $t = T_5$ and the sleep signal is de-asserted, and in $t = T_6$, the header device is turned on. From $t = T_6$ to $t = T_7$, the virtual *Vdd* is charged up to the *Vdd* level. As the *Vdd* increases, the energy savings per cycle is gradually reduced, reaching zero at $t = T_7$ (when the unit is completely active). Therefore, the energy savings are proportional to the idle time that can be exploited by this mechanism while executing an application.

2.2.2.3 Dynamic Voltage and Frequency Scaling (DVFS)

The dynamic power dissipation equation (previously discussed) clearly shows the significant leverage that voltage and frequency adjustment may provide (potential cubic

influence in the dynamic power dissipation). However, this comes at the cost of performance and reliability: by reducing voltage and frequency, the performance is reduced as well, not achieving a cubic reduction in the energy consumption due to the longer execution time. That is, even though the dynamic power dissipation may be reduced by a cubic factor, the longer execution time (which implies in more static power dissipation) prevents a cubic reduction in the total energy consumption (KAXIRAS; MARTONOSI, 2008). Reliability is also reduced when reducing the voltage, because it allows low energy particles to create a critical charge that leads to a transient fault (ZHU; MELHEM; MOSSÉ, 2004).

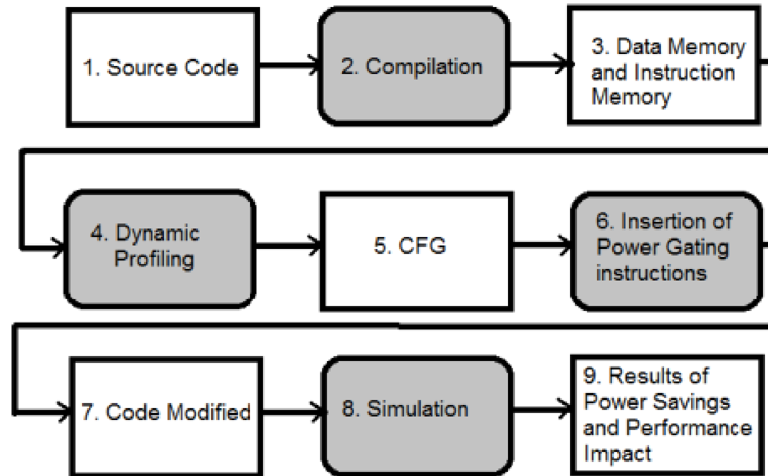
2.2.3 Energy Optimization on the ρ-VEX Processor

Giraldo, Wong and Beck (2016) propose to insert customized instructions at compile time (static) to power-gate functional units and parts of the register file, based on the profiling of the application, on the ρ-VEX processor. Their methodology follows the flow depicted in Figure 2.7. First, the application is compiled, then it is profiled to obtain information about the basic blocks that were executed. With this information, the Control Flow Graph (CFG) (explained in detail next) of the application is generated, then it is analyzed in order to determine which are the best spots to insert power gating instructions (always placed at the beginning of the basic block). Finally, the application is executed once more with the power gating instructions in order to evaluate the energy savings and the incurred performance overhead.

The CFG that is built based on the application profiling (i.e., considering conditional branch and loop information) is a data structure that comprises the transitions probabilities to the other basic blocks and the number of idle cycles of each basic block. This is done for each functional unit. An example of CFG is depicted in Figure 2.8, in which the transition probability from basic block B_2 to B_3 is of 70%, and B_3 to B_4 is 90%. Therefore, the expected number of idle cycles starting at a given basic block is defined by the weighted sum considering the number of idle cycles and the transition probability. In the example, equation (2.1) is obtained for such functional unit.

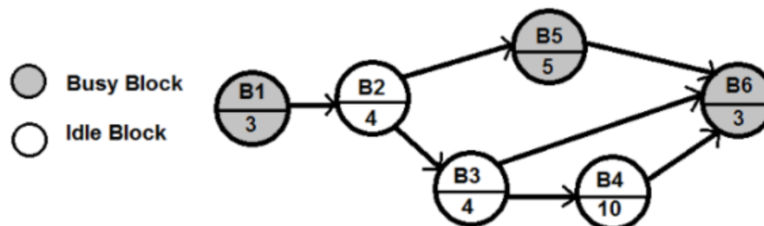
$$T = 4 + (0.7 * 4) + (0.7 * 0.9 * 10) = 13.1 \tag{2.1}$$

Figure 2.7: Methodology Flow of the Compiler-directed Power Gating



Source: (GIRALDO; WONG; BECK, 2016)

Figure 2.8: Example of Control Flow Graph for a Given Functional Unit



Source: (GIRALDO; WONG; BECK, 2016)

This value represents the expected number of idle cycles that a power gating instruction could exploit if it was to be inserted in the given basic block. The authors consider the break-even point to be of 10 cycles (HU et al., 2004). Hence, in this example, a power gating instruction would be added at the beginning of the basic block B_2, as the estimated number of idle cycles is 13.

In addition, the wake-up time is of 3 cycles for this technology. This means that after starting a basic block in which a functional unit must be turned on, this unit will take three cycles to be completely active, which implies that an instruction at the beginning of the basic block that needs this functional unit will have to wait it become ready. On the other hand, if the instructions at the beginning of the basic block do not use the functional unit that is being activated, the execution may continue normally (i.e., there is no need to stall the processor if all functional units that will be used are already activated).

For the register file power gating, the application is profiled and the register file is divided into eight groups of 8 registers each. Based on the profiling, portions of the

register file are shut down at the beginning of the application, turning off the blocks that are never used.

2.2.4 Energy-aware Fault Tolerance

In this subsection, works that combine energy optimization mechanisms with fault tolerance to reduce the energy consumption of the protected circuit will be discussed. Fault tolerance techniques usually increase the energy consumption of the application by increasing the power dissipation of the circuit (spatial redundancy) or the execution time (time redundancy). In addition, when using DVS or DVFS to reduce the energy consumption, the circuit becomes less reliable as the voltage is reduced. Therefore, the number of SEUs is increased exponentially as voltage decreases (HAZUCHA; SVENSSON; WENDER, 2000; HAZUCHA; SVENSSON, 2000; ZHU; MELHEM; MOSSÉ, 2004).

Pop et al. (2007) propose to increase fault tolerance and minimize the energy consumption in a no-fault scenario by scheduling and applying voltage scaling on a heterogeneous distributed time-triggered system, in which the processes are statically scheduled. The transient faults are tolerated by re-executing the process.

Oh and McCluskey (2002) combine procedure call duplication with statement duplication to reduce the energy consumption overhead when compared to the duplication of every program instruction. This approach is only able to detect errors and it reduces the energy consumption, on average, by 25% depending on the required detection latency when compared to the full instruction duplication.

2.3 Performance

Performance may be exploited at different levels: the most common are at instruction (ILP) or thread (TLP) level. Next, we will discuss in detail these two parallelism exploitation granularities (for both VLIW and superscalar processors).

2.3.1 ILP and TLP exploitation

2.3.1.1 Instruction-Level Parallelism (ILP)

ILP is exploited by analyzing the instructions that have to be executed, the ones that are independent of each other (i.e., a given instruction that does not depend on the

result from the other) can be executed concurrently if there are available functional units. The identification of such instructions can be performed during run-time or compile time. The former is adopted by superscalar processors and the latter by VLIW ones.

Superscalar processors use dependency-checking mechanisms, instruction queue, reorder buffer and other hardware components to exploit ILP during run-time. Two execution paradigms may be applied to the instructions: in-order and out-of-order (HENNESSY; PATTERSON, 2017).

- In-order processors will exploit ILP by executing instructions in parallel, however, if a given instruction depends on an instruction that is currently being executed, the following instructions in the instruction queue must wait the former instruction start its execution so they can be issued to the available functional units.
- Out-of-order processors allow the instructions in the instruction queue to be issued as soon as their operands are available and there is an available functional unit to execute the given operation. Therefore, they are not executed in the compiled order. Out-of-order execution allows the performance to be improved (when compared to in-order execution) as more instructions are considered to be executed at a given time, exploiting the functional units more efficiently. However, the cost of this flexibility comes with increased power dissipation and area overhead for control mechanisms.

On the other hand, VLIW processors exploit ILP by means of a compiler, executing several operations (instructions) per cycle depending on the processor's issue-width and the intrinsic ILP available in the application. These instructions are organized into words (bundles), and all instructions in a bundle are executed in parallel. VLIW processors occupy less area and dissipate less power when compared to traditional superscalar processors, since the process of scheduling instructions is statically done by a compiler. Therefore, the hardware of a VLIW processor is much simpler: the instruction queue, reorder buffer, dependency-checking and many other hardware components are not needed. However, in several cases, it is not able to fill all slots of the bundle with independent instructions (ADITYA; MAHLKE; RAU, 2000). The solution is filling the unused slots with NOPs. These NOPs require memory bandwidth to be fetched, potentially increasing cache misses, which would result in performance degradation and extra energy consumption.

In order to amortize such costs, several techniques have been proposed to remove these NOPs: compressed encoding for VLIW instruction (TREMBLAY et al., 2000;

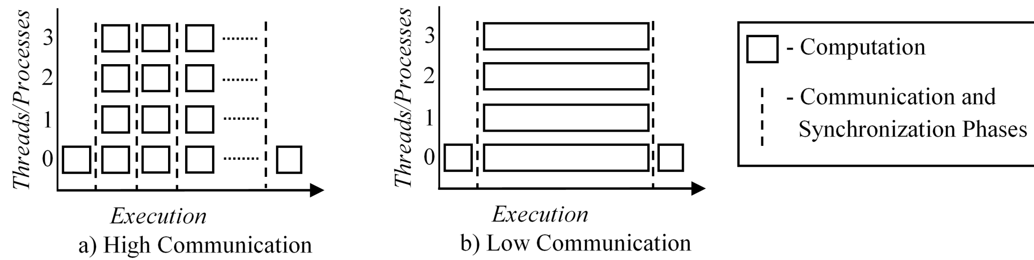
COLWELL et al., 1991; CONTE et al., 1996; JEE; PALANIAPPAN, 2002); instruction template bits (SHARANGPANI; ARORA, 2000; WAERDT et al., 2005); and stop-bits (FISHER; FARABOSCHI; YOUNG, 2005; RAJE; SIU, 1999; SUGA; MATSUNAMI, 2000; HUBENER et al., 2014). Even so, the functional units of the issue slot responsible for executing the NOP (whether it was removed from code or not) will still be idle, which potentially allows this idle hardware to be exploited in a more efficient way: executing duplicated instructions for fault tolerance, shutting down these functional units to save energy, or executing another thread to improve performance.

2.3.1.2 Multiple Applications and Thread-Level Parallelism (TLP)

Multicore architectures exploit parallelism by executing several applications or several threads from the same application concurrently. Therefore, improving the performance of the system or application. In order to speed up the development of parallel applications, Parallel Programming Interfaces (PPIs) are used, such as, Open Multi Processing (OpenMP) (CHAPMAN; JOST; Van Der Pas, 2008), POSIX threads (Pthreads) (BUTENHOF, 1997), or Message Passing Interface (MPI) (SNIR, 1998). Each of these PPIs have different characteristics regarding the management of the threads, distribution of the workload and synchronization. For instance, OpenMP and Pthreads use shared variables in memory for data exchange, and MPI uses message passing. OpenMP offers a set of compiler directives, library functions, and environment variables for the development of multithreaded programs. Pthreads offers functions that allow a fine-grain adjustment of the workload, and the creation/termination of the threads, distribution of the workload and control of the execution are entirely defined by the programmer (BUTENHOF, 1997).

OpenMP synchronizes the threads using busy-waiting (i.e., the threads access the shared memory repeatedly until the end of synchronization). On the other hand, Pthreads synchronizes the threads by blocking them with mutexes, therefore, waiting on standby until the end of the synchronization (TANENBAUM; WOODHULL, 1987). MPI offers communication primitives (e.g., send and receive) to exchange data. The creation and termination of the processes may be dynamic or static at the beginning and end of the execution, depending on the MPI version that is used. In multicore environments, MPI communications are implemented using shared memory regions to store First-In-First-Out (FIFO) queues of each MPI process (CHANDRAMOWLISHWARAN; KNOBE; VUDUC, 2010). Therefore, TLP exploitation is not transparent for the developer and

Figure 2.9: Multithreaded Application Behavior Regarding Communication



Source: (LORENZON; CERA; BECK, 2016)

the performance improvements greatly depend on the characteristics of the application, target architecture, and chosen PPI.

Figure 2.9 depicts different communication behaviors that multithreaded applications may present: applications with low communication requirements (i.e., CPU-intensive applications that distribute the workload at the beginning of the execution and in the end join the results of the threads) present a speedup close to the ideal (Figure 2.9(a)), which is the reduction of the execution time by a factor close to the number of available cores. On the other hand, high communication applications constantly exchange data between the threads during the execution (Figure 2.9(b)), which implies in several synchronization points, consequently reducing the speedup of the parallel application (LORENZON; CERA; BECK, 2016). In order to quantify the amount of concurrency that an application has, the TLP metric proposed by Gao et al. (2014) is presented in (2.2).

$$TLP = \frac{\sum_{i=1}^{n} c_i i}{1 - c_0} \tag{2.2}$$

In which, c_i represents the fraction of time that *i* cores are concurrently running different threads, *n* is the number of cores, and $1 - c_0$ is the non-idle time fraction. The closer this value is to the total number of threads, the more TLP is available.

2.3.2 Adaptive Multi-core Architectures

Traditional core micro-architectures are not able to efficiently exploit the TLP available in the applications. Most commercial processors have large out-of-order cores (Intel Core i7) or small cores (ARM A15/A7). Large Out-of-Order (OoO) cores provide

high performance for single threaded programs by exploiting ILP, however, they are extremely power-inefficient for TLP exploitation due to the complex OoO cores. On the other hand, small cores are able to exploit the TLP without wasting energy and area at the cost of reduced single-thread performance.

In order to cope with high ILP and TLP programs, heterogeneous chip multiprocessors have been proposed (HILL; MARTY, 2008; MORAD et al., 2006; SULEMAN et al., 2009). These processors provide a few large cores for single-thread performance and many small cores for multithreaded applications. However, the number of cores of each type must be chosen during design time, which restricts the ability of the core to adapt itself for different applications that do not fit the pre-determined number of cores, resulting in sub-optimal performance and energy consumption.

Processor architectures with the objective of overcoming the drawbacks of heterogeneous cores were also proposed (BOYER; TARJAN; SKADRON, 2010; GIBSON; WOOD, 2010; GUPTA et al., 2010; IPEK et al., 2007; KIM et al., 2007; PRICOPI; MITRA, 2012; PUTNAM; SMITH; BURGER, 2011; WATANABE; DAVIS; WOOD, 2010). In these approaches, several small cores are used to provide high throughput to multi-threaded programs and these cores are "fused" into a large core when single thread performance is needed.

MorphCore (KHUBAIB et al., 2012) follows the same idea of adjusting the number of cores during run-time, but instead of fusing small cores, it uses a large out-of-order core that is able to "morph" into several small cores for multi-thread performance. This project choice was made in order to avoid additional latencies in the pipeline stages and to avoid the data migration among the caches when switching modes. The application starts running in the large OoO core and the mode switching is based on the number of active threads. Hence, when the number of active threads increases, the core switches to in-order mode (switching back to OoO mode if the number of active threads is reduced). As the mode switching mechanism is implemented in hardware, the operating system does not require any modification.

2.4 Fault Tolerance, Energy Consumption, and Performance Trade-off

A few works have proposed to target all three axes: Tricriteria Scheduling Heuristics (TSH) (ASSAYAD; GIRAULT; KALLA, 2011) proposes an offline scheduling heuristic that produces a static multiprocessor schedule, based on a given application graph and

Table 2.1: Core Parameters

Core Configuration Mode	F(GHz) / V(Volt)	Buffer Size (IQ, LSQ, ROB)	Width (Fetch, Issue)
CCMA (Avg. core)	1.6/0.8	36,128,128	4,4
CCMN (Narrow)	2/1	24,64,64	2,2
CCML (Large)	1.4/0.8	48,128,256	4,4
CCMS (Small)	1.2/0.7	12,16,16	1,1

Source: (SRINIVASAN; KOREN; KUNDU, 2015)

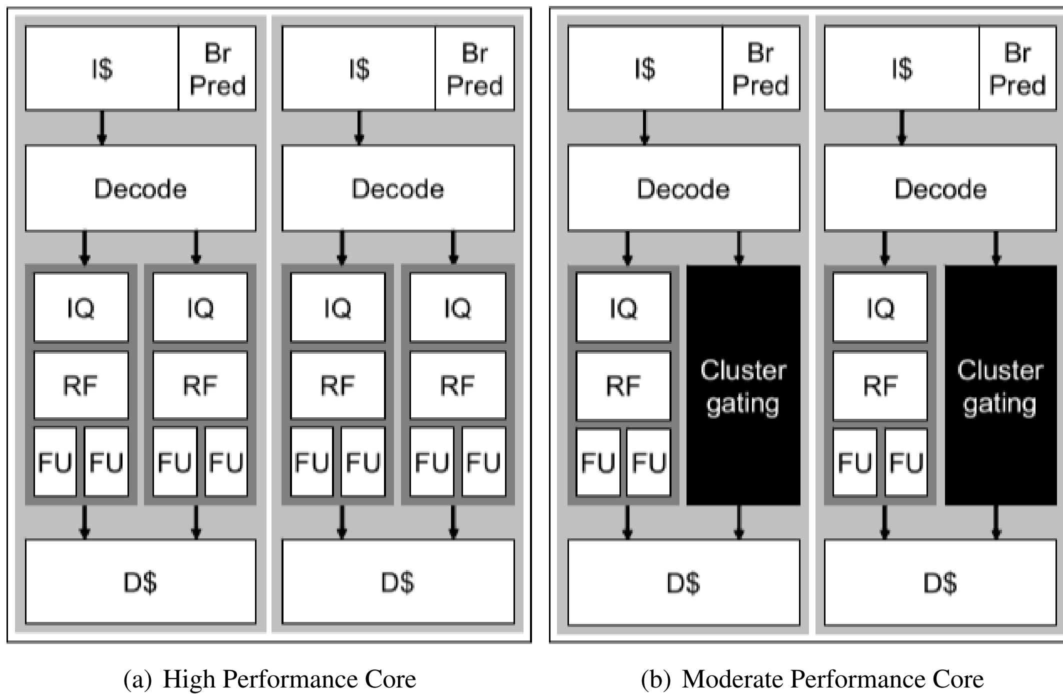
a given multiprocessor architecture (homogeneous and fully connected). In order to increase reliability, the instructions are replicated; and to reduce the energy consumption DVFS is applied. A greedy scheduling algorithm takes as input the application and architecture graphs, power and reliability constraints, and the execution time of the operation considering the maximum frequency to meet the reliability and energy requirements and minimize the schedule length based on the aforementioned techniques.

Srinivasan, Koren and Kundu (2015) propose a multi-objective strategy to choose the best core type considering power efficiency and reliability. Four Alpha processor core configurations are considered, varying voltage; frequency; size of the instruction queue, load-store queue and reorder buffer; and fetch and issue widths, these configurations are presented in Table 2.1. Hardware counters are used to estimate the power dissipation and the Architectural Vulnerability Factor (AVF) (discussed in detail later) of the processor, and with this data, a Cobb-Douglas production function is applied to choose the best core for a given part of the application. Even though reliability is considered, no solution to protect the cores (i.e., fault tolerance mechanism) is proposed.

Ramírez et al. (2012) present a high-level reconfiguration approach that, based on user-defined constraints, changes the configuration of a heterogeneous multicore processor. Their approach is heavily based on the profiling of the applications before the execution, which must be done for all different processor frequencies in a heterogeneous processor. At every (pre-defined) number of cycles, a reconfiguration is triggered. The reconfiguration engine receives as input the defined reliability level, power budget, and performance counters, then, it defines what will be the frequency and voltage, and if the Error Correction Code (ECC) and the L2 cache should be enabled. The ECC is used to provide additional reliability and the L2 cache can be disabled to save power. Therefore, this approach relies on static profiling of the application, limiting the dynamic adaptation.

Sato and Funaki (2008) investigate the trade-off between the aforementioned axes in a Multiple Clustered Core Processor (MCCP). In order to modify the organization of

Figure 2.10: Evaluated Cluster Configurations



(a) High Performance Core (b) Moderate Performance Core

Source: (SATO; FUNAKI, 2008)

the processor during runtime, some clusters are turned off, therefore, allowing to change between high-performance (2-issue dual core) and moderate performance (single issue dual core) processors (both superscalar processors are homogeneous and are depicted in Figure 2.10). To improve fault tolerance, Redundant Multithreading (RMT) technique is used, in which the threads of the application are duplicated and compared. Their contribution is to choose which configuration will be applied to the next part of the application, which is done solely based on the past Instructions Per Cycle (IPC). That is, based on the IPC of the last sample window, they try to predict the future IPC. Therefore, the power and reliability do not influence the decision mechanism, only performance is considered, meaning that these axes will be improved only by those phases in which the moderate performance core is chosen (because it is more reliable and consumes less energy, according to the authors). The techniques discussed in this subsection will be further compared to the ones that are being proposed in this work in Section 2.5.4.

2.5 Contributions of this Thesis

As aforementioned, the goal of this work is to implement an adaptive and polymorphic processor designed to dynamically trade-off performance, fault tolerance, and

energy consumption. In order to do so, the following methodology was adopted: for each axis, techniques to optimize the given axis were evaluated, identifying the most appropriate ones considering the incurred overhead in the other axes and its ability to optimize the axis on focus, which is discussed next.

2.5.1 Fault Tolerance

For this axis, the techniques were evaluated considering the power dissipation and performance overhead, ability to be completely dynamic, and do not change the binary code. Next, each of these characteristics will be discussed.

In order to achieve the best trade-off between the axes, we need to choose techniques that interfere the least as possible in the other axes to maximize the benefits of applying each of the chosen approaches. In addition, the chosen mechanism must be completely dynamic, as the whole processor design that is proposed in the scope of this work is adaptive, which eliminates the need for pre-processing. Finally, modifying and recompiling the binary code may not be a trivial task, leading to incompatibility with future processors and losing backward compatibility. Thus, the chosen mechanisms must comprise all these characteristics. With this in mind, a duplication with instruction rollback implemented in hardware was chosen and three variations of this mechanism were implemented and compared.

2.5.2 Energy Consumption

As previously discussed, DVFS reduces the energy consumption, but it also reduces the reliability of the system in addition to reducing performance. Clock gating is a lightweight technique that can be applied to reduce dynamic energy consumption, however, the circuit still dissipates leakage power when clock gated. On the other hand, power gating is able to reduce both static and dynamic energy consumption with low performance overhead and not affecting the reliability of the system. Therefore, we chose to implement a power gating mechanism for the energy optimization axis. As aforementioned, Giraldo, Wong and Beck (2016) propose to apply power gating to the Functional Units (FUs) and the register file of the ρ-VEX processor at compile time. On the other hand, we combined power gating with a dynamic program phase detection to identify, during run-time, the best spots to apply this technique to the FUs. We have considered

the same overheads for the power gating mechanism as Giraldo, Wong and Beck (2016) and, although the energy savings between these two mechanisms are similar, the exact difference cannot be estimated because the chosen benchmarks are not the same.

2.5.3 Performance and Learning

Considering the performance axis, an ILP control module was developed to create free slots by splitting bundles into two cycles when more fault tolerance or a power gating phase needs to be extended. Also, a learning module was integrated into the dynamic version of the ρ-VEX processor, which tests and chooses the best hardware configuration to execute a given application based on its behavior.

The current implementation of the learning module focuses on multiple (single-threaded) applications, as the ρ-VEX currently does not have support for multi-threaded applications. However, support for OpenMP is currently being implemented, therefore, as future work, we aim to extend the proposed polymorphic processor to cope with multi-threaded applications as well.

2.5.4 Fault Tolerance, Energy Consumption, and Performance Trade-off

There are several differences when comparing the approaches that trade-off these three axes to the adaptive processor that is being proposed in this work. First, instead of a static scheduling heuristic, an adaptive design is being proposed, which is able to dynamically balance the axes of performance, fault tolerance, and energy consumption without relying on any profiling of the application prior to the execution. In addition, lightweight fault tolerance mechanisms are applied to the proposed processor in order to efficiently improve the reliability of the processor, instead of using the intrinsic characteristics of the cores (no fault tolerance mechanism) or techniques that consume at least two times more energy (RMT).

Differently from previous work, in the polymorphic processor, we propose to use a metric that comprises the axes of fault tolerance, energy, and performance to decide which configuration of the processor should be used, instead of using only performance metrics, such as Sato and Funaki (2008). Moreover, several heterogeneous configurations are available and evaluated during runtime in the proposed processor. Finally, the proposed work has the goal of balancing these three axes by exploiting the same hardware components (i.e., the issue-slots) in each of the optimization mechanisms.

3 IMPLEMENTED TECHNIQUES

This chapter presents the details of each technique that was implemented in this work considering the axes of fault tolerance, energy consumption, and performance.

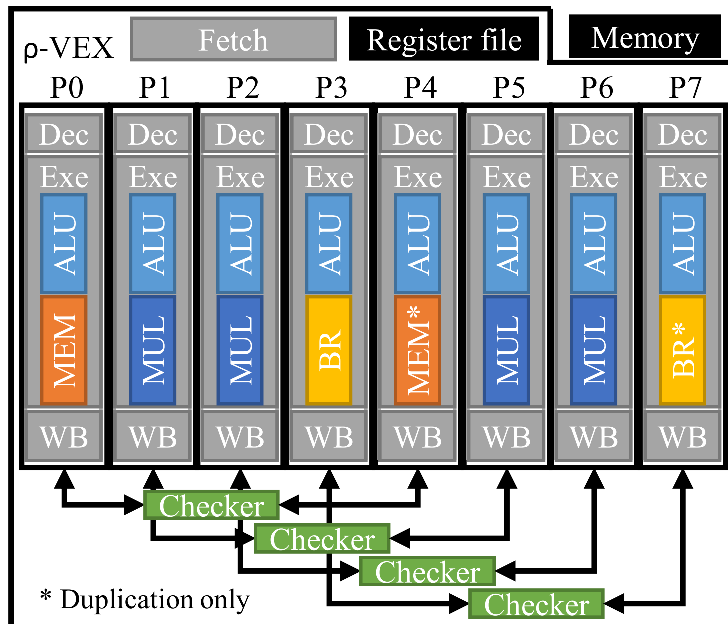
3.1 Fault Tolerance

Next, we will discuss the fault tolerance techniques that were implemented. In this section, only fault tolerance mechanisms will be considered, so the idle hardware will be used for that purpose. For that, as already mentioned, a hardware-based instruction duplication with rollback was implemented, which varies its behavior according to the system requirements and application. In order to evaluate the trade-offs regarding area, power dissipation, and performance, two different approaches were proposed, and a third approach is discussed in Appendix A (which was an initial implementation that was not used in neither versions of the proposed processor). First, the general implementation of the instruction duplication will be explained since it is common to the three techniques; followed by the discussion of the specifics of each one.

3.1.1 Basic Duplication with Rollback

The fault-tolerant implementation of the ρ-VEX is depicted in Figure 3.1. The pipelanes are numbered from P_0 to P_7; *Dec* stands for the decode stage, *Exe* for the execution (two cycles), and *WB* for the write-back stage. To keep the overhead low (area and delay), the duplication pairs are placed in the following order: *pipelane 0* with *pipelane 4*, *pipelane 1* with *pipelane 5*, and so on. Therefore, the issue-slots are combined in a way that the first four pipelanes are compared with the four last ones. This approach efficiently exploits the scheduling mechanism of the HP VEX compiler, which always schedules the instructions starting from the lower issue-slots (from *0* to *7*). An additional memory and branch unit must be added to the core to allow duplication of all instructions. These two specific extra functional units are used only to execute duplicated instructions (i.e., they cannot be used for regular instructions), since the ρ-VEX does not support more than one memory or branch operations per cycle. For the sake of comparison, a fault-tolerant 4-issue version was implemented: it has full duplication (i.e., all pipelanes are duplicated in hardware), so it is also composed of eight pipelanes, as depicted in Figure 3.2. Each

Figure 3.1: Fault Tolerance Implementation of the ρ-VEX Processor



Source: The Author

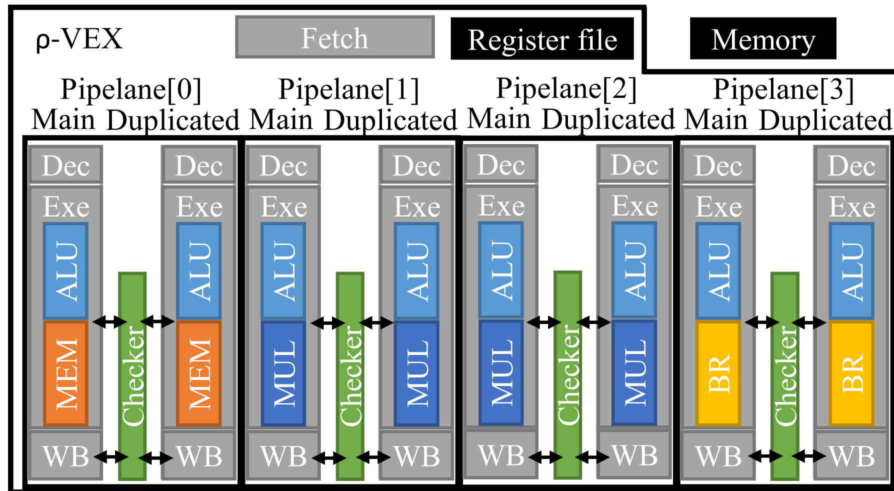
duplicated pipelane executes the same instructions than its regular counterpart with the full duplication approach.

A checker compares the results (i.e., all output signals) of the pipelanes that are executing duplicated instructions (e.g., arithmetic operations, jump address of a branch, or the values of a memory operation are checked) to detect errors. The destination register, the register file's and memory's write enable signals are also compared. When an error is detected, a rollback mechanism is triggered to correct it by executing the last uncommitted bundle again. The Program Counter (PC) for the rollback is stored in a register and, in the case of an error, this stored PC overwrites the current PC, this mechanism is depicted in Figure 3.3.

As the memory and register file were not modified in the meantime (between the rollback PC and the current PC), the pipeline is simply flushed, and the writing to the memory and register file are blocked, avoiding memory corruption. Once the rollback PC is loaded, the instruction corresponding to that PC is fetched again, and the execution resumes from that point. Both the checkers and the rollback mechanism do not affect the critical path of the processor, as they operate in parallel to the rest of the processor. The memory and the register file are considered to be protected with ECC.

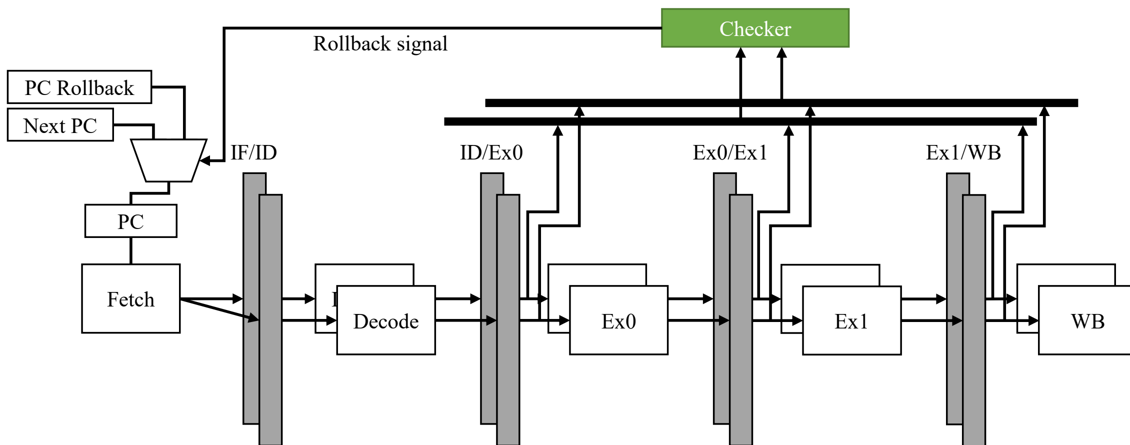
This approach has a fixed cost of five cycles to refill the pipeline, which is negligible considering the total number of cycles of an application and that this cost is only paid in case of an error. An approach to only flush the faulty instruction and the following

Figure 3.2: Full Duplication Configuration for the 4-issue ρ-VEX Processor



Source: The Author

Figure 3.3: Rollback Mechanism



Source: The Author

ones could be implemented, thus, the performance cost would be variable (from 3 to 5 cycles). However, more control logic to identify the faulty instruction and guarantee that it does not affect the other instructions would imply in more area and power dissipation. As the proposed work aims at providing the best trade-off between several axes, the current fixed cost approach (used only in case of an error) is likely to be more cost effective comparing to the zero to two-cycle overhead, when considering performance, area, and power altogether. In addition, the application does not have to be modified at all, as all the proposed techniques were implemented in hardware. Hence, any compiler that supports the VEX instruction set architecture may be used to compile the applications (e.g., HP VEX compiler, GNU Compiler Collection (GCC) VEX, and others).

Each of the three proposed methods in the fault tolerance axis is suitable for different system requirements, considering area, performance and power dissipation. Next,

each one will be discussed.

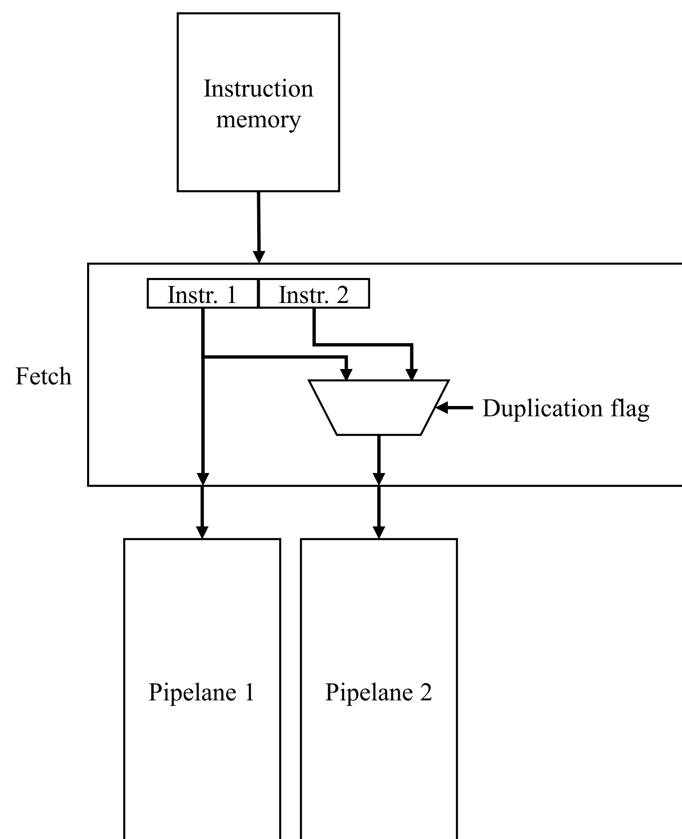
3.1.1.1 Spatial duplication

In this technique, the idle pipelanes are used to execute duplicated instructions when possible (i.e., when there are NOPs). Therefore, the verification is done on a per cycle basis. After fetching an instruction word, each pipelane receives one instruction, for decoding and further execution. We have modified this process so that the pipelane receives the program instruction (no duplication), or the instruction from another pipelane (when a NOP is found, it is replaced with a duplicated instruction).

No additional accesses to the memory are required for both instructions and data: instruction words are fetched (one access), then the bundles are divided into the pipelanes (applying duplication when possible), as depicted in Figure 3.4. The memory and the process of transferring the instructions to the pipelanes are considered to be ECC protected. For data memory accesses, the memory is also accessed once. For data loads, the result is divided into the pipelanes that contain memory units (performing duplication or not depending on the duplication flag) (Figure 3.5(a)), and for data stores, the result and address are first compared by the checker, then, if both match, they are transferred to the memory (Figure 3.5(b)). As the whole process is dynamic, this approach is completely transparent to the application.

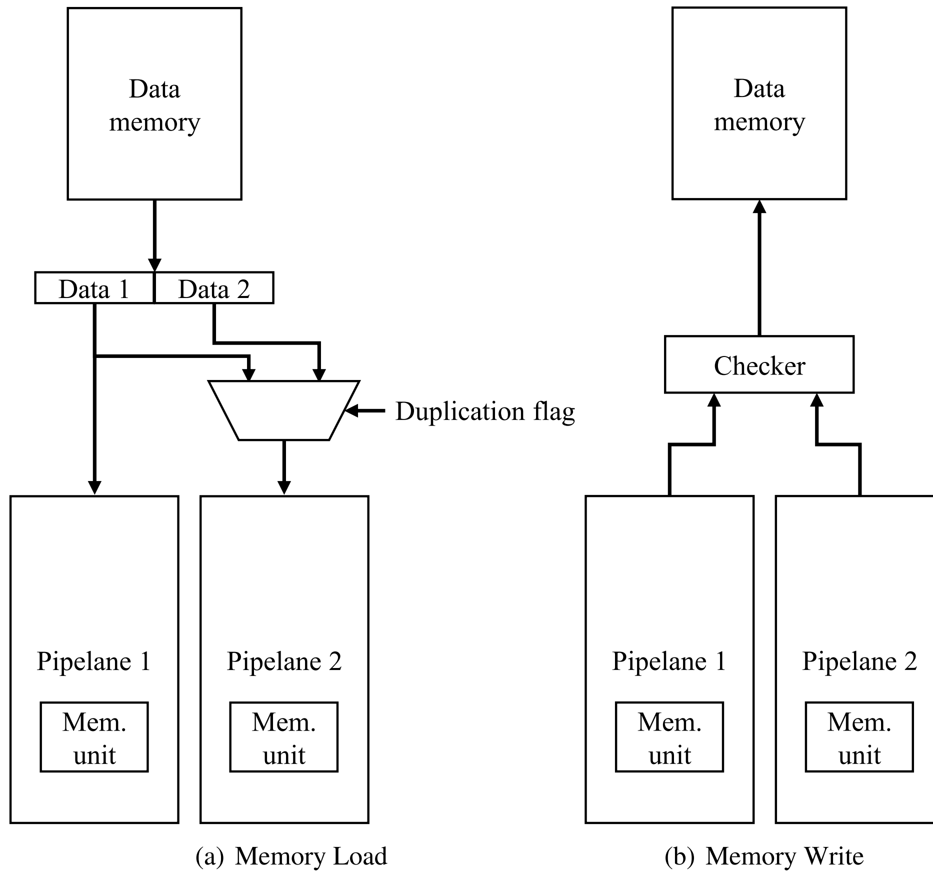
An example of code execution comparing the original (unprotected) 8-issue version with the spatial duplication approach is presented in Figure 3.6(a) and Figure 3.6(b). In this figure, the C_1 to C_n depict the cycle *1* to *n* of the execution, and the I_x the program instructions that are being executed. Note that there are NOPs in some bundles due to the lack of parallelism, which will waste energy to be executed and will not provide any meaningful computation. This mechanism will exploit those NOPs to execute duplicated instructions, increasing the reliability and not affecting performance, as all instructions of a VLIW bundle are executed in parallel.

Figure 3.4: Instruction Memory Access and Instruction Duplication



Source: The Author

Figure 3.5: Data Memory Access and Instruction Duplication



(a) Memory Load (b) Memory Write

Source: The Author

Figure 3.6: Code Execution Example

Program instructions | Duplicated instructions

	P0	P1	P2	P3	P4	P5	P6	P7
C_0	I0	I1	I2	I3	NOP	NOP	NOP	NOP
C_1	I0	I1	I2	NOP	NOP	NOP	NOP	NOP
C_2	I0	I1	I2	I3	I4	NOP	NOP	NOP
C_3	I0	I1	I2	I3	I4	I5	I6	NOP
C_4	I0	I1	NOP	NOP	NOP	NOP	NOP	NOP
C_5	I0	I1	I2	I3	I4	I5	I6	I7

(a) Unprotected 8-issue VLIW Processor

	P0	P1	P2	P3	P4	P5	P6	P7
C_0	I0	I1	I2	I3	I0	I1	I2	I3
C_1	I0	I1	I2	NOP	I0	I1	I2	NOP
C_2	I0	I1	I2	I3	I4	I1	I2	I3
C_3	I0	I1	I2	I3	I4	I5	I6	I3
C_4	I0	I1	NOP	NOP	I0	I1	NOP	NOP
C_5	I0	I1	I2	I3	I4	I5	I6	I7

(b) Spatial Duplication

Source: The Author

3.1.1.2 Temporal and Spatial Duplication

This technique takes the former one step further, allowing temporal duplication to be applied when spatial duplication is not able to duplicate all instructions in the bundle. An example on which temporal duplication is able to improve the reliability by duplicating more instructions is depicted in Figure 3.7. In Figure 3.7(b), the instructions I_4, I_5, I_8, *and* I_9 are duplicated one cycle later than the original bundle. In order to maintain the consistency of the register file and the memory, the results are only committed after all instructions of a given bundle are checked. That is, after the duplicated instruction is executed and its result is compared to the one from the original instruction, or after verifying that those instructions will not be duplicated because the buffer is full, which is explained next.

Figure 3.8 depicts the additional modules for this technique, which includes two buffers: the issue buffer and the commit buffer. The issue buffer is responsible for storing those instructions that could not be duplicated when only spatial duplication was applied. For instance, instructions I_4, I_5, I_8, *and* I_9 from Figure 3.7(b). In this case, the pending (duplicated) instructions are kept in the buffer until there is a free slot so they can be scheduled. The commit buffer stores the bundles which still have pending instructions to be executed and checked. Once all instructions from a given bundle are verified, the bundle is committed. The buffer size can be configured during design time and it may be modified during runtime when the polymorphic and adaptive version of the ρ-VEX is used, which will be explained in detail in Chapter 4.

The instruction scheduler first applies spatial duplication when each bundle is going to be executed. If there are instructions that could not be duplicated because there were not enough empty slots, these instructions will be stored in the buffers for temporal duplication. On the other hand, if there are still idle slots, the instruction scheduler will get pending instructions from the buffer and fill the bundle.

When the buffer is full, the temporal duplication is not applied to the next instructions until there is space in the buffer. Therefore, those instructions that cannot be stored in the buffer will be committed without verification. The temporal duplication also exploits the memory latency to execute duplicated instructions; whenever there is a cache miss, the instruction scheduler will start using those idle cycles to schedule the instructions that were waiting in the buffer. In the remaining cycles of the cache miss, the core is power gated in order to reduce the energy consumption while the processor is waiting for

Figure 3.7: Temporal and Spatial Duplication Execution Example

Program instr.	Spatial dup.	Temporal dup.

	P0	P1	P2	P3	P4	P5	P6	P7
C_1	I0	I1	I2	I3	NOP	NOP	NOP	NOP
C_2	I4	I5	I6	I7	I8	I9	NOP	NOP
C_3	I10	I11	NOP	NOP	NOP	NOP	NOP	NOP

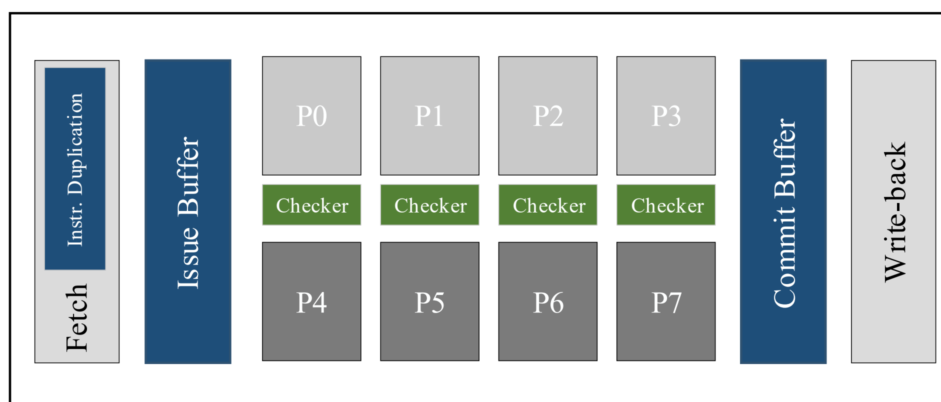
(a) Unprotected Processor

	P0	P1	P2	P3	P4	P5	P6	P7
C_1	I0	I1	I2	I3	I0	I1	I2	I3
C_2	I4	I5	I6	I7	I8	I9	I6	I7
C_3	I10	I11	I4	I5	I10	I11	I8	I9

(b) Temporal and Spatial Duplication

Source: The Author

Figure 3.8: Temporal and Spatial Duplication Core Overview



Source: The Author

the memory. A similar approach that applies power gating while the processor is stalled on a long memory access is presented by Jeong et al. (2012).

In order to increase the flexibility for the scheduling of the duplicated instructions, we have added extra functional units in each pipelane, which means that each pipelane is able to execute any instruction: memory, control, or logic. The extra cost of these functional units are taken into account and the area overhead is further discussed in the results section. As the overhead of adding these extra units proved to be low, we decided to maintain the flexibility of scheduling any instruction to any pipelane, instead of increasing the control logic to verify which instruction could be scheduled to each pipelane. In addition, a forwarding mechanism was implemented to get the updated values from registers that are in the buffer and were not committed yet. Thus, avoiding processor stalls when such

dependencies are detected.

3.2 Energy Optimization

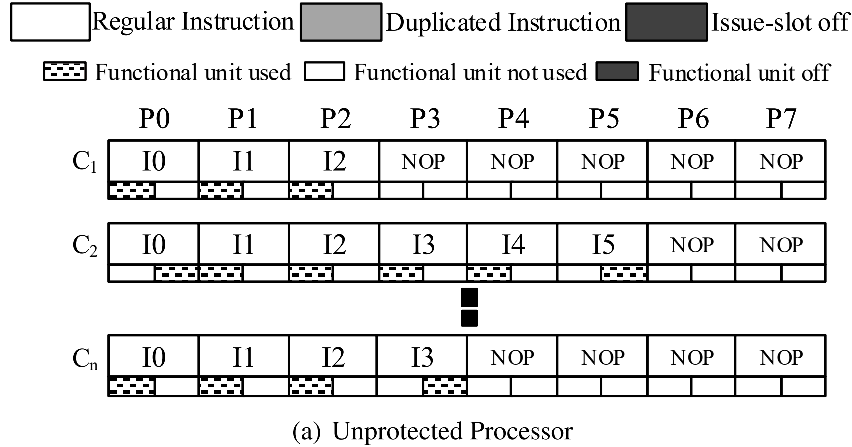
As previously discussed in Chapter 2, clock gating is not able to reduce the static power of the processor. Therefore, we have chosen to implement a Power Gating (PG) mechanism so both dynamic and static energy can be reduced. In order to not increase the design space even further, a DVFS mechanism, which affects the reliability of the circuit when low voltages are applied, will be evaluated as future work.

The PG is applied to the functional units of the processor. When all functional units from a given pipelane are turned off, the whole pipelane is also turned off to reduce energy consumption even more. To cope with the additional cost of the PG mechanism, the module that is being power gated must remain shut down for at least the break-even time (as discussed in detail in Chapter 2). The break-even is the time necessary to compensate the additional energy consumption of the gating transistor. After this point, the longer the circuit remains turned off, the more energy will be saved (HU et al., 2004). Considering the technology parameters used in this work, the wake-up time of this transistor is three cycles (HU et al., 2004). Therefore, to minimize these overheads, the power gating mechanism is applied to the granularity of Basic Blocks (BBs) (i.e., a code sequence with no branches in except to the entry; and no branches out except at the exit). Whenever a new BB starts to execute, its instructions are analyzed so it can be evaluated which functional units are required for the execution of that BB. Based on this evaluation, a power gating configuration is saved for that basic block for future reuse (this is controlled by the decision module and will be discussed in detail in Section 4.2).

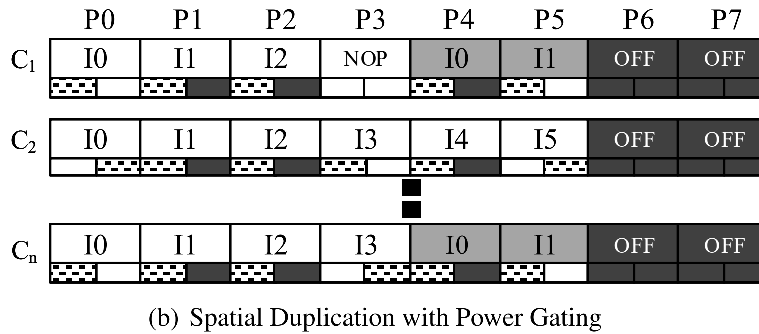
When a given functional unit is turned off during the execution of a BB and the next BB needs this functional unit at its very beginning, it takes three cycles so it is completely active. In this case, the processor has to wait for the FU to be active (stalling the execution). On the other hand, if the FU is only needed after the wake-up time, it is ready by the time the given instruction that needs it starts executing, and there is no performance overhead.

Figure 3.9(a) depicts an example of code execution of a BB on the regular unprotected 8-issue processor. Then, Figure 3.9(b) presents the use of those functional units that were idle to execute duplicated instructions in the pipelanes P_4 and P_5 (increasing fault tolerance) and turn off the idle pipelanes P_6 and P_7 (saving energy). In the case of

Figure 3.9: Basic Block Execution Example



(a) Unprotected Processor



(b) Spatial Duplication with Power Gating

Source: The Author

pipelanes P_4 and P_5, at least one functional unit is used in this BB (at C_2), so the whole pipelane could not be shut down, due to the power gating costs. However, pipelanes P_6 and P_7 may be completely turned off as only NOPs are executed during the whole BB.

3.3 Performance

In the performance axis, we evaluate two techniques: an ILP control mechanism that was developed to artificially create empty slots so a given PG phase is maximized, and the exploitation of the dynamic behavior of the ρ-VEX processor to change the hardware configuration so the hardware can adapt itself according to the application's requirements. The latter will be further discussed in Chapter 4, in which the proposed polymorphic processor will be presented. Next, the variations of the ILP control module are detailed.

3.3.1 ILP Control

The ILP control mechanism can be used to either reduce the ILP so more instructions can be duplicated (which focus on reliability) or to maximize PG phases that

would be interrupted by a few instructions (which focus on energy optimization). The reliability-oriented and energy-oriented ILP control mechanism adapt the execution based on a threshold, which can be static or dynamic. The former is defined at design time, and the latter (only available for the reliability-oriented mode) is modified during runtime based on reliability requirements defined at design time. Each of these modes has a particular behavior for the threshold and will be explained in detail next.

3.3.1.1 ILP Control for Fault Tolerance

As previously explained, the spatial duplication exploits idle hardware to provide fault tolerance. However, when the VLIW bundle has more than half of the issue-width filled with instructions, the duplication will not be full and may not offer the desired level of fault tolerance, as depicted in Figure 3.6(b) at C_2, C_3, and C_5.

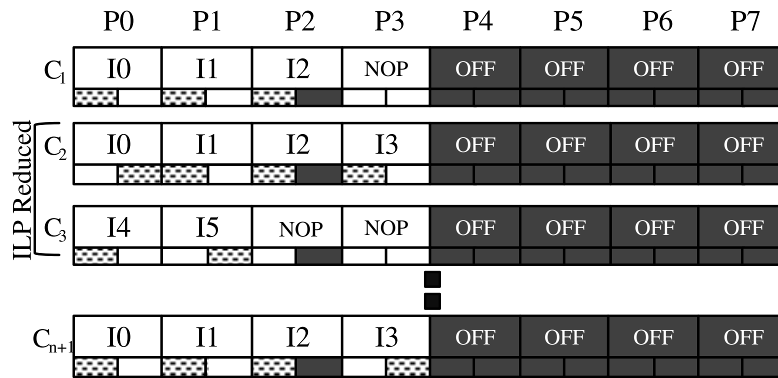
Therefore, proposed technique is able to perform the trade-off between performance and fault tolerance using an ILP threshold. If the ILP in a given moment is high and the application still needs more fault tolerance, this method will reduce the ILP for that purpose. This flexibility comes at a cost in area and power. However, as it will be shown, it is still low when compared to other techniques.

This process is controlled by the decision module (which will be explained in the next subsection) and it can be tuned by configuring the threshold that will activate the ILP reduction. A "utilization value" is calculated at every bundle and changed according to the ILP available in the current bundle. A dedicated hardware is used to calculate this value. When the utilization value reaches the threshold, the current bundle (if it has more than half of the issue-width occupied) is divided into two, so it is possible to apply full duplication to each half of the bundle (trading-off performance for fault tolerance).

3.3.1.2 ILP Control for Energy Optimization

A power gating phase may be interrupted by a few instructions: Figure 3.9(b) shows a BB in which a given pipelane is idle, except for one instruction (i.e., I_4 or I_5 at C_2). In this case, according to the previous discussion on the break-even point, power gating could not be applied to this pipelane, therefore losing significant opportunities for energy savings. The ILP control module (which is controlled by the decision module, explained in the next subsection) splits bundles in a dynamic fashion to increase the length of a potential power gating phase, so more energy savings can be reached.

Figure 3.10: Spatial Duplication with PG and ILP Control



Source: The Author

Let us consider the previous example presented in Figure 3.9(b) again: the pipelanes P_4 and P_5 would remain turned on due to the execution of instructions I_4 and I_5 at C_2 (and therefore they would be used for replication). The presence of the ILP Control Module adds another alternative: by applying the ILP reduction (Figure 3.10), these same pipelanes can now be turned off (since they now reach the break-even constraints), maximizing the power gating phase. Therefore, while the energy consumption and the sensitive area will reduce, it also restricts the duplication of some instructions. It is evident that there is a trade-off in the number of instructions that can be replicated; the pipelanes that can be power gated; and the parallelism that can be lost (and therefore, performance) so even more pipelanes can be turned off. For that, a decision module was developed and it will be described in the next chapter.

4 PROPOSED ADAPTIVE AND POLYMORPHIC PROCESSORS

The proposed adaptive and polymorphic processors are composed of a subset of the aforementioned techniques. The adaptive processor has a decision module that is responsible for choosing which technique will be applied at each phase of the application, based on pre-defined parameters. While the polymorphic processor includes a dynamic optimization algorithm that evaluates and chooses the best processor configuration to trade-off performance, energy consumption, and fault tolerance. Next, the two versions of the ρ-VEX processor will be further explained (static and dynamic), followed by the techniques of the proposed processors.

4.1 ρ-VEX Processor Background

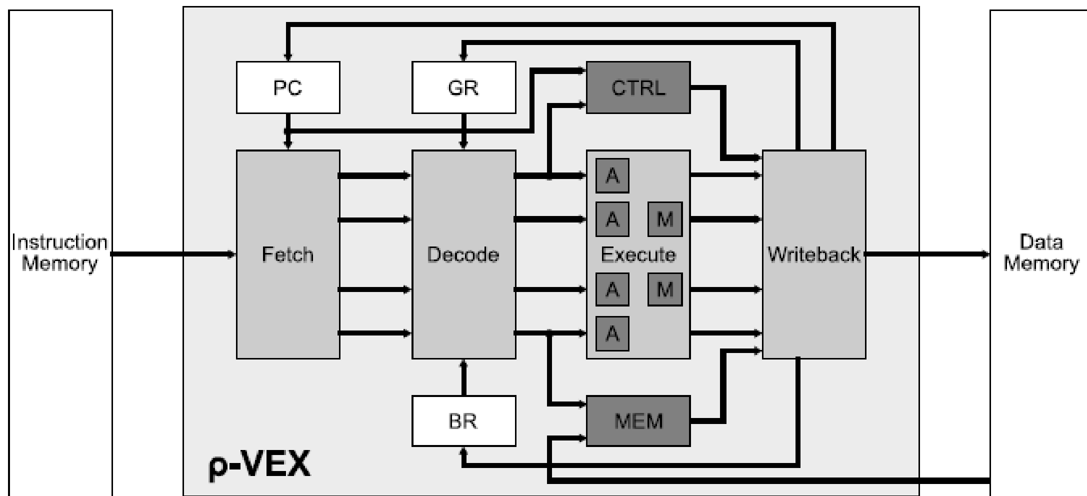
The ρ-VEX processor (WONG; Van As; BROWN, 2008) was chosen as target architecture for the proposed design because it is able to cope with all mechanisms that were implemented. In addition, it allows a detailed simulation and accurate measurements of area, power dissipation, and performance as it is implemented in VHSIC Hardware Description Language (VHDL), which is generally not available (or available with reduced accuracy) on high-level simulators. Each version of the processor is detailed next.

4.1.1 Static ρ-VEX Processor

The static version has a five-stage pipeline, and it can be configured to have a different number of issue slots (e.g., 2, 4, or 8). Each pipelane (issue slot) may contain different functional units from the following set: Arithmetic Logic Unit (ALU) (always present), multiplier, memory, and branch units. An example of a 4-issue organization is depicted in Figure 4.1. The issue slots (pipelanes) of a VLIW processor occupy about 45% of the core total area and the register file (which occupies the rest) can be protected with parity (GAISLER, 1997; MCNAIRY; BHATIA, 2005) or ECC (SLEGEL et al., 1999).

Several compilers may be used to compile VLIW code for the ρ-VEX processor: HP VEX compiler, GCC VEX, LLVM, and others, and the compiler is responsible for scheduling independent instructions to be executed concurrently. In addition, this version of the processor does not have cache memories.

Figure 4.1: Static ρ-VEX organization (4-issue)

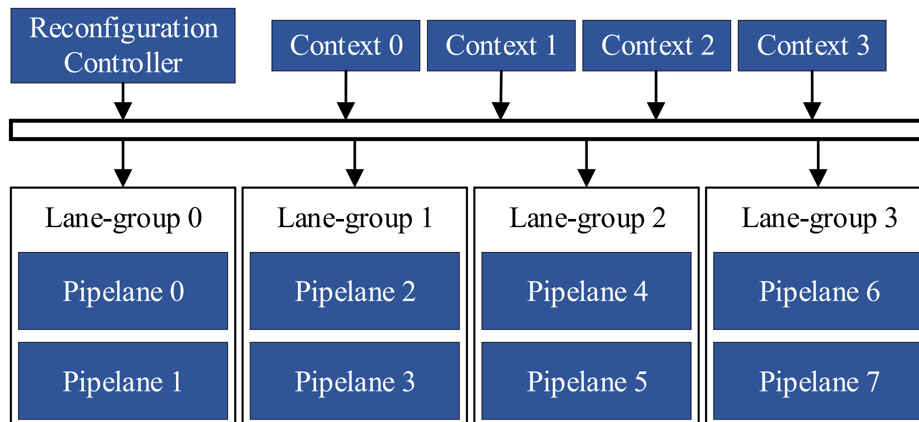


Source: (WONG; Van As; BROWN, 2008)

4.1.2 Dynamic ρ-VEX Processor

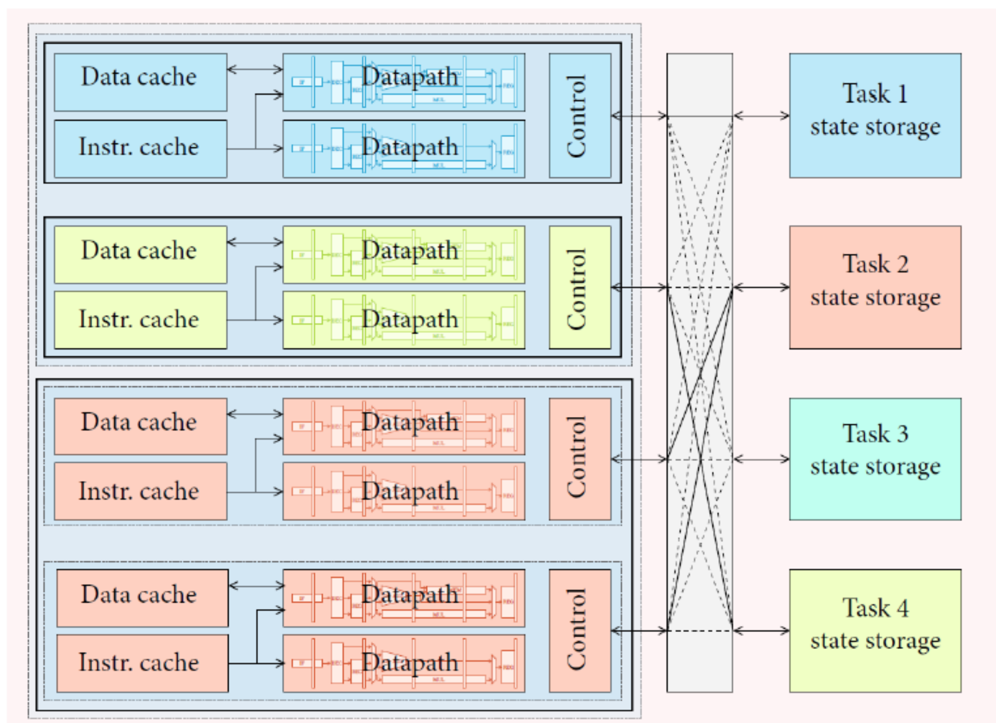
The dynamic version of the ρ-VEX processor (ANJAM; NADEEM; WONG, 2011; BRANDON et al., 2017), takes the static one step further by allowing the code to be executed in different issue-widths during runtime. The processor can be used as a single wide core (8-issue), two medium cores (two 4-issue), or four small cores (four 2-issue). Figure 4.2 depicts the general overview of the processor interconnections: it is composed of four groups, each group containing two pipelanes; the context reflects the actual configuration of the processor, for instance, if the processor is configured to small cores, four contexts (each with a single group) will be used. For medium cores, each of the two contexts will use two groups (i.e., resulting in a 4-issue core for each context), and the large core will execute a single context with four groups (eight pipelanes). Finally, the reconfiguration controller is responsible for switching the configuration of the processor according to a given configuration request that writes to the configuration control register. Figure 4.3 presents the general overview of the ρ-VEX polymorphism. In this example, *Task 1* is running in the 2-issue mode, *Task 2* in 4-issue, *Task 3* is idle and *Task 4* is also running in a 2-issue configuration. Moreover, this version of the processor supports cache memories.

Figure 4.2: Interconnection Between Register Contexts and Processor Pipelanes Controlled by the Reconfiguration Controller



Source: Adapted from (BRANDON et al., 2017)

Figure 4.3: Dynamic ρ-VEX Schematic



Source: (HOOZEMANS; STRATEN; WONG, 2017)

The 8-issue processor uses 8 write and 16 read ports in the register file, the 4-issue 4W and 8R, and the 2-issue 2W and 4R. Therefore, when changing the configuration (between large, medium, and small cores) the number of ports is not modified. On the other hand, the number of registers needs to be quadrupled, because each core needs an exclusive register file of 64 registers (up to four cores are supported) (HOOZEMANS et al., 2015).

The main advantages of adapting the issue-width to the behavior of the application are:

- *Single-thread applications:* different applications present distinct characteristics regarding hardware usage, and even within the same application the hardware usage may vary (e.g., phases with high or low ILP). In addition, by adapting the issue-width to the application's requirements, energy can be saved by turning off idle resources.
- *Multiple threads/applications:* this approach can exploit the idle resources to allocate other threads and applications in the idle resources, efficiently exploiting the available hardware as the new threads may be scheduled to use the idle resources without affecting the execution of the other running threads.

When it comes to generating the binary code, traditional VLIW processors require code recompilation every time that the hardware organization is changed (as the ILP is exploited statically, the compiler must know exactly which are the available resources). Therefore, if the hardware configuration were to be changed during run-time to exploit the different phases characteristics that an application can present, and consequently, use the available hardware more efficiently, several binaries would have to be loaded into memory so they would be available during execution. Moreover, a switching control mechanism would be required to guarantee the correct state of the processor when one binary is changed to another. In order to tackle this issue, modifications to both hardware and compiler are proposed by Anjam, Nadeem and Wong (2011), Brandon and Wong (2013), Brandon et al. (2015).

Splitting bundles cannot be made simply by dividing the instruction word into smaller parts, as hazards can be introduced by this process, consequently leading to wrong computation, as previously discussed. Therefore, the authors use a post-assembler tool to sort the instructions in a way that they can be executed in different cycles, which is able to avoid most of the hazards. However, there will still be cases in which is not possible to split the instructions, as follows.

Figure 4.4: Example of Bundle that Cannot be Split Using the Generic Binary Scheme

```
1    c0    shru $r0.13 = $r0.15, 24
2    c0    shru $r0.9 = $r0.13, 16
3    c0    shru $r0.15 = $r0.9, 8
```

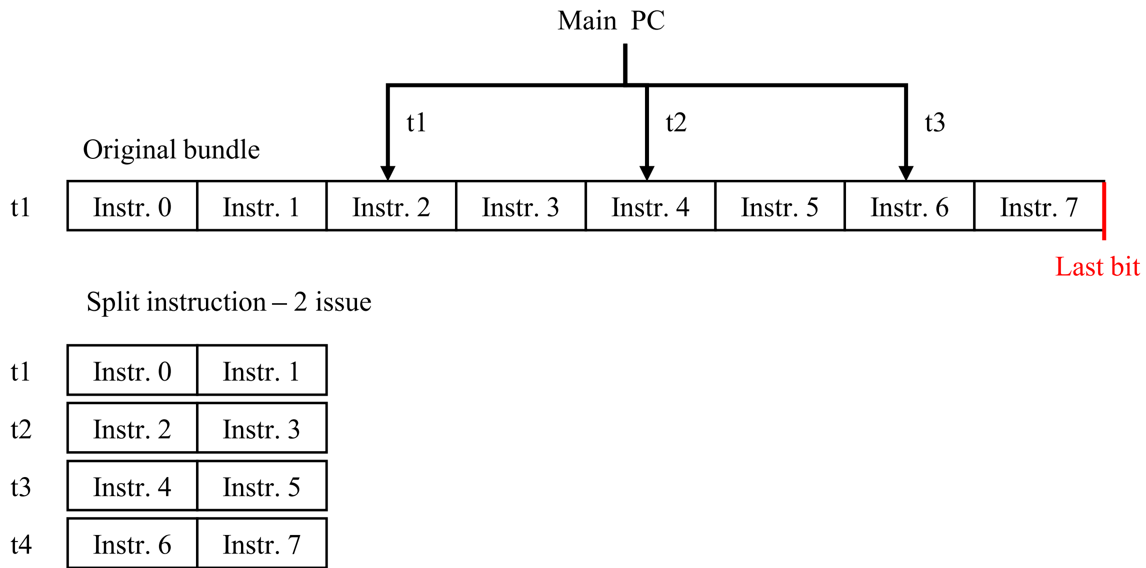
Source: (BRANDON; WONG, 2013)

Figure 4.4 presents an example of a bundle that cannot be split. That is, if it were to be split, a Read After Write (RAW) hazard would be introduced and the computed result would be wrong. Hence, this is detected by the post-assembler tool by analyzing the false dependency graph of the instructions. If a cycle is present in this graph, the bundle cannot be divided. Another requirement is for bundles that contain branch instructions. In such cases, the branch must be executed only after the other instructions of that bundle, otherwise, the branch would be treated before the other instructions have time to finish the computation, which would also lead to wrong computation.

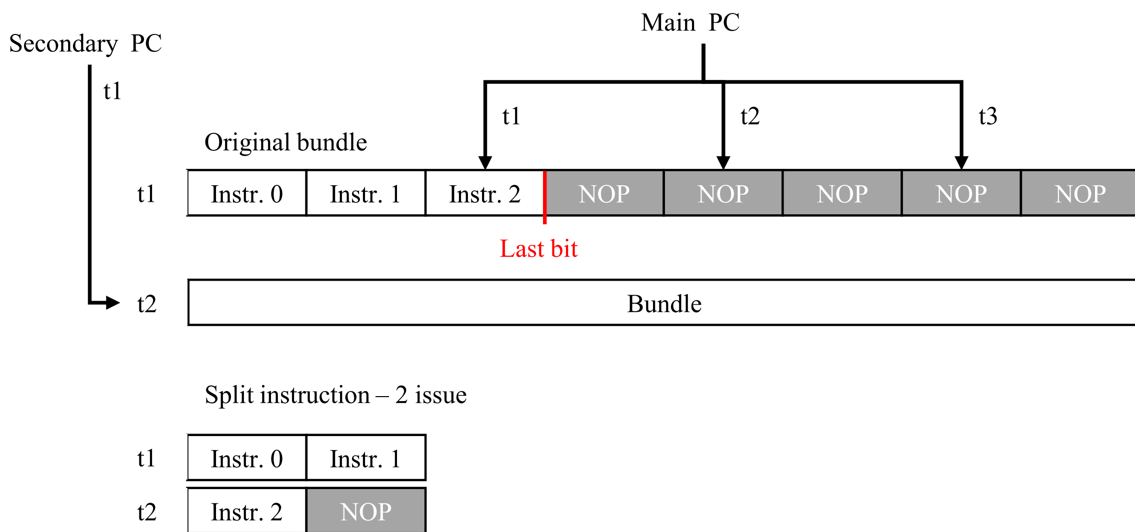
As aforementioned, in several cases, the compiler is not able to fill all bundle with program instructions (filling them with NOPs). However, splitting instructions that only have NOPs would only decrease performance as no useful computation is performed. In order to address this issue, the authors inserted a bit to identify the last instruction of the bundle (called last bit). Hardware support was added so the instructions that come after the one identified by the last bit are skipped. To do so, a second program counter was added. This new PC is equal to the address of the next 8-issue bundle. When the last bit is set, the second PC (next 8-issue PC) is used to fetch the instruction, instead of the main PC (which contains the address to the next part of the current bundle).

Figure 4.5 depicts an example of instruction split for an 8-issue bundle that has no NOPs (Figure 4.5(a)), and a bundle that has an elevated number of NOPs. Figure 4.5(b) illustrates how the last bit and the secondary PC work. Considering the example of a bundle that has three instructions followed by five NOPs, if it were to be split to 2-issue mode, the last bit is placed in *Instr. 2*. Therefore, at t_2 (in the split bundle) the last bit is detected and the remaining two bundle parts (each containing two NOPs are skipped. When the instruction is split, the main PC follows the addresses for the split parts (instead of the 8-issue word). This means that at t_1 it will point to the next part of the bundle (*Instr. 2*), at t_2 to *Instr. 4*, and so on. As aforementioned, the secondary PC is responsible for keeping track of the next 8-issue bundle. Thus, when the last bit is detected, the secondary PC overwrites the main PC, which skips the execution of the last two bundle parts.

Figure 4.5: Instruction Split Example



(a) Instruction Split from 8-issue to 2-issue

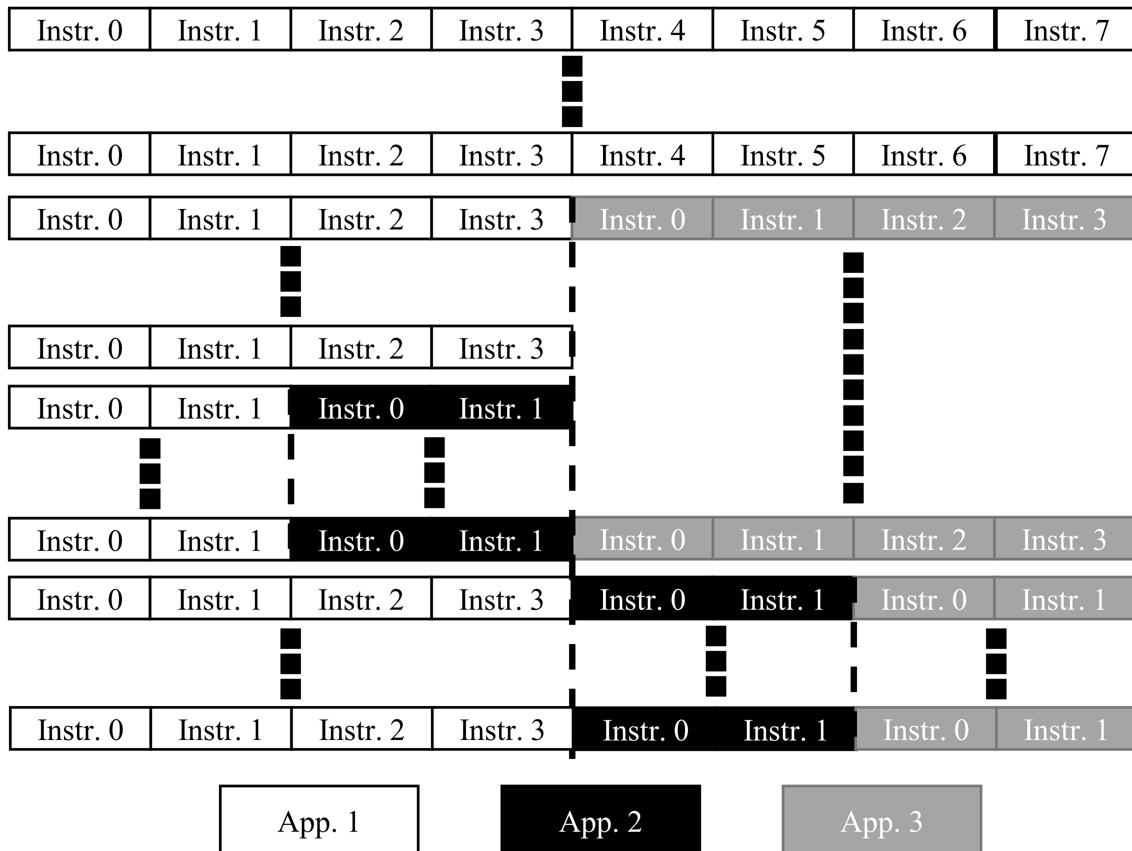


(b) Last Bit and Secondary PC when Splitting Bundles

Source: The Author

Figure 4.6 depicts the execution of three applications that exploit the ability of dynamic issue-width adaptation, in this example the *App. 1* starts executing in 8-issue mode, then switches to 4-issue, so the *App. 2* can execute in parallel in the remaining four issue slots. Later on, the *App. 1* is switched to 2-issue mode, and the *App. 3* starts executing in two issue slots, and finally *App. 1* goes back to 4-issue mode and the other two applications go to 2-issue mode. Therefore, the applications may be executed faster or slower depending on the issue-width configuration and intrinsic ILP from the application. Applications may exploit low ILP phases to reduce the issue-width, consequently freeing this idle hardware to the other applications, or the applications may be forced to run at a

Figure 4.6: Dynamic Issue-width Adaptation



Source: The Author

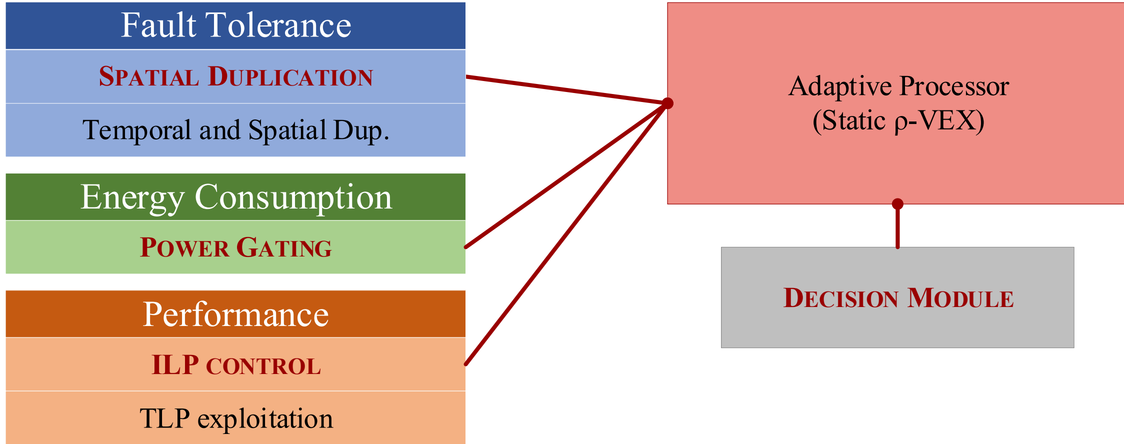
reduced speed (when the issue-width is lower than the ILP) if more applications have to be executed in a given moment.

4.2 Proposed Adaptive Processor

The adaptive processor is based on the *static* version of the ρ-VEX processor. Therefore, adaptive optimization techniques are applied to the processor, but the issue-width is not changed during runtime. As aforementioned, the dynamic issue-width adaptation will be used in the polymorphic processor.

The adaptive design comprises the spatial duplication mechanism, the ILP control module, and it can enable or disable the PG module. These techniques are highlighted in Figure 4.7. Next, two configurations are discussed: reliability-oriented and energy-oriented.

Figure 4.7: Adaptive Processor Overview



Source: The Author

4.2.1 Reliability-oriented Configuration

When the focus is on reliability, the PG module is not used. As aforementioned, the threshold can be either configured at design time or modified during runtime based on pre-defined requirements (which will be explained in detail in Chapter 6). As aforementioned, the utilization value is compared to the threshold. The utilization value is presented in (4.1) and calculated from the ratio between the sum of the number of used issue slots on the high part (pipelanes 4 to 7) of each bundle and the number of executed bundles that use more than half of the issue-width. Hence, this value represents the average utilization of the issue slots considering the bundles on which full duplication without the ILP reduction cannot be applied. Note that only bundles that have some instruction at the high part will change the utilization value; otherwise, the full duplication will be automatically applied, since it incurs no performance penalties. It also guarantees correct program execution by detecting data-dependent operations: instructions that write to a certain register in the first half of the instruction word and read from the same register on the second half would introduce a read after write hazard when split, resulting in wrong computation. These hazards are detected in hardware, and these instructions are not split, preserving program correction.

$$UtilizationValue = \frac{\sum_{i=1}^{n}(\#Instr_{HighPart})}{\#Bundles} \tag{4.1}$$

Figure 4.8: Spatial Duplication with ILP Control

	P0	P1	P2	P3	P4	P5	P6	P7
C_0	I0	I1	I2	I3	I0	I1	I2	I3
C_1	I0	I1	I2	NOP	I0	I1	I2	NOP
C_2	I0	I1	I2	I3	I0	I1	I2	I3
C_3	I4	NOP	NOP	NOP	I4	NOP	NOP	NOP
C_4	I0	I1	I2	I3	I0	I1	I2	I3
C_5	I4	I5	I6	NOP	I4	I5	I6	NOP
C_6	I0	I1	NOP	NOP	I0	I1	NOP	NOP
C_7	I0	I1	I2	I3	I0	I1	I2	I3
C_8	I4	I5	I6	I7	I4	I5	I6	I7

(a) Threshold = 1

	P0	P1	P2	P3	P4	P5	P6	P7
C_0	I0	I1	I2	I3	I0	I1	I2	I3
C_1	I0	I1	I2	NOP	I0	I1	I2	NOP
C_2	I0	I1	I2	I3	I4	I1	I2	I3
C_3	I0	I1	I2	I3	I0	I1	I2	I3
C_4	I4	I5	I6	NOP	I4	I5	I6	NOP
C_5	I0	I1	I2	I3	I0	I1	I2	I3
C_6	I0	I1	NOP	NOP	I0	I1	NOP	NOP
C_7	I4	I5	I6	I7	I4	I5	I6	I7

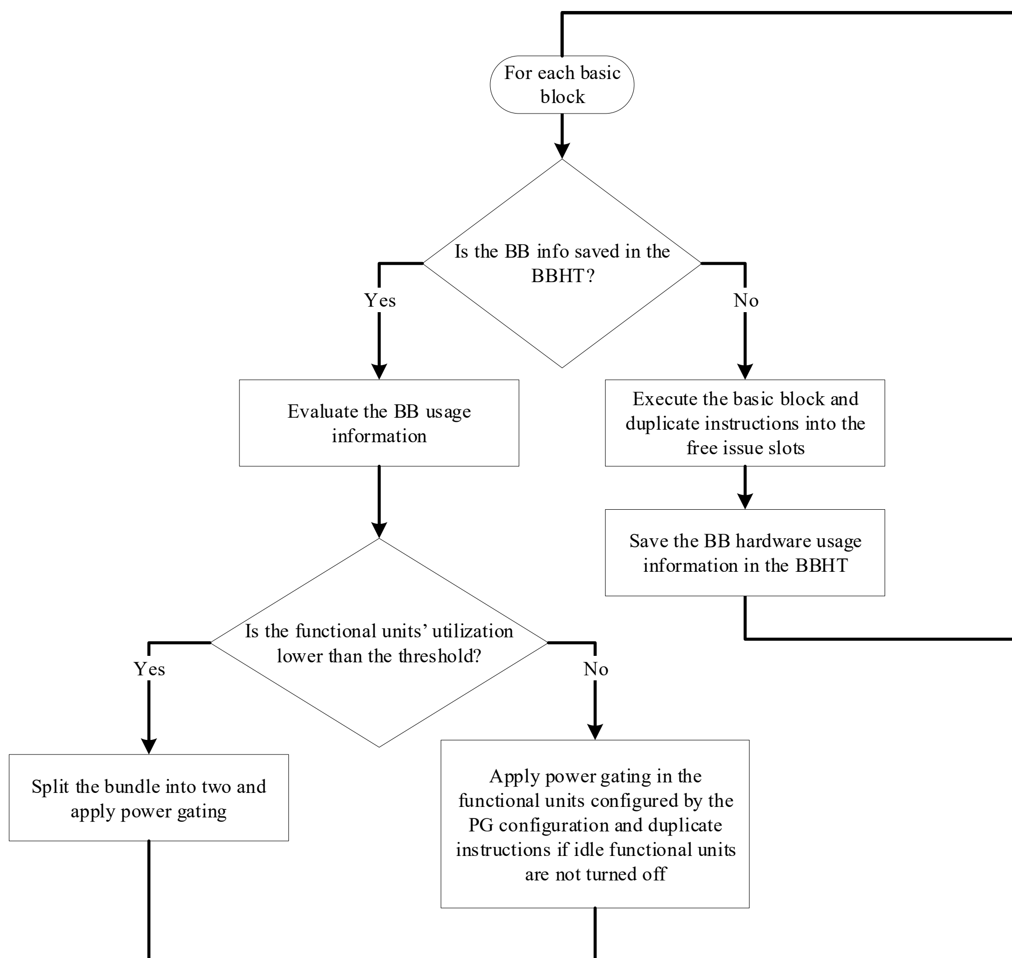
(b) Threshold = 2

Source: The Author

Examples of code execution using different thresholds (1 and 2) are depicted Figure 4.8(a) and Figure 4.8(b), respectively. The instructions that are split into two cycles (allowing full duplication) are highlighted by the arrows on the right side of the instruction word. When the threshold is equal to 1, every bundle that has more instructions than the half of the issue-width is split into two, because the utilization value will always be at least 1 for those bundles (e.g., C_2, C_4, and C_7). When setting the threshold to 2, the bundle at time C_2 will not be divided because the average utilization value will be equal to 1, which is below the threshold. The instruction bundle at C_3 will be divided because the utilization value will be equal to 2 (4 used issue slots/2 bundles). The same reasoning goes to the instruction at time C_6, which has a value above the threshold.

Let us analyze Figure 3.6 again. As it can be observed, there is no performance overhead when the spatial duplication is used (Figure 3.6(b)). However, eight instructions would not be duplicated. By using ILP reduction with the threshold equal to 1 (Figure 4.8(a)), we would have 50% of performance degradation with the ability to duplicate all instructions. If a threshold equal to 2 (Figure 4.8(b)) is chosen, there would be 33% of performance degradation and one instruction would not be duplicated. Hence, either fault tolerance or performance can be prioritized for a given application by changing the technique and/or the threshold value.

Figure 4.9: Adaptive Processor's Execution Flow



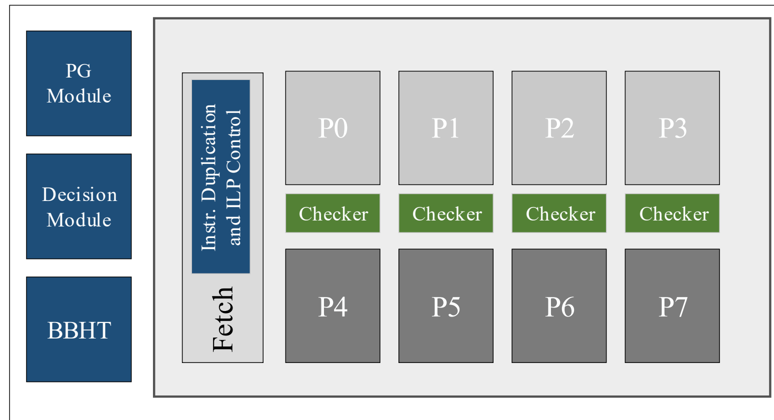
Source: The Author

4.2.2 Energy-oriented Configuration

When the focus is energy consumption, the operation flow depicted in Figure 4.9 is applied and the overview of the processor with the additional modules is presented in Figure 4.10, both are explained in detail next:

1. Whenever there is an opportunity (i.e., a NOP is being executed on any pipelane of the upper half of the VLIW word), the correspondent instruction will be duplicated.
2. Whenever a program phase that respects the break-even constraints is detected in a given pipelane or functional unit, power gating will be applied. This will influence the previous item (duplication), since fewer pipelanes will be available for duplication.

Figure 4.10: Overview of the Processor with Fault Tolerance, Energy Optimization, and Performance Management Mechanisms



Source: The Author

3. The ILP control can split instruction words, as already explained. This open more opportunities for power gating, which will positively impact (2) and, by consequence, negatively influence (1).

The fault-tolerant mode (1) is automatic and always on. For (2), the decision module detects, at runtime, which parts of the hardware can be shut down; and, for (3), when and how the ILP reduction shall be applied.

For (2), the decision module detects program phases as the application is executed. This is done by using a similar approach as Sherwood, Sair and Calder (2003), in which the information regarding the hardware utilization of each basic block is saved for future reuse; in our case, in a direct-mapped memory. This memory is called Basic Block History Table (BBHT), which is indexed by the PC of the first instruction of the BB. Each entry of this memory contains one bit per functional unit that indicates if it will be turned on or off (i.e., power gated) next time the same BB is executed. If the functional unit was used at least one time during the execution of the BB, it means it must be turned on next time the same BB is found, so the correspondent bit is set to *0*; otherwise, it is set to *1*. As there are 16 functional units (two per pipelane), 16 bits are needed for this purpose. Adding to them, there are 32 bits of the PC; and two control bits, in a total of 50 bits per entry (called PG configuration).

Every time a BB ends its execution (reaches a branch instruction), the PC of the next BB to be executed is searched in the BBHT. In case the configuration for the next BB is not found (because that BB was never executed before or because it was replaced), the decision module will build its PG configuration. If the entry is found, the decision

module will turn off the functional units according to the correspondent bits in the PG configuration.

The ILP control for the energy-oriented mode (3) represents the utilization ratio of a given functional unit during the execution of the BB, defining whether instructions will be split into two or not. For instance, if the threshold is configured to 50%, a given functional unit on the high part of the issue-width must be used more than 50% so it will remain active during that BB, otherwise (if it is used less than 50%), the instructions will be split into two cycles and the correspondent functional units will remain turned off to maximize the power gating phase.

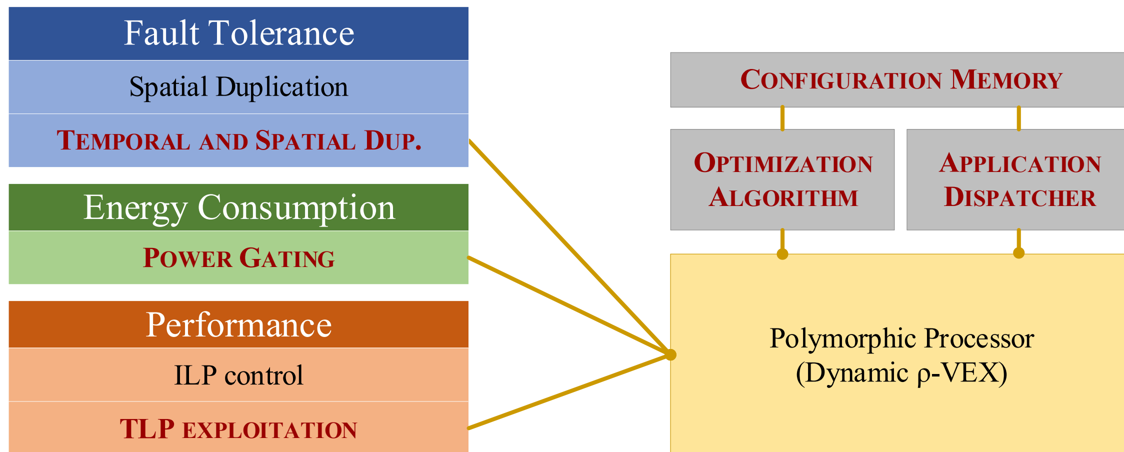
Therefore, the higher the threshold value is, the more VLIW instructions will be split. By consequence, there will be more program phases so power gating can be applied, and fewer instructions will be replicated (energy optimization will be given more weight). By adjusting the threshold to its minimum value, fault tolerance and performance will be prioritized over energy consumption.

4.2.3 Final Remarks Regarding the Threshold Configuration

Note that the ILP threshold for the PG mechanism has different behavior from the one for the instruction duplication. While the ILP threshold for the PG splits bundles that do not have sufficient ILP to reach the threshold (which is calculated for each BB), the threshold for the instruction duplication splits bundles that exceed the defined threshold (which is calculated considering the average ILP). In addition, only one of these two thresholds is active in a given configuration (which is defined at design time).

By configuring the threshold, the designer can tune the processor to be more performance, energy optimization, or reliability-oriented. This tuning highly depends on the target application's and system's requirements. For instance, one may require having an application that does not spend more than X Joules, executes in less than Y cycles, or has a certain degree of reliability. These constraints can be met by adjusting the threshold value and testing the application, or enabling/disabling a certain mechanism. In this work, the threshold for the ILP control mechanism for reliability was evaluated with several values (from minimum to maximum) and the energy-oriented ILP mechanism was evaluated with its minimum and maximum values, so the trade-off between all axes can be evaluated.

Figure 4.11: Polymorphic Processor Overview



Source: The Author

4.3 Proposed Polymorphic Processor

The proposed polymorphic processor is based on the *dynamic* version of the ρ-VEX processor. The available dynamic issue-width adaptation was used and extended so the implemented techniques could also dynamically change during runtime. Figure 4.11 presents the chosen techniques (introduced in Chapter 3) to be part of this processor, which are also detailed next:

- *Fault Tolerance:* Temporal and spatial duplication (Section 3.1.1.2), as it was the technique with more flexibility and capable of duplicating more instructions than the previous ones.
- *Energy Consumption:* Power gating (Section 3.2) with dynamic application profiling to detect phases in which certain FUs could be shut down to save energy.
- *Performance:* Application dispatcher to exploit TLP and execute several benchmarks in parallel. The ILP control mechanism was not used because the temporal duplication already exploits the execution of duplicated instructions in different time slots, so a more complex mechanism is required to exploit the benefits of controlling the ILP together with the temporal duplication mechanism, which will be implemented as future work.

The runtime adaptation process for the aforementioned techniques comprises: modify the buffer size, enable or disable the power gating mechanism, and change the issue-width. Figure 4.11 also depicts the additional modules for the polymorphic processor. The optimization algorithm comprises two main phases: learning and runtime (detailed in the

next subsections). The buffers contain several banks, which can be shutdown to dynamically re-size the available buffer size. The bank size can be configured during design time, and each bank can be powered on/off independently by a bypass switch that enables or disables the power supply to a given bank (WANG; KOREN; KRISHNA, 2011; PONOMAREV; KUCUK; GHOSE, 2001). Note that for a bank to be turned off, no instructions can be in the buffer. However, for the proposed processor, the buffer will always be empty when the buffer size is changed, because the buffer is only used inside the kernel and the configuration switch occurs after the kernel was completely executed. The PG mechanism is enabled and disabled through a control flag, and the hardware modules responsible for these functions are shut down when these mechanisms are disabled. Finally, the dynamic ρ-VEX provides the issue-width adaptation through a switch network.

The *Configuration Memory* contains two small memories that are used to (1) store the list of configurations that are going to be evaluated by the optimization algorithm and (2) store the best configuration (for each benchmark, identified by its Process ID (PID)) after the learning phase. The *Application Dispatcher* is responsible for scheduling the applications considering the available issue slots and the configurations that were dynamically chosen by the optimization algorithm, therefore, minimizing the idle hardware and executing multiple applications concurrently.

4.3.1 Learning Phase

In this phase, a learning algorithm was implemented to evaluate different configurations and find which one delivers the best trade-off considering fault tolerance, energy consumption, and performance. This learning is done by changing the hardware configuration (polymorphic behavior) during runtime so each execution the application's kernel can be evaluated with a different configuration. For this, the kernel of the application must be identified using *pragmas*, so the compiler is able to generate a code that will inform the hardware which are the regions that need to be tested and optimized. Even though the insertion of a *pragma* makes the proposed processor lose the complete transparency to the programmer that it had so far, its use is very simple and straightforward, the programmer only needs to mark which region corresponds to the kernel of the application.

In order to evaluate the trade-offs among the aforementioned axes and choose the best configuration, the Mean Work Per Unit of Energy to Failure (MWPUETF) metric is proposed. This metric is composed of the Mean Work to Failure (MWTF), which was

adapted from (REIS et al., 2005a), and the energy consumption of the application. The MWTF equation is presented in (4.2), where the *core utilization* is the ratio between the number of program instructions and the total number of instructions (program instructions plus NOPs). With that, it is possible to capture the trade-off between performance and fault tolerance. This allows evaluating the reliability of different issue-widths after removing the influence of the NOPs and the difference in execution time when the issue-width is changed. To obtain the failure rate, a fault injection campaign was conducted and faults were injected at the design's gate-level, using the Simbah-FI framework, which will be detailed in Chapter 5. The metric MWPUETF is depicted in (4.3), which is the ratio between the MWTF and the energy consumption. Each of these metrics is obtained during runtime by using hardware performance counters, with the exception of the failure rate, which is obtained from a previous fault injection campaign. Moreover, the MWPUETF can be used to prioritize specific axes, for instance, prioritize energy consumption, or performance, or fault tolerance. To allow the modification of the weight of each axis, the MWPUETF formula is extended to (4.4), where $a + b + c = 1$. Thus, a, b, and c can be tuned at design time so the adaptive processor considers different weights when evaluating the trade-off among the axes.

$$\begin{aligned} MWTF &= \frac{amount\ of\ work\ completed}{number\ of\ errors\ encountered} \\ &= \frac{core\ utilization}{(failure\ rate) \times (execution\ time)} \end{aligned} \tag{4.2}$$

$$MWPUETF = \frac{MWTF}{energy\ consumption} \tag{4.3}$$

$$MWPUETF = (\frac{core\ utilization}{failure\ rate})^a \times \frac{1}{(exec.\ time)^b \times (energy\ cons.)^c} \tag{4.4}$$

The basic idea of the learning algorithm is to evaluate a different hardware configuration each time a given kernel is executed until the best configuration is found. This means that the application will run with a sub-optimal configuration until the final configuration is found, but as we will demonstrate in the results chapter, this learning phase can be performed with minimal overhead and the processor is able to adapt itself to any application.

Algorithm 1 presents the learning phase. It has a list of configurations to be evaluated that can be selected during design time and it receives the $kernelList$ as a parameter, which is the identification of the target applications' kernels. The learning algorithm will run until it finds the best configuration for each kernel. In addition, the number of steps of the learning algorithm is reduced by aborting the evaluation of configurations that will not result in an improvement in the result (this will be explained in detail next).

The following optimization decisions are made regarding the learning flow:

- The issue-width is tested in the following order: 2, 4, then 8-issue. In addition, the number of tests is reduced if the MWPUETF decreases when increasing the issue-width. For instance, if the 4-issue core results in a worse MWPUETF when compared to the 2-issue, the 8-issue core will not be tested, because it will result in a MWPUETF that is lower than the 2-issue. This is explained by the following reasoning, the 8-issue consumes more energy (which negatively affects MWPUETF), but the improvement in performance in most cases is much lower than the theoretical 2x (when going from 4-issue to 8-issue) because the compiler is not able to fill all bundles with independent instructions, as already discussed in Chapter 2. Therefore, the energy consumption weighs more than the performance in this scenario.
- For the temporal duplication, there are a few parameters that can be modified to trade-off performance, reliability, energy consumption, such as the buffer size, and use of PG. During the learning, the buffer size stops being tested when the MWPUETF gets worse than for the previous buffer size. When this happens, it means that increasing the energy consumption (larger buffer) does not outweigh the increase in the number of duplicated instructions (if any, because the previous buffer size could be enough to duplicate all instructions that were stored in the buffer).

Therefore, by applying these two optimization strategies, it is possible to reduce the number of steps until the best configuration is found, since those additional steps would not deliver the best configuration and would only create more overhead.

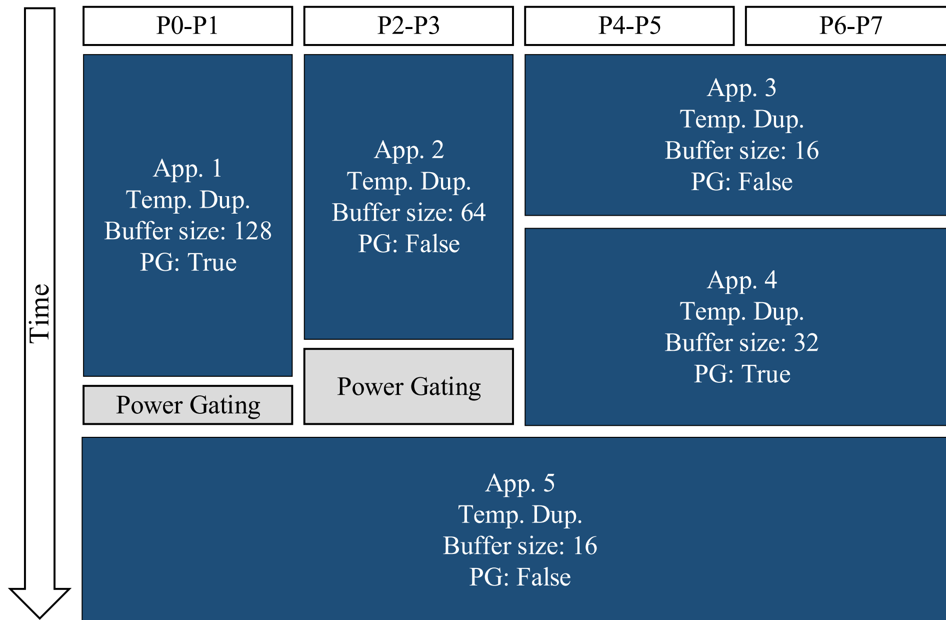
After the first kernel of the queue is obtained in Algorithm 1, the optimization module verifies if there are enough idle slots to schedule this kernel with a given configuration (*l. 4-5*), if there are, the defined configurations are evaluated, one for each execution of the given kernel. While testing the temporal duplication, if the result for the current buffer ($resultCurrentBuffer$) is better than the result for the previous buffer, it is saved in the *configuration memory* (*l. 12*), otherwise, it stops testing further buffer

Algorithm 1: OptimizationAlgorithm-LearningPhase($kernelList$)

```
1  optQueue ← kernelList
2  while kernel in optQueue do
3      currentKernel ← get first application of optQueue
4      if currentKernel was not tested in this time-slot and there are idle slots then
5          if currentKernel fits in the current unused issue-slots then
6              for testConfig in configList do
7                  if testConfig = "TemporalDuplication" then
8                      for bufferSize in bufferSizeList do
9                          for TDconfig in TDconfigList do
10                             Execute currentKernel with
                                 testConfig, TDconfig, bufferSize
11                             if currentResult > previousResults then
12                                 Save currentResult
13                         if resultCurBuffer < resultPrevBuffer then
14                             Break to the next testConfig
15                 else
16                     Execute currentKernel with testConfig
17                     if currentResult > previousResults then
18                         Save currentResult
19             if currentResult is better than the previous issue-width then
20                 if current issue-width < max(issue-width) then
21                     currentKernel.nextIssue ← next issue-width
22                     Put currentKernel back in the optQueue
23                 else
24                     Save currentResult as the best one
25             else
26                 Mark the previous result as the best one
27         else
28             Put currentKernel back in the queue
29     else
30         Put currentKernel back in the queue
31         if there are idle slots then
32             Apply PG on idle slots
33         Wait for an application to end
34         Update time-slot
35 return the best configuration for each application
```

sizes (*l. 14*). Then, the result for the current issue-width of this kernel is compared to the one from the previous issue-width (*l. 19*), if it is better, the next issue-width is evaluated (until it tests the maximum issue-width), otherwise, the best result is the $currentResult$. This is repeated for all kernels that fit in the available issue-slots (*l. 4*). For example, in a

Figure 4.12: Application Scheduling Example



Source: The Author

given moment, four kernels running on 2-issue mode can be executed in parallel. When all issue-slots are occupied, the optimization module waits for a kernel to end, so it can evaluate the next kernel to be scheduled (*l. 30-34*). In case there are still idle slots after all kernels tried to be scheduled (because they did not fit in the available slots), PG is applied to those slots, to reduce the energy consumption (*l. 31-32*).

4.3.2 Runtime Phase

In the runtime phase, the best configuration was already determined by the learning mechanism, and it was stored in the *configuration memory*. Therefore, the *application dispatcher* is responsible for continuing the execution of the applications while trying to schedule as many applications as possible in parallel. The execution time of each kernel was already saved by the learning phase, so in the runtime phase the applications are sorted by the descending order of the execution time (in order to allow more flexibility in the scheduling for the other benchmarks) and the applications are scheduled to fill the available slots.

An execution example is depicted in Figure 4.12, in which the *App. 1* and *App. 2* continue the execution after the learning phase in the best configuration for each of these applications, which is the 2-issue processor with temporal duplication and buffer size of 128 with PG for *App. 1* and a buffer of 64 without PG for *App. 2*. At the same time, the

other four pipelanes are occupied with *App. 3* having a buffer size of 16 without PG. After *App. 3* finishes the execution, *App. 4* is scheduled, maintaining the 4-issue configuration with another buffer size and PG configuration. Finally, *App. 5* is scheduled in the 8-issue mode, occupying all pipelanes. Note that power gating is applied to the empty slots that cannot be used to fit another application (between *Apps. 1 and 5*, and *Apps. 2 and 5*).

5 PROPOSED HYBRID FAULT INJECTOR

5.1 Motivation

Since the terrestrial neutron flux is extremely low, field-testing at sea level takes long periods (from months to more than one year) and requires thousands of devices to assess the reliability of a single hardware component. Therefore, alternative approaches to evaluate the reliability of a system are necessary. High-energy particle accelerators (KOBAYASHI et al., 2004; LESEA et al., 2005; YAHAGI et al., 2002) are used to evaluate the devices at approximately the same terrestrial neutron flux but at an accelerated rate (10^7 to 10^8 times) (VIOLANTE et al., 2007). However, this approach has many drawbacks: elevated financial cost, as there are only a few laboratories in the world that can provide such radiation sources; it is not possible to control with precision which parts of the circuit that will be exposed to the radiation beam; and it is only possible to test the target hardware under radiation after the chip is produced and deployed. Examples of laboratories that provide such equipment for testing are the Los Alamos Neutron Science Center (LANSCE), with a source of neutrons, and the Rutherford Appleton Laboratory (ISIS), with a source of neutrons and muons.

Another approach is to statistically determine the reliability of a structure by computing the Architectural Vulnerability Factor (AVF) (MUKHERJEE et al., 2003), which is the probability that a fault in a particular structure will result in an error. In order to compute the AVF, the processor state bits required for Architecturally Correct Execution (ACE) are tracked. If a fault affects a storage cell that contains one of these bits, the program's output will present an error, when not using any fault tolerance technique. Therefore, the AVF for a single-bit storage cell is the fraction of time that it holds ACE bits. Un-ACE bits (bits that are not mandatory for correct execution) come from NOP instructions, performance-enhancing instructions such as pre-fetching, dynamically dead code, and logical masking. However, computing AVF for complex structures and processors is a difficult process, because calculating and weighing the residency and bandwidth of every bit is extremely difficult, which very likely leads to significant error margins.

Fault injection comes as a flexible, cheap, and controllable alternative, which can be implemented in hardware, software, or using simulated environments; and they may be applied to several project stages, such as conception, design, prototype, and operational phases (HSUEH; TSAI; IYER, 1997).

Hardware fault injection uses additional hardware to introduce faults into the system. It can be categorized into injection with and without contact. The former has direct contact with the target system and produces voltage and current changes in order to inject the fault. Examples of this method are pin-level probes and sockets. The latter has no physical contact with the system and an external source produces a physical phenomenon in the system, for instance, heavy-ion radiation or electromagnetic interference.

In order to inject SEUs in the configuration bits of FPGAs, the bitstream is modified. Nazar and Carro (2012) propose a platform to perform the fault injection and evaluate the correctness of the outputs that requires only one FPGA board.

Software fault injection has the main advantage that it does not need special hardware (fault injector or FPGA board) to perform the fault injection and it may be inserted into the application itself or between the application and the operating system. On the other hand, it is restricted to inject faults only into locations that the software has access and it may disturb the workload on the target system. This approach may be applied during compile-time or run-time. In the former, the program instructions are modified; the latter uses triggers to activate the fault injection during runtime, such as time-outs, exceptions and traps, and code insertion.

When applying fault injection in simulated environments (in which the target system is simulated), both high-level and low-level simulations are possible, which represent the target's system level of detail and result in different levels of accuracy (the lower the level, the more accurate). The main advantages of this approach are: cost - can be applied at design time, before deployment (so the designer can change the circuit in case its reliability does not meet a certain criteria); high controllability of the fault injection - no need for special equipment or hardware components; and flexibility to choose between different fault models. Even though both high- and low-level simulations are slow when compared to the real hardware implementation, they allow the evaluation of the circuit in early design stages.

Fault injectors for high-level simulators usually only change the value of the registers during the execution of an application (e.g., the architectural registers of a processor), which leads to highly inaccurate results: for example, given a complex superscalar processor, the architectural registers (those visible to the programmer) represent a small fraction of all registers present in the system (i.e., the pipeline registers, instruction queue, branch history buffer and so on will not be affected). On the other hand, when one considers low-level simulators (e.g., gate-level simulation), faults can be injected in the hardware module's low-level signals, which has high controllability and accuracy.

Even though behavioral RTL simulation provides more details than high-level simulators (i.e., access to the pipeline registers, instruction queue, etc.), the visibility of such signals is not enough to guarantee an accurate fault injection process. Therefore, gate-level simulations are required to provide detailed information about the target modules as they are based on the post-synthesis netlist of such modules. However, simulating complex hardware in such level of detail is very time-consuming. In the proposed fault injector, we combine the accuracy of gate-level fault injection with the simulation speed of RTL simulation, by executing the circuit in two levels of detail. Next, several fault injectors will be compared, followed by the implementation details of the proposed fault injector.

5.2 Fault Injectors - Related Work

Some of the high-level simulators that were modified to inject faults during the application's execution are discussed next: GemFI (PARASYRIS et al., 2014) is an example of such technique, as it provides a framework for injecting faults in the Gem5 simulator (BINKERT et al., 2011). A number of parameters are passed to the simulation in order to characterize the fault, including location, thread, time, and behavior. FIMSIM (YALCIN et al., 2011) presents a similar approach to Parasyris et al. (2014); however, it was built on top of the M5 (BINKERT et al., 2006) simulator. Relyzer (HARI et al., 2012) uses two high-level simulators (Simics (MAGNUSSON et al., 2002) and GEMS (MARTIN et al., 2005)) to speed up the fault injection process.

Kaliorakis et al. (2015) use a fault injector for the MARSSx86 architectural simulator (PATEL et al., 2011) and their approach is to terminate the simulation when a fault is masked, or when the fault is expected to result in a failure. This reduces the simulation time, but it also compromises the accuracy of the method, as the estimation on whether the fault results in a failure may be incorrect. These previous approaches are not able to accurately represent the behavior of the system under test as the faults are injected only at high-level signals and registers, and they are restricted to a few ISAs.

Goswami (1997) and Kalbarczyk et al. (1999) propose to associate high-level effects with low-level errors, in order to reduce the simulation time. However, such statistical models may not reflect the actual error propagation when the interactions with the rest of the system are considered (CHO et al., 2013).

Cho et al. (2015) propose a hybrid fault injector that comprises a high-level simulator (Simics) and gate-level simulation for accurate fault injection. Even though the authors are able to achieve high speedups by using a high-level simulator to execute the application before and after the fault is injected, the use of high-level simulators greatly restricts the possibility of injecting faults in designs that are not supported by the Simics simulator (and other high-level simulators), which supports few processor architectures (Wind River, 2017). Kooli, Natale and Bosio (2016) present an evaluation of faults that affect data and instruction caches by combining fault injection with analytical techniques. They use a cache emulator to trace the cache contents for a given program execution. Then, they randomize the time and bit position to inject the faults. When an unused position of the memory is the target for the fault injection, they consider that the fault was masked and do not inject the fault in the application. Otherwise, a fault is injected in the software layer. That is, the assembly of the program is modified to corrupt a certain data or to replace one instruction for another one (e.g., replace an *ADD* instruction for a *SUB*). Therefore, their approach is restricted to cache memories only, requires the assembly code to be modified and does not have access to the low-level signals of the processor (which are necessary so the faults can be distributed evenly across the area of the chip).

FITSEC (EJLALI et al., 2003) proposes a hybrid fault injector that uses both behavioral RTL simulation and FPGA-based emulation to run the application and inject the faults. However, the hardware description of the target module must be modified so the faults can be modeled, which implies that the user must manually choose which gates/signals are subject to faults, and determine the time in which the fault will be activated. As FPGA boards are required to inject the faults, this approach is not scalable to several simulations in parallel because it involves financial cost and is technology dependent.

SWAT-Sim (LI et al., 2009) analyzes the behavior of gate-level faults and proposes probabilistic microarchitecture-level fault models to mimic gate-level faults, which results in an overhead of three times when compared to RTL simulation. In addition, their approach is only applied to small combinational logic blocks (e.g., ALU or decoder).

The proposed fault injection framework is able to maintain the fault injection accuracy of gate-level simulation and speed up the simulation process by executing the remaining of the application in behavioral mode. The framework automatically and transparently switches the current context between these two simulation modes when the fault is going to be injected. Moreover, the proposed approach does not require any modification in the application to be executed, and it can be applied to complex circuits.

5.3 Fault Injection Framework

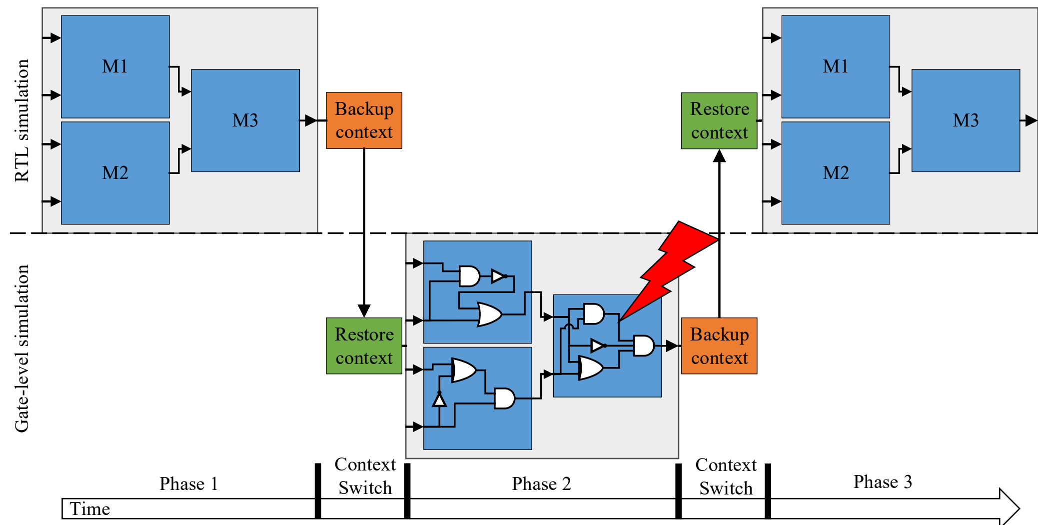
In order to seek the best solution considering accuracy and simulation speed, we also propose the Simulation-based Hybrid Fault Injector, Simbah-FI, which combines the simulation speed of behavioral RTL simulation with the accuracy of gate-level simulation, therefore allowing faults to be accurately injected at an accelerated rate. This is done by using two projects of the target hardware. That is, the same hardware description is used to generate two different projects, one with high and the other with low level of detail (which will be discussed in detail later). Thus, the gate-level simulation is used for accurate fault injection, and the remaining of the application is executed in behavioral mode, so the simulation can be accelerated and still maintain cycle-accuracy. In addition, the proposed fault injector allows faults to be injected in several simulations in parallel, which can be exploited to improve even more the fault injection speed. Note that the number of parallel simulations depends on the number of cores and memory available on the host computer.

5.3.1 Implementation

The framework was developed in Tool Command Language (TCL), and the Modelsim simulator was used to simulate the target design. In order to run the fault injector, the following information is required:

- Hardware description of the module to be simulated (VHDL or Verilog). The same hardware description is used for both projects: one that will be used for behavioral mode simulation; and other, which is comprised of post-synthesized modules that contain internal signals at gate-level, that will be used for low-level fault injection. As the behavioral simulation contains less information about the target circuit, it is used to speed up the simulation before and after the fault injection, while maintaining the simulation's cycle accuracy.
- Post-synthesis netlist generated by the synthesis tool. The user chooses the target modules of the design under test and saves the generated list of signals to a file so the framework can choose which will be the affected signals.
- Execution time of the application. This is required so the execution can be automatically aborted if the application gets stuck in a loop due to a control flow failure. The user can insert this value in the configuration of the fault injector, or the framework can obtain it automatically by running a fault-free circuit in the first execution.

Figure 5.1: Fault Injection Flow



Source: The Author

5.3.2 Fault Model

The proposed framework injects SETs in the circuit, and the following fault model was used:

- *Fault type*: The injected faults are transient and comprise a SET that will affect a signal from the design.
- *Injection place*: The faults are injected into any atomic signal of the target module(s). All internal and low-level signals are considered (gate-level).
- *Injection instant*: Follows a uniform probability function in the range between zero and *t* equal to the expected execution time of the application without faults.
- *Fault duration*: To increase the likelihood of the SET to be captured by a flip-flop, the signal is forced for the duration of one clock cycle.

5.3.3 Hybrid Mechanism

Fig. 5.1 depicts the flow for the hybrid fault injection mechanism, which is completely automatic and transparent to the user. The M_{1-3} represent hardware modules from an arbitrary target circuit, and it is explained in detail next.

The context switch represents the required information to restore the simulation from the exact same point, which depends on the target circuit that is being evaluated.

The configuration of the context signals can be easily done in the framework by specifying which are the required signals for the context switch (target circuit dependent), and for our case study (a complex VLIW processor) the following information is saved: current program counter of the application; register file, branch registers, and link registers; and data memory.

The fault injection flow is divided into three main phases as follows.

- ***Phase 1*** : The application is started in the RTL simulation, and it is executed until a random time. Then, the context is saved.
- ***Phase 2*** : After restoring the context, the gate-level simulation (which considers all the internal signals) is started from the point in which ***Phase 1*** ended. The framework chooses a random bit among all signals and injects a fault on it. For this, the fault injector checks the current value of the chosen bit and inverts it during a clock cycle. The gate-level simulation runs for a few more cycles – this parameter is configurable, and it was chosen as 15 cycles for the experiments - so we can guarantee that the fault had enough time to propagate in the circuit before switching back to the RTL mode.
- ***Phase 3*** : After ***Phase 2***, the context is restored once more so we can speed up the remaining of the application as well, and the application is executed until completion.

In order to coordinate these context switches, additional flags are saved to files so the simulations can be managed in an automatic and transparent fashion. After the execution, the memory is compared to the golden copy in order to detect if there are any data failures, and the number of executed cycles is compared to the expected value to detect control flow failures. This approach is technology independent, as the functionality of the circuit is tested without depending on any technology-specific parameters, such as feature size.

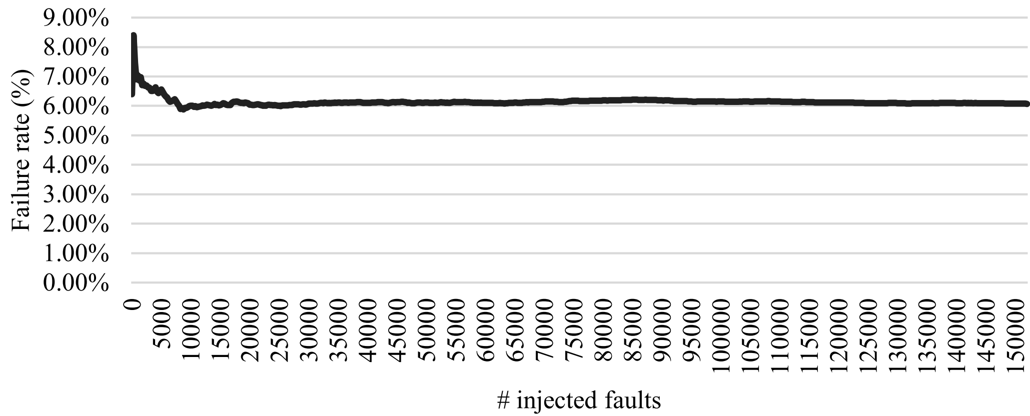
5.4 Fault Injection Accuracy

The proposed fault injector is able to maintain the accuracy of gate-level simulation and speed up the simulation by executing the remaining of the application in behavioral mode. Running a part of the application in behavioral mode does not change the accuracy of the proposed method because the actual fault injection is performed in

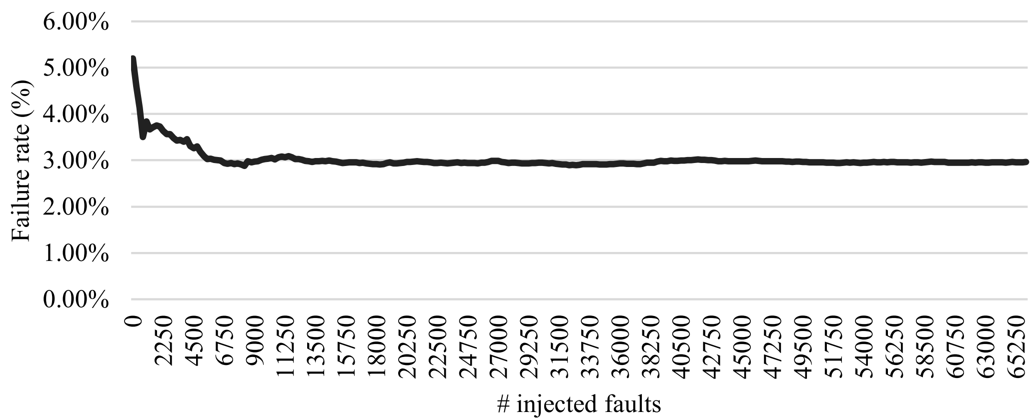
the gate-level simulation. The fault injection accuracy comparison between RTL and gate-level will be presented and discussed in Section 6. After the fault is injected and propagated through the circuit, behavioral simulation can be used to execute the remaining of the application cycle-accurately and faster than gate-level simulation. Examples of the stabilization of the failure rate as more faults are injected for three benchmarks are depicted in Figure 5.2. Therefore, for each benchmark a minimum of 10K faults are injected in order to get the stable behavior.

In addition, it is possible to inject faults in several simulations in parallel, which speeds up, even more, the fault injection process. As aforementioned, the number of parallel fault injections is dependent on the available resources (i.e., number of cores and memory). These parallel simulations are entirely independent of each other. Therefore, there is no interference in the results. In our setup, we executed more than 50 simulations in parallel, which can be extended as more computing resources are available.

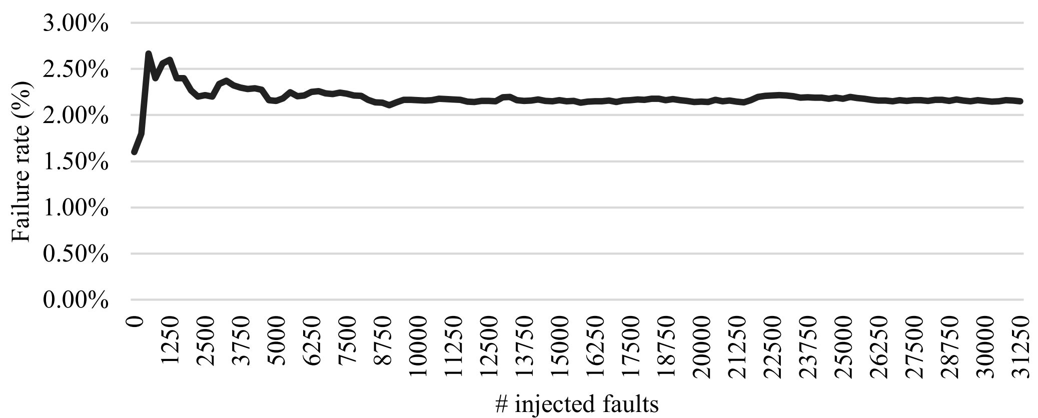
Figure 5.2: Failure Rate Behavior as more Faults are Injected (Unprotected Processor)



(a) CJPEG



(b) CRC



(c) NDES

Source: The Author

6 RESULTS

6.1 Methodology

By having access to the hardware description of the ρ-VEX processor, we are able to measure performance at cycle-level; measure energy consumption and area after synthesis; and inject faults at the gate-level. The synthesis tools used were: Cadence Encounter RTL compiler to obtain power dissipation and ASIC area, using a 65nm Complementary Metal–Oxide–Semiconductor (CMOS) cell library from STMicroelectronics (the operating frequency was set to 200MHz).

CACTI-P (LI et al., 2011) was used to estimate the area and energy consumption of the following modules:

- BBHT: 256 lines of 64 bits each (the closest power of two of 50 bits), one write and one read port. This size was chosen after evaluating different configurations for the BBHT.
- Buffer sizes for the temporal duplication: buffers of 16, 32, 64, and 128 entries were evaluated. Even though both buffers contain the same amount of entries, the commit buffer occupies more area and dissipates more power as it stores the whole bundle in each entry. On the other hand, the issue buffer stores a single instruction. In order to allow the dynamic resizing, each buffer is divided into banks of 16 entries each.
- *Configuration memory* module for the optimization algorithm: This module comprises two small memories, the first has 32 read-only entries for storing the list of configurations that are going to be evaluated by the optimization algorithm (27 entries are used in the current evaluation); the second has 16 entries to store the best result and the corresponding configuration, for each benchmark during the learning phase (11 entries are used as we are evaluating 11 benchmarks).
- Cache: the instruction and data caches have 16KB for the 8-issue configuration (the 4-issue and 2-issue will access 8KB and 4KB, respectively, therefore, maintaining the 16KB for the whole dynamic processor). The caches are 4-way associative with a block size of 4 bundles (128B for the 8-issue) and a *next-line* prefetch mechanism, which will also fetch the following line whenever a cache miss occurs.
- Main memory: a 512MB memory is used, having a miss latency of 30 cycles.

All the previous memories and buffers have ECC enabled and their size can be modified during design time (note that the buffers and caches can still reduce their sizes during runtime by applying power gating as discussed in Chapter 4. For the area and energy consumption evaluation, all additional modules that were implemented are also taken into account, which include those that were implemented in VHDL and the memories that were simulated with CACTI.

For the dynamic version of the ρ-VEX the whole process of reconfiguring processor takes 8 cycles, which is the time required to flush the pipeline, decode the new configuration and start the execution (HOOZEMANS et al., 2015). In addition, the dynamic core requires that the number of registers is quadrupled in order to have four contexts (4x2-issue) of 64 registers each, resulting in a total of 256 registers. The same reasoning applies to the buffer sizes: each configuration supports up to 128 buffers, so a total of 512 entries is required.

For the energy consumption estimation, when there is no switching activity, only static power is considered. Otherwise, the average switching activity of the circuit is considered to be 30% (GEUSKENS; ROSE, 2012) for the dynamic power dissipation. The switching activity of 30% was chosen because it is the traditionally assumed value for system level analysis of microprocessors (GEUSKENS; ROSE, 2012). During the VHDL simulations, it is verified whether the functional unit is used or not in each cycle. This simplified model was used due to the complexity of measuring and synthesizing the real switching activity of each part of the circuit, given the very significant simulation times.

6.1.1 Benchmarks

The benchmark set is composed of 16 applications from the WCET (GUSTAFSSON et al., 2010) and Powerstone (SCOTT et al., 1998) benchmark suites. Several compilers may be used to compile VLIW code for the ρ-VEX processor: HP VEX compiler, GCC VEX, LLVM, and Open64, and the compiler is responsible for scheduling independent instructions to be executed concurrently. The HP VEX compiler was chosen because it is more stable and robust than the other compilers.

A soft-float library (HAUSER, 2002) was used for floating-point operations, and the original input data from the benchmark suite was used. The floating-point benchmarks (*LUDCMP*, *Minver*, and *Qurt*) were compiled with LLVM (LATTNER; ADVE, 2004)

with the back-end modified to support the VEX ISA (JOST; NAZAR; CARRO, 2016), because the HP VEX compiler does not support soft-floating point. The benchmarks were compiled with the *-O3* flag and each one is discussed next.

- *Adaptive Differential Pulse-Code Modulation (ADPCM)*: completely structured code (i.e., no conditional branches, no exit from loop bodies) and contains loops;
- *CJPEG*: compresses an image file using the integer implementation of the Discrete Cosine Transform (DCT) method. Contains loops and arrays;
- *CRC*: cyclic redundancy check computation on 40 bytes of data. It has complex loops and many decision statements;
- *Discrete Fourier Transform (DFT)*: uses arrays and nested loops;
- *Engine*: engine control application. It has sequences of condition statements and loops;
- *Expint*: performs series expansion for computing an exponential integral function. Contains nested loops and arrays;
- *FIR*: finite impulse response filter (signal processing algorithms) over a 700 items long sample. It consists of an inner loop with varying number of iterations and loop-iteration dependent decisions;
- *JPEG*: JPEG 24-bit image decompression standard. Uses arrays and nested loops;
- *LU Decomposition (LUDCMP)*: performs calculations based on floating point arrays with the size of 50 elements;
- *Matrix multiplication*: multiply two 20x20 matrices. It has multiple calls to the same function, nested function calls, and triple-nested loops;
- *Minver*: inversion of floating point matrix. Floating value calculations in 3x3 matrix with nested loops;
- *NDES*: complex embedded code. It does bit manipulations, shifts, and array and matrix calculations;
- *POCSAG*: POCSAG paging communication protocols. Contains loops and several condition statements;
- *Qurt*: root computation of quadratic equations. The real and imaginary parts of the solution are stored in arrays;
- *Sums*: recursively executes multiple additions on an array;
- *x264*: contains arrays, matrices, and loops to calculate a sum of absolute differences of the H.264 video encoding standard.

For the polymorphic processor, the previous benchmarks are considered to be kernels for a larger application to simulate complex application behavior. Therefore, each application will execute these kernels several times. In order to define how many times each kernel will be executed, the actual number of iterations of complex parallel applications was considered. These numbers were obtained from: Dongarra, Heroux and Luszczek (2015), Seo, Jo and Lee (2011), Che et al. (2009), Petersen and Arbenz (2004), Quinn (2003), McCalpin (1995), Bhatt et al. (1992).

6.1.2 Temporal Duplication and Optimization Module Simulators

The duplication with rollback mechanism (Section 3.1.1) and the spatial duplication (Section 3.1.1.1), in addition to the PG mechanism and the ILP control module were all implemented in VHDL and these mechanisms were implemented to the static version of the ρ-VEX processor. Since then, the ρ-VEX was completely rewritten so it could allow the dynamic behavior that is described in Section 4.1.2. In order to exploit such characteristics of the new processor, we started porting these mechanisms to the new dynamic version. However, as the new core was developed from scratch, it was modified considerably and it would take a considerable amount of time to fully port these mechanisms. Thus, we decided to implement simulators to mimic the behavior of the dynamic core, allowing us to evaluate the proposed polymorphic processor. In order to estimate area and power of such design, the additional modules (e.g., caches, buffers, additional registers of the register file, etc.) were simulated in CACTI or synthesized in the Cadence compiler.

Two extra simulators were also implemented: one that simulates the temporal duplication (in addition to the spatial one), allowing more flexibility in the instruction duplication; and the second one not only mimics the behavior of the dynamic version of the ρ-VEX, but also implements an optimization algorithm that dynamically evaluates different core configurations and chooses the best one considering the trade-off between the axes of energy consumption, fault tolerance, and performance. Next, each of these simulators will be discussed.

- *Temporal Duplication Simulator:* The temporal duplication simulator was developed in Python and it receives as input the trace of the application that was executed in the ρ-VEX processor (which was modified to save this trace file). As the static version of the ρ-VEX processor that was used in this work does not have caches, a

cache model was included in the simulator, so we are able to also exploit those cycles that the core is idle waiting for the memory to duplicate instructions that were waiting in the issue buffer. Each parameter of the simulator can be tuned to allow the evaluation of different configurations, which includes the size of the buffers, issue-width, enable or disable the PG module, list of benchmarks to be simulated, and all configurations of the memory model, including associativity, block size, prefetch size, miss penalty, and cache size.

- *Optimization Module Simulator:* This simulator was developed to both mimic the dynamic behavior of the ρ-VEX and to choose the most appropriate core configuration for each benchmark during its execution. The parameters for this simulator are the list of benchmarks and the configurations that are going to be tested.

The remaining of this chapter is divided into three parts: first the results for the hybrid fault injector will be presented, followed by the results of the techniques that were developed for the static ρ-VEX processor. Finally, the techniques for the dynamic version and the polymorphic processor will be evaluated. As the temporal and spatial duplication mechanism was implemented in the dynamic version of the processor, which includes the cache memory model that was not available in the static version, its results will be evaluated in Section 6.4.1 (Polymorphic core section).

6.2 Hybrid Fault Injector Results

In this subsection, the accuracy comparison between RTL and gate-level fault injection is presented followed by the speedup evaluation of the Simbah-FI framework. The results for the fault injector were obtained from the 8-issue version of the ρ-VEX processor.

6.2.1 RTL vs Gate-level: Accuracy Comparison

We conducted a series of experiments to show the need for our hybrid simulator: if the accuracy of both levels were the same, one would only need to execute and inject faults to the RTL version, without the necessity of switching between the RTL and gate-level versions.

First of all, we assessed the number of signals. Considering the above-mentioned configuration of the ρ-VEX processor, the netlist has 16,128 signals for the target modules

Table 6.1: Accuracy Comparison between RTL and Gate-level Fault Injection

	RTL fault injection			Gate-level fault injection		
Benchmark	**Failure rate** (%)	**Control flow failure** (%)	**Data failure** (%)	**Failure rate** (%)	**Control flow failure** (%)	**Data failure** (%)
ADPCM	5.21	2.62	97.38	3.66	18.17	81.83
CJPEG	8.15	3.62	96.38	6.07	10.75	89.25
CRC	3.64	4.23	95.77	2.95	22.12	77.88
DFT	3.17	13.01	86.99	2.68	48.55	51.45
Average	4.71	4.78	94.04	3.64	21.40	73.55

Source: The Author

in the RTL simulation, while the gate-level simulation generates a netlist with 36,786 signals. Therefore, the gate-level simulation provides 2.28 times more information about the target circuit than the RTL one, as it has access to all internal signals after synthesis. By consequence, it is able to accurately represent the area distribution of the circuit, as it includes all internal signals of the functional units and other hardware structures. This means that each hardware module will have a different probability of being affected by a fault depending on its area and its internal circuitry (that can be more robust or sensitive to faults depending on the logic that is performed).

Moreover, to demonstrate the accuracy difference between these two descriptions, the first four benchmarks of the previous benchmark list were used for RTL fault injection, and they were compared to the gate-level fault injection. These results are presented in Table 6.1. On average, the failure rate for the RTL fault injection was 30% higher than the gate-level. In addition, RTL fault injection resulted in a huge difference in the distribution of the faults: control flow failures went from 21.4% to only 4.78%, and data failures from 73.55% to 94.04% on average. These experiments reinforce that RTL fault injection is not able to provide accurate results, requiring gate-level fault injection.

6.2.2 Fault Injection Performance

Table 6.2 presents the benchmarks' number of cycles, the simulation time for the RTL simulation, hybrid fault injector (Simbah-FI), and the fault injection in gate-level. The standard deviation for the simulation time is less than 1%. As aforementioned, behavioral RTL simulation is faster than gate-level simulation, however, it does not provide sufficient detail on internal signals for accurate fault injection.

Table 6.2: Number of Cycles and Simulation Time Comparison

		Simulation time (seconds)				
Benchmark	**Num. cycles**	**RTL simulation (NO fault injection)**	**Simbah-FI overhead vs. RTL sim. (seconds)**	**Gate-level fault injection**	**Simbah-FI**	**Speedup**
ADPCM	568	0.70	3.51	3.18	4.22	0.75
CJPEG	411	0.65	3.50	2.75	4.15	0.66
CRC	13,289	4.61	3.51	31.35	8.12	3.86
DFT	32,575	9.28	3.58	58.48	12.86	4.55
Engine	691,437	147.80	3.42	816.19	151.22	5.40
Expint	9,341	2.78	3.65	12.29	6.43	1.91
FIR	119,392	37.49	3.48	431.60	40.97	10.54
JPEG	1,448,615	396.66	3.26	3,240.02	399.92	8.10
LUDCMP	44,558	11.89	3.59	93.54	15.47	6.05
Matmult	111,050	29.65	3.49	258.31	33.15	7.79
Minver	12,224	3.57	3.62	24.36	7.19	3.39
NDES	28,527	7.99	3.58	53.96	11.57	4.66
POCSAG	18,926	6.21	3.54	49.45	9.75	5.07
Qurt	17,972	5.16	3.49	39.99	8.65	4.62
Sums	319	0.58	3.62	1.86	4.20	0.44
x264	15,089	5.60	3.48	48.87	9.08	5.38

Source: The Author

On the tested computer (Intel Core i7-7700K, 8GB DDR4), Simbah-FI can inject faults with an average overhead of 3.52 seconds over the original RTL simulation, but with the same accuracy as the gate-level fault injection. This overhead is a result of the time taken for the context switches between the two simulation levels (from RTL to gate-level and vice-versa) and to run the simulation at gate-level to inject the fault. Note that this overhead is fixed, regardless the application at hand or its original execution time.

For instance, the gate-level fault injection would take 3,240.02 seconds (54 minutes) to inject a single fault in the *JPEG* application. On the other hand, Simbah-FI reduces this time to 399.92 seconds (6.6 minutes), which is basically the original time spent for the RTL simulation plus 3.26 seconds, which are spent switching contexts and simulating at gate-level when the fault is injected. The same scenario can be observed for the *Engine* application, which would take 816.19 seconds (13.6 minutes) in gate-level fault injection, while the hybrid fault injection reduces this time to 151.22 seconds (2.5 minutes), with 3.42 seconds spent in the context switch.

As Modelsim is an event-driven simulator, the more complex the application (i.e., the more transitions the signals have), the higher will be its simulation time. Naturally,

the higher number of cycles, the higher the simulation time as well. Therefore, when executing an application in gate-level, the simulation time will be influenced by the following factors:

- The switching activity of the signals (i.e., the overall percentage of the transitions from '0' to '1' or from '1' to '0'), which depends on the application that is being executed;
- The number of cycles of the target application;
- The number of signals to be simulated, which is be higher than RTL simulation.

Simbah-FI's speedup varies from 0.44 (extremely simple and small benchmark) to 10.54 times (complex benchmark). In the case of small benchmarks, the overhead of the context switch is higher than executing the whole application in gate-level mode. However, for large benchmarks, Simbah-FI speeds up the execution by several times and the overhead of the context switch turns out to be negligible.

Table 6.3 presents the number of transitions that were performed to execute each benchmark in gate-level, the maximum number of transitions that such benchmark could generate if all signals were transitioned in every cycle, and the resulting switching activity. The benchmark with the maximum speedup was the FIR, with a speedup of 10.54 times, and as it can be observed in Table 6.3, it is the benchmark with the highest switching activity (12.52%), which also influences in the gate-level simulation's speed. Both the number of transitions and the number of cycles have a very strong Pearson's correlation with the simulation time, with a correlation factor of 0.99 and 0.97, respectively.

6.3 Adaptive ρ-VEX Processor

For the adaptive processor (i.e., adaptive optimization techniques that were implemented in the *static* version of the ρ-VEX processor), the results are presented for the 8-issue design. The 4-issue is evaluated only when comparing the full duplication mechanism, which was applied to the 4-issue, having a total of eight pipelanes. In addition, two versions are assessed: without and with power gating, as follows.

Table 6.3: Number of Transitions and Switching Activity from Gate-level Simulation

Benchmark	**Number of transitions**	**Maximum transitions**	**Switching activity (%)**
ADPCM	1,630,293	18,343,407	8.89
CJPEG	1,375,953	13,352,220	10.31
CRC	28,099,961	422,152,689	6.66
DFT	74,403,051	1,035,877,944	7.18
Engine	1,145,735,577	21,981,759,786	5.21
Expint	12,159,231	289,488,846	4.20
FIR	445,066,119	3,553,534,398	12.52
JPEG	3,836,163,241	46,053,205,584	8.33
LUDCMP	98,440,457	1,416,829,497	6.95
Matrix Mult.	263,947,463	3,529,881,894	7.48
Minver	24,384,221	388,899,303	6.27
NDES	62,392,015	874,506,828	7.13
POCSAG	44,905,749	601,962,585	7.46
Qurt	40,517,361	571,633,971	7.09
Sums	383,967	10,427,448	3.68
x264	53,919,303	479,980,518	11.23

Source: The Author

6.3.1 Spatial Duplication Without Power Gating

Next, the spatial duplication fault tolerance techniques (i.e., spatial duplication with and without ILP control) are evaluated in terms of performance, fault tolerance, energy, power, and area.

6.3.1.1 Failure Rate and Performance

Table 6.4 presents the failure rate and performance of the chosen applications in several configurations (4-issue full duplication; and spatial duplication with and without ILP control, using $Threshold = 1.75$, 2, or 2.5 and $Threshold = 1$), and unprotected versions (4- and 8-issue). Note that, in some benchmarks, results of the spatial duplication with ILP reduction are not shown for a threshold greater than 1, since the failure rate does not decrease significantly. On average, the unprotected processors have a failure rate of 6.61% and 3.73% for the 4- and 8-issue, respectively. These failure rates are

Table 6.4: Failure Rate and Performance Degradation Comparison

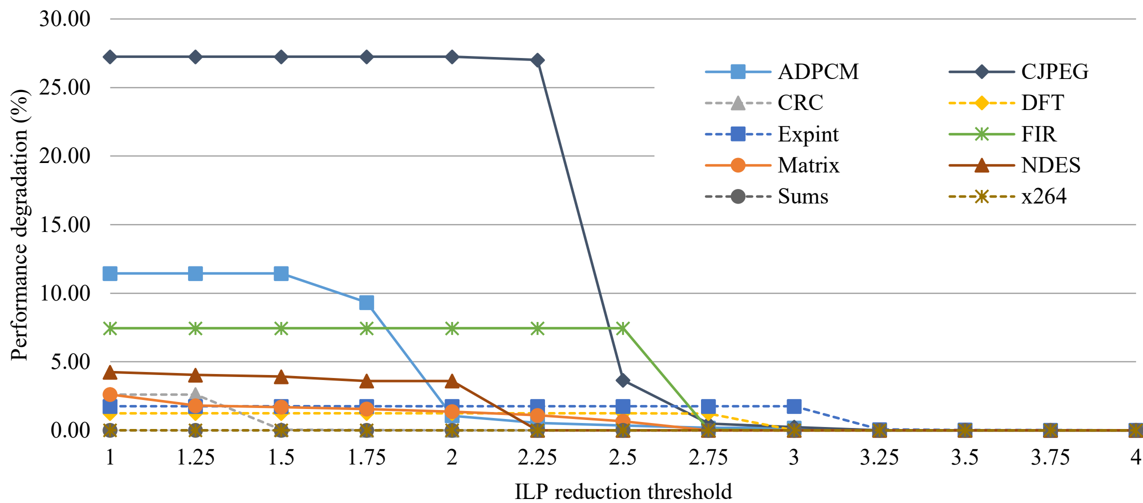
		Unprot.	Prot.	Unprot.	Protected						
		4-issue	Full dup.	8-issue	Spatial Dup.	Threshold					
ADPCM	Failure rate (%)	6.93	0.06	3.66	0.66	T = 1	0.59	T = 1.75	0.65	T = 2	0.66
	Exec. Cycles	571	571	568	568	T = 1	633	T = 1.75	621	T = 2	574
CJPEG	Failure rate (%)	9.55	0.02	6.07	2.33	T = 1	0.79	T = 2.5	2.12		
	Exec. Cycles	508	508	411	411	T = 1	523	T = 2.5	426		
CRC	Failure rate (%)	5.20	0.06	2.95	0.33	T = 1	0.32				
	Exec. Cycles	13289	13289	13270	13270	T = 1	13616				
DFT	Failure rate (%)	4.63	0.07	2.68	0.38	T = 1	0.15				
	Exec. Cycles	35072	35072	32575	32575	T = 1	32979				
Expint	Failure rate (%)	4.21	0.05	2.37	0.13	T = 1	0.13				
	Exec. Cycles	9341	9341	9097	9097	T = 1	9257				
FIR	Failure rate (%)	10.94	0.04	5.93	1.21	T = 1	0.93				
	Exec. Cycles	119392	119392	111769	111769	T = 1	120095				
Matrix Mul.	Failure rate (%)	9.91	0.08	5.68	1.30	T = 1	0.17	T = 2	0.53		
	Exec. Cycles	111050	111050	111025	111025	T = 1	113929	T = 2	112547		
NDES	Failure rate (%)	3.99	0.04	2.09	0.42	T = 1	0.24				
	Exec. Cycles	28527	28527	27499	27499	T = 1	28667				
Sums	Failure rate (%)	5.52	0.04	2.96	0.37	T = 1	0.37				
	Exec. Cycles	332	332	319	319	T = 1	319				
x264	Failure rate (%)	5.21	0.09	2.94	0.33	T = 1	0.33				
	Exec. Cycles	15102	15102	15089	15089	T = 1	15090				

Source: The Author

similar to other commercial processors, such as the IBM Power6 microprocessor, which has masking of 95% (RAMACHANDRAN et al., 2008).

The protected versions present the following failure rate: 0.05% for the full duplication, 0.75% for the spatial duplication, and 0.4% ($Threshold = 1$) for the spatial with ILP reduction. The unprotected 8-issue has a lower failure rate than the unprotected 4-issue due to the elevated number of NOPs in the VLIW instruction; therefore, the prob-

Figure 6.1: Performance Degradation when Varying the ILP Reduction Threshold



Source: The Author

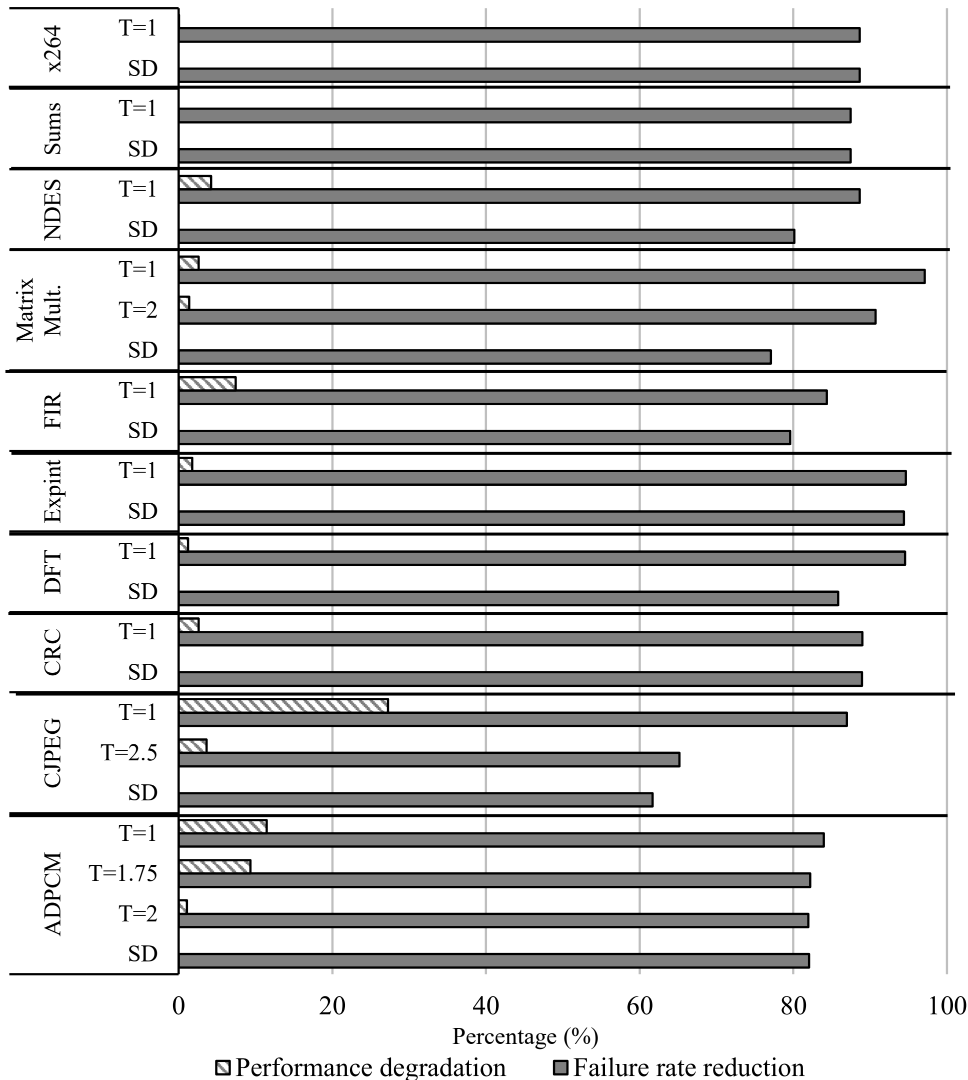
ability of a flipping bit affect the result of an instruction is lower than on the 4-issue configuration. The failure rate comprises the detection and correction; all errors are detected, but not all can be corrected in time. Even though there is no latency for the fault detection, the circuit delay may prevent the memory or the register file to be blocked for writing in time. In these specific cases, the memory and register file are blocked a moment after the incorrect data began to be written, hence, generating wrong results in some cases.

The only approach that affects the performance of the applications is the spatial with ILP reduction; all others have no performance overhead. Figure 6.1 presents the performance degradation (*Y axis*) according to the threshold (*X axis*). It varies from zero to 27.25% with the threshold equal to 1 (the lowest possible value). As we increase the threshold, the performance degradation is reduced, being negligible (less than 1%) at 3.5. Therefore, performance is degraded as the threshold is reduced; on the other hand, fault tolerance is increased.

Figure 6.2 depicts the trade-off between failure rate and performance of the spatial duplication without and with ILP reduction, with different thresholds, normalized to the unprotected 8-issue version. The T stands for *"Threshold"* for the spatial with ILP reduction. The failure rate reduction varies from 61.68% (*CJPEG* executing with spatial duplication without ILP control) to 97.09% (*matrix multiplication* on spatial with $T = 1$), while performance degradation reaches up to 27.25% (*CJPEG* on spatial with $T = 1$), when compared to the unprotected version.

In the *CJPEG* benchmark, for instance, when switching from threshold 2.5 to 1, the failure rate is further reduced from 65.65% to 86.96% (when compared to the unpro-

Figure 6.2: Spatial Duplication with ILP Control Normalized to the Unprotected Version



Source: The Author

tected 8-issue), and the performance degrades from 3.65% to 27.25% (also compared to the unprotected version). Therefore, for this benchmark, there is a large improvement in fault tolerance, which comes at the high cost of performance. For benchmarks such as the *ADPCM*, the performance degradation of changing the threshold from 2 to 1 is greatly increased (1.06% to 11.44%), while the fault tolerance improvement is minimal (81.96% to 83.98%). On the other hand, other benchmarks, such as the *matrix multiplication*, present high fault tolerance improvements with low impact on performance: with 2.62% of performance degradation, the failure rate reduction goes from 77.08% (no ILP control) to 97.09% ($Threshold = 1$).

6.3.1.2 Dynamic Threshold Adaptation

In this subsection, the dynamic threshold adaptation is exploited when executing a given application. This approach will adapt the threshold in order to cope with a given Acceptable Failure Rate Variation (AFRV) defined a priori by the designer before execution. In this experiment, the threshold starts at its lowest value, and is gradually increased in order to reduce the performance degradation according to the AFRV: If the failure rate increases more than the AFRV, the threshold will be restored to its last value in order to maintain the failure rate within the bounds defined by the user.

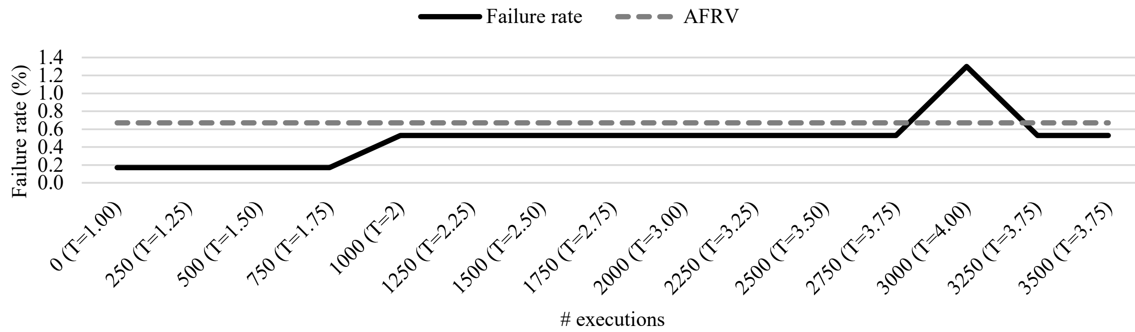
Figure 6.3 depicts this approach being applied to three benchmarks. The application is executed in batches of 250 times, and for each batch, the threshold is gradually increased if the failure rate does not increase more than the AFRV (in this example, AFRV=0.5%). Figure 6.3(a) presents the *matrix multiplication* benchmark, in which the threshold is gradually increased from 1 to 3.75 without reaching the AFRV value. However, when the threshold is increased to 4, the failure rate surpasses the AFRV, which triggers the threshold reduction back to 3.75 and restore the acceptable failure rate defined by the user. In Figure 6.3(b) (*ADPCM*) the threshold is increased from 1 to 4 without reaching the AFRV limit, and Figure 6.3(c) (*CJPEG*) reaches the AFRV value with a threshold equal to 2.5, which is reduced back to 2.25 for the next executions.

Figure 6.4 presents the performance improvement and the failure rate variation that the dynamic threshold provides when compared to the $Threshold = 1$. In the *ADPCM* benchmark, the dynamic threshold is able to improve the performance by 11.44% with a failure rate that varies from 0.59 (T=1) to 0.66; for the *matrix multiplication*: 2.61% speedup with 0.17 to 0.53% failure rate variation; finally, for *CJPEG*, 0.2% speedup and no failure rate variation. Therefore, the dynamic threshold approach can be used to reduce the performance overhead and still maintain the failure rate within the bounds defined by the user.

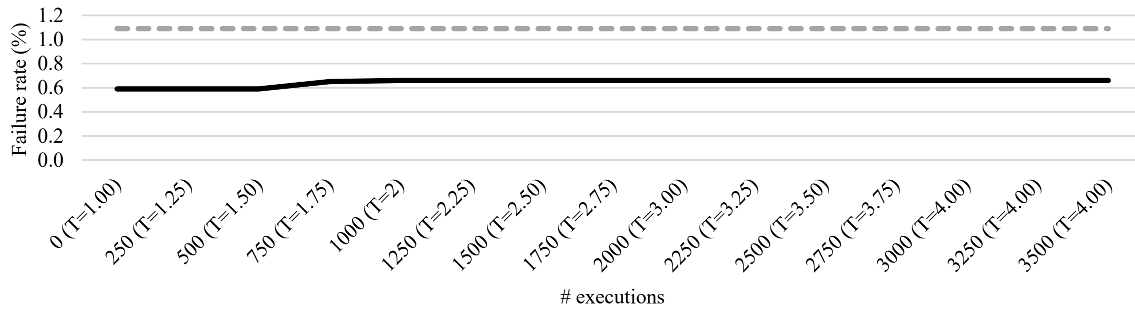
6.3.1.3 Area, Power Dissipation, and Energy Consumption

Table 6.5 presents the area (both FPGA and ASIC versions) and power consumption (ASIC only) for all VLIW configurations. As it can be observed, the overhead for the 4-issue full duplication is small in terms of area and power dissipation when compared to the unprotected 4-issue, even though the pipelines are duplicated (the area for each checker is less than 1%). The area overhead for the FPGA is 30% in Look-Up Tables

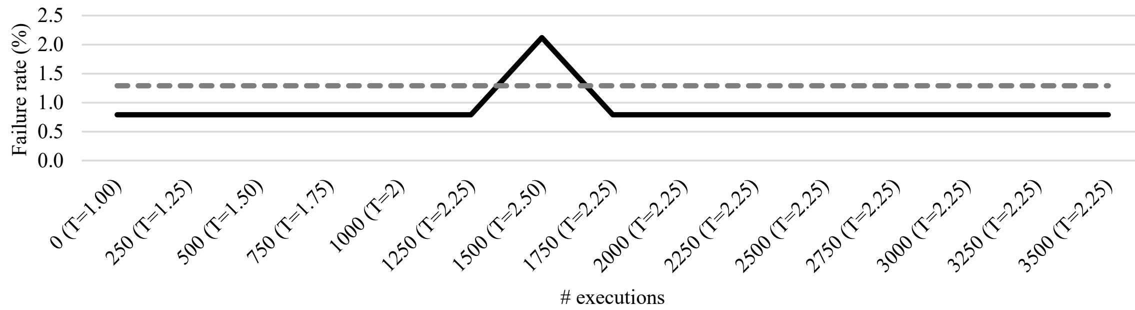
Figure 6.3: Dynamic Threshold Adaptation



(a) Matrix Multiplication



(b) ADPCM



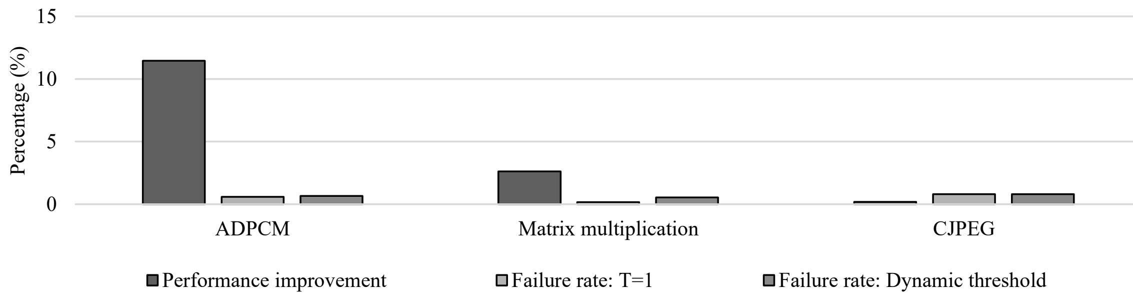
(c) CJPEG

Source: The Author

(LUTs) and 35% in registers; for the ASIC is 50%, while the power dissipation overhead is 35%. For the spatial duplication without ILP control, the overhead is extremely low: 4.6% for the FPGA and 3.5% for the ASIC, while the power dissipation overhead is 5.3%. The overhead for the spatial approach with ILP reduction is higher because of extra control circuitry. However, the overhead is still low when compared to other techniques, being 18.3% for the FPGA, 14.1% for the ASIC and 27.6% in power dissipation.

Therefore, each approach has its advantages depending on the target application and its requirements. For instance, the spatial duplication approach is able to exploit idle hardware for low ILP applications in a completely transparent manner with extremely low overhead. For high ILP applications, the spatial with ILP reduction can guarantee a certain amount of duplicated instructions and therefore fault tolerance. Even though

Figure 6.4: Performance Improvement and Failure Rate Variation for the Dynamic Threshold Approach when Compared to the Threshold=1



Source: The Author

Table 6.5: Area and Power Dissipation Comparison

		FPGA		ASIC	
		Registers	LUTs	Cells	Power dissipation (nW)
Unprotected	4-issue	3,058	16,006	28,041	2,298,962.51
	8-issue	3,974	35,075	66,967	7,484,818.25
Protected	Full. Dup. (4-issue)	4,102	20,819	42,121	3,109,613.33
	Spatial Dup.	4,206	36,672	69,305	7,878,161.31
	Spatial with ILP reduction	4,834	41,485	76,407	9,553,048.27

Source: The Author

the spatial duplication is able to exploit those idle slots, it is limited to duplicate the instructions within the same bundle (in the same cycle). The temporal duplication takes this technique one step further, by also allowing instructions that cannot be duplicated in the same bundle to be stored in a buffer for execution in another cycle. The latter was implemented in the dynamic version of the processor (which was simulated in this work) and its results are presented in the Section 6.4.1.

6.3.1.4 Comparison With Other Fault Tolerance Techniques

As already mentioned, the main limitations of software-based redundancy are the increase in the code size, energy consumption and performance overheads that come with it. On the other hand, hardware-based redundancy approaches increase area, power dissipation with little or no performance overhead. When compared to the previous works discussed in Chapter 2, the proposed fault tolerance approaches for the static version of the ρ-VEX processor have low detection latency, low performance and energy consumption overhead, and do not modify the application's code, which is depicted in Table 6.6.

Table 6.6: Fault Tolerance Techniques Comparison

Technique	Area Overhead	Performance Degradation	Power Dissipation Overhead	Energy Consumption Overhead	Code Size Increase
Spatial Dup.	3.5%	∼0%	5.2%	30-45%	0%
Spatial Dup. with ILP control	14.1%	∼0-27.25%	27.6%	66-88%	0%
DMR with Rollback (XIAOGUANG et al., 2015) (YANG; KWAK, 2010)	0%	51-100%	0%	>0% N/A	100%
TMR	200%	∼0%	∼200%	∼200%	0%
Partial TMR (CHEN; LEU, 2010)	100%	0.6-34.3%	∼100%	>100%	0%
Reduced TMR (SCHÖLZEL, 2007)	100%	0-100%	∼100%	>100%	>0%
TFT (WALI et al., 2015)	50%	1%	88%	>0% N/A	0%
Reduced TMR - SW (HU et al., 2005)	0%	30-60%	0%	>0% N/A	100%
Flip-flops TMR (ANJAM; WONG, 2013)	200%	∼0%	∼200%	∼200%	0%
DWC - SW (BOLCHINI, 2003) (HU et al., 2009)	0%	28-106%	0%	>0% N/A	109-217%
DWC Opt. - SW (MITROPOULOU; PORPODAS; CINTRA, 2014)	0%	29%	0%	>0% N/A	100-150%

Source: The Author

For instance, the fault tolerance approach developed in the scope of this thesis has lower detection latency and simpler rollback control structure when compared to duplication with rollback mechanisms that use checkpoints; it is able to detect and correct faults with less hardware cost than TMR approaches (even when compared to those that apply TMR to only some modules of the processor) when compared to other hardware-based techniques (in **bold**). Software-based techniques (in *italic*) naturally do not affect the area nor the power dissipation, but they create a performance overhead and increase the code size, both affecting total energy consumption of the system, as the application will take longer to execute and the memory will be more stressed (even though none of the software-based cited approaches evaluated the energy consumption overhead); and it is suitable for both FPGAs and ASICs, as it does not need hardware reconfiguration.

Table 6.7: Evaluated Designs and Their Techniques for Fault Tolerance and PG on the Static Processor

Design	FT technique	PG	ILP Control
Unprotected	None		
Unprotected with PG	None	✓	
Dupl. with ILP ctrl	Dupl. with rollback		✓
Dupl. and PG w/o ILP ctrl	Dupl. with rollback	✓	
Dupl. and PG with ILP ctrl	Dupl. with rollback	✓	✓
TMR	Triplication		

Source: The Author

6.3.2 Spatial Duplication With Power Gating

In this section, the results for the PG mechanism together with fault tolerance techniques are presented in terms of failure rate, performance, energy consumption, area, and Energy-Delay-Failure Product (which will be explained later). Table 6.7 presents the six designs that were evaluated and which techniques each design uses, as discussed next.

1. ***Unprotected*** : Baseline processor, without fault tolerance and power gating mechanisms (prioritizes performance);
2. ***Unprotected with PG*** : Baseline processor, without fault tolerance and with power gating (prioritizes energy consumption);
3. ***Duplication with ILP control*** : fault tolerant only, without power gating. It duplicates instructions whenever possible and splits bundles to increase fault tolerance (prioritizes fault tolerance);
4. ***Duplication and PG without ILP control*** : fault tolerant and with power gating (it weighs fault tolerance and power gating equally);
5. ***Duplication and PG with ILP control*** : fault tolerant, with PG and with ILP control (it weighs more energy consumption than fault tolerance);
6. ***Triple Modular Redundancy (TMR)*** : included in the experiments for comparison purposes only. The processor is triplicated and the results are voted in order to mask the faults, using the traditional TMR technique (it is only fault tolerant).

The absolute results in terms of performance, energy consumption, and failure rate for these designs are presented in Table 6.8, and the relative numbers are explained next.

Table 6.8: Performance, Energy Consumption, and Fault Tolerance Comparison

Bench.	Metric	1) Unprot. (baseline)	2) Unprot. with PG	3) Dup. with ILP control	4) Dup. and PG w/o ILP control	5) Dup. and PG w/ ILP control	6) TMR
ADPCM	Perf. (cycles)	568	596	633	596	630	568
	Energy cons. (J)	4.83E-08	4.62E-08	9.39E-08	6.88E-08	6.32E-08	1.45E-07
	Failure rate (%)	3.66	3.66	0.59	0.89	0.90	0.00
DFT	Perf. (cycles)	32,575	41,630	32,979	41,630	42,024	32,575
	Energy cons. (J)	2.23E-06	1.45E-06	4.09E-06	2.44E-06	2.34E-06	6.68E-06
	Failure rate (%)	2.68	2.68	0.15	0.33	0.36	0.00
Engine	Perf. (cycles)	691,437	724,482	695,852	724,482	723,553	691,437
	Energy cons. (J)	4.66E-05	3.08E-05	8.54E-05	5.45E-05	5.07E-05	1.40E-04
	Failure rate (%)	1.80	1.80	0.24	0.27	0.34	0.00
Expint	Perf. (cycles)	9,097	9,305	9,257	9,305	9,509	9,097
	Energy cons. (J)	6.36E-07	4.10E-07	1.17E-06	6.53E-07	6.45E-07	1.91E-06
	Failure rate (%)	2.37	2.37	0.07	0.09	0.10	0.00
JPEG	Perf. (cycles)	1,448,615	1,620,785	1,572,092	1,620,785	1,745,014	1,448,615
	Energy cons. (J)	1.13E-04	8.02E-05	2.10E-04	1.29E-04	1.18E-04	3.38E-04
	Failure rate (%)	2.15	2.15	0.12	0.22	0.22	0.00
LUDCMP	Perf. (cycles)	44,558	47,222	47,545	47,222	50,076	44,558
	Energy cons. (J)	3.13E-06	2.25E-06	5.93E-06	3.86E-06	3.72E-06	9.39E-06
	Failure rate (%)	2.49	2.49	0.19	0.35	0.40	0.00
Mat.Mul.	Perf. (cycles)	111,025	117,563	113,929	117,563	118,057	111,025
	Energy cons. (J)	7.50E-06	5.79E-06	1.39E-05	9.70E-06	8.54E-06	2.25E-05
	Failure rate (%)	5.68	5.68	0.17	0.25	0.25	0.00
Minver	Perf. (cycles)	12,224	12,779	12,767	12,779	13,063	12,224
	Energy cons. (J)	8.34E-07	6.40E-07	1.56E-06	1.13E-06	1.10E-06	2.50E-06
	Failure rate (%)	2.77	2.77	0.34	0.49	0.63	0.00
POCSAG	Perf. (cycles)	18,926	21,247	21,027	21,247	23,302	18,926
	Energy cons. (J)	1.64E-06	1.19E-06	3.13E-06	1.94E-06	1.80E-06	4.91E-06
	Failure rate (%)	1.97	1.97	0.15	0.36	0.36	0.00
Qurt	Perf. (cycles)	17,972	18,691	19,016	18,691	19,554	17,972
	Energy cons. (J)	1.25E-06	9.72E-07	2.37E-06	1.69E-06	1.64E-06	3.76E-06
	Failure rate (%)	1.80	1.80	0.12	0.22	0.23	0.00
Sums	Perf. (cycles)	319	321	319	321	321	319
	Energy cons. (J)	2.00E-08	1.66E-08	3.68E-08	3.04E-08	3.04E-08	6.01E-08
	Failure rate (%)	2.96	2.96	0.27	0.30	0.30	0.00
x264	Perf. (cycles)	15,089	20,203	15,090	20,203	20,203	15,089
	Energy cons. (J)	1.08E-06	7.31E-07	2.00E-06	1.18E-06	1.18E-06	3.23E-06
	Failure rate (%)	2.94	2.94	0.33	0.45	0.55	0.00

Source: The Author

6.3.2.1 Performance

Figure 6.5(a) presents the relative performance overhead of the five versions when compared to the ***Unprotected***. As aforementioned, when power gating is being used, and a given BB starts its execution, a given functional unit may not be completely turned on and ready to execute instructions. When this occurs, the processor must be stalled, incurring in performance overhead. This is the case for the ***Unprotected with PG*** and ***Duplication and PG without ILP control***. Both present an average performance overhead of 9.5%, varying from 0.6% to 33.9%, as these designs do not have the ILP control mechanism activated. When the ILP control is activated in the ***Duplication and PG with ILP control***

version, more energy is saved as the power gating phases will be extended, and this design incurs in an average performance overhead of 13%. The ***Duplication with ILP control***, which does not use PG, has an average overhead of 4.4%, varying from zero to 11.4%. Finally, the ***TMR*** maintains the performance of the unprotected processor, with a huge overhead in energy, which will be discussed next.

6.3.2.2 Energy and Area

Figure 6.5(b) shows the relative energy consumption considering the ***Unprotected*** version as baseline once more. When applying power gating to the baseline version (2), which does not have fault-tolerance mechanisms, the average energy consumption is reduced by 26.3%, varying from 4.3% to 35.5%. As aforementioned, applying fault-tolerance mechanisms increase the energy consumption of the design, resulting in an average overhead of 200% in the ***TMR*** (since the processor is triplicated), and 86.9% (from 83.4% to 94.5%) in the ***Duplication with ILP control*** version.

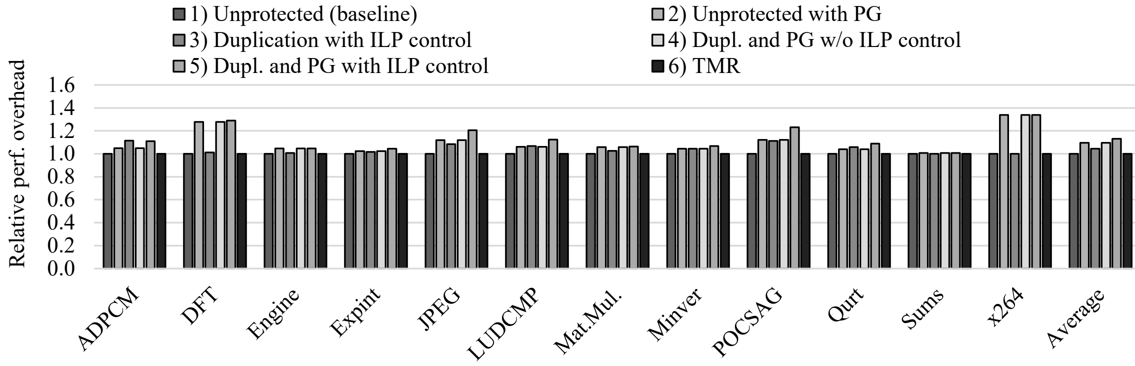
When compared to the ***Duplication with ILP control***, the ***Duplication and PG without ILP control*** is able to reduce the energy consumption by 34%, on average; and by 37.2%, in the ***Duplication and PG with ILP control*** (from 17% to 44%). Therefore, the ***Dupl. and PG*** approaches are able to significantly reduce the energy consumption when compared to a protected version of the processor. Even when compared to the ***Unprotected*** design (that does not have any fault-tolerance mechanism), the ***Dupl. and PG*** versions consume, on average, only 23.3% and 17.4% more energy, for the version *without* and *with* ILP control, respectively (while the non-fully adaptive ***Duplication with ILP control*** consumes 86.9% more).

The additional modules of the proposed design with duplication, PG and ILP control incur an area overhead of only 15.05% compared to the unprotected version, which is low considering that we have mechanisms for fault tolerance, energy optimization, and performance management all implemented in a single processor.

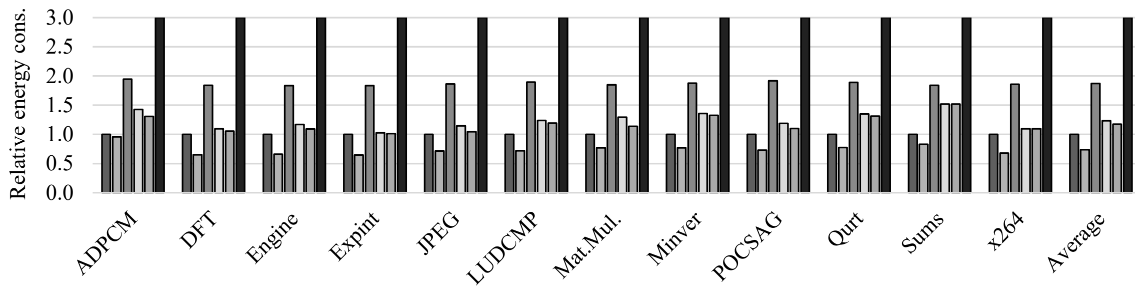
6.3.2.3 Failure Rate

Figure 6.5(c) depicts the failure rate of the chosen benchmarks for each design after the fault injection campaign. The ***Unprotected with PG*** maintained the failure rate of the ***Unprotected*** design, as no fault-tolerance mechanism is applied. Therefore, the probability of a failure occurring remains the same. However, an application being executed with PG will take more time to be executed, which will increase the time it is exposed

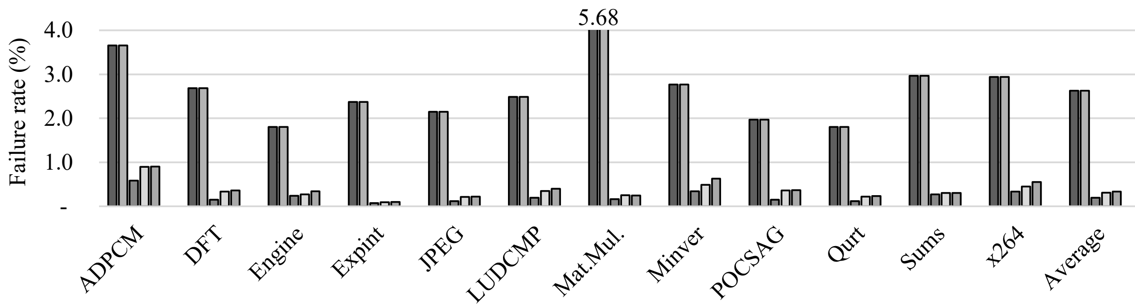
Figure 6.5: Performance, Energy Overhead, Failure Rate, and EDFP Comparison



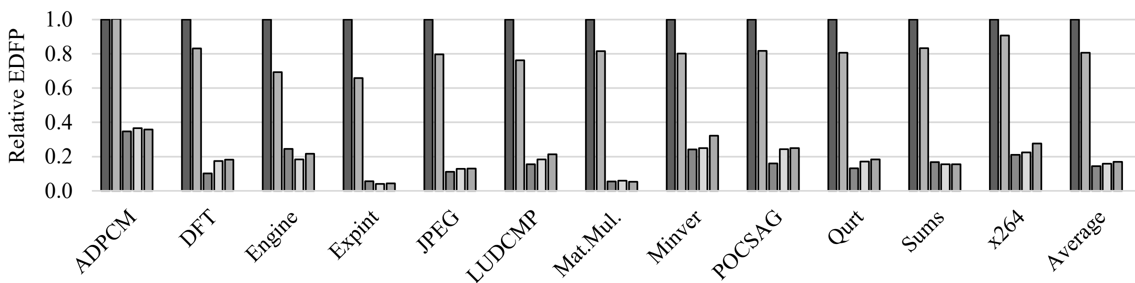
(a) Relative Performance Overhead Compared to the Unprotected Version



(b) Relative Energy Consumption Compared to the Unprotected Version



(c) Failure Rate Comparison



(d) Relative Energy-Delay-Failure Product (EDFP) Compared to the Unprotected Processor

Source: The Author

to energetic particles. As expected, all the other versions present lower levels of failure rate than the ***Unprotected***. As aforementioned, the ***Unprotected*** has, on average, 2.63% of failure rate; while the ***Duplication with ILP control***, 0.20%; the ***Duplication and PG without ILP control***, 0.31%, and the ***Duplication and PG with ILP control*** 0.34%.

The ***TMR*** executes every instruction three times and votes the correct result, so

it is able to mask all single faults that are injected (i.e., 0% of failure rate). Obviously, it is the best alternative when it comes to fault tolerance only. However, it has a huge overhead in energy and area. On the other hand, the ***Dupl. and PG*** versions trade-off energy consumption, performance, and fault tolerance. Therefore, these versions have a slightly higher failure rate when compared to the techniques that focus only on fault tolerance, because they will not duplicate every single instruction of the program in order to allow energy optimization through power gating.

6.3.2.4 EDFP - Energy-Delay-Failure Product

To better analyze the trade-off among all axes (energy consumption, performance, and fault tolerance) for each design, we extended the equation for Energy-Delay Product (EDP) to Energy-Delay-Failure Product (EDFP). The EDFP is the product among energy consumption, performance, and the failure rate of a given application running on a certain processor configuration. Figure 6.5(d) presents the relative EDFP when compared to the baseline configuration (i.e., the ***Unprotected*** processor). Note that the lower the EDFP, the better, as the goal is to reduce the energy consumption, delay (i.e., performance overhead) and failure rate. In addition, note that the EDFP for the ***TMR*** design will always be zero because such configuration masks all single faults in the processor. Therefore, when the processor is completely protected against transient faults (including protected checkers and voters), it is not possible to use EDFP to evaluate the trade-off as the failure rate would be zero. This boundary-value imprecision of the EDFP metric occurs also in other well-established fault-tolerance metrics such as the Mean Instructions to Failure (MITF) (WEAVER et al., 2004) and MWTF (REIS et al., 2005a), which are metrics that evaluate the performance-reliability trade-off. However, applying ***TMR*** comes with a huge overhead in area and energy consumption (200%). The other designs are compared in terms of EDFP next.

The average relative EDFP is of 0.81 for the ***Unprotected with PG***, which means that the energy optimization is proportionally higher than the performance overhead for such design. For the ***Duplication with ILP control*** version, even though the energy consumption is increased when compared to the baseline processor, the failure rate reduction compensates such overhead, resulting in an average EDFP of 0.14. The ***Dupl. and PG*** versions (4)-(5) have an average EDFP of 0.16 and 0.17, respectively. Therefore, these designs efficiently trade-off the target metrics in order to balance the axes of fault tolerance, energy consumption, and performance.

6.4 Polymorphic ρ-VEX Processor

6.4.1 Temporal and Spatial Duplication Mechanism

The EDFP metric can be used to evaluate the trade-offs between fault tolerance, performance, and energy consumption, however, only for the same configuration of the processor (i.e., 8-issue, or 4-issue, or 2-issue). In order to compare the reliability of different issue-widths (that have different sensitive area and execution time), the MWTF metric must be considered. The latter was previously presented in Section 4.3.1 and allows the evaluation of the trade-off between fault tolerance and performance while taking into account the applications' execution time and the sensitive area of the processor. As aforementioned, this metric was extended to the MWPUETF, which is a combination of the MWTF with the energy consumption of the application, allowing the assessment of the trade-off between these three axes, with different issue-widths.

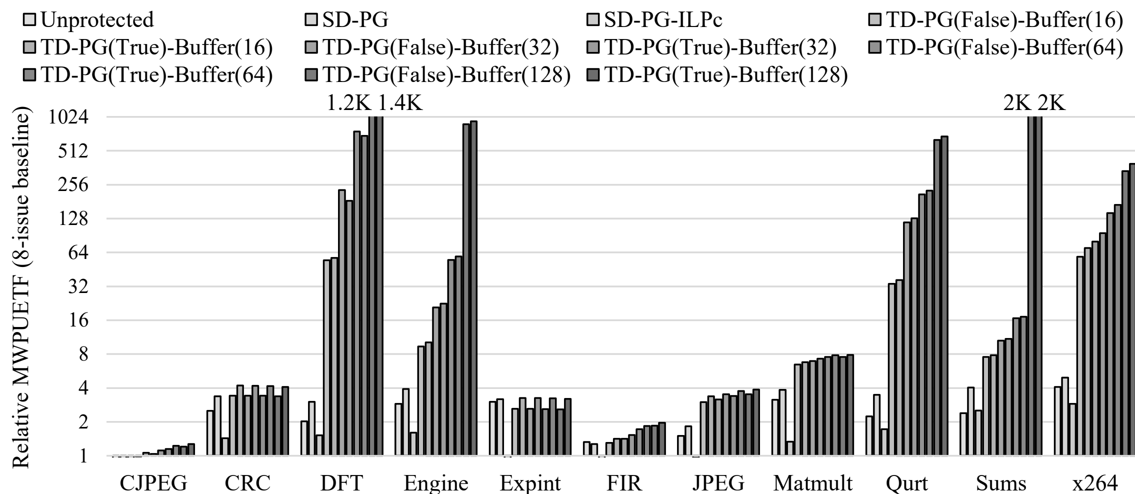
6.4.1.1 Mean Work Per Unit of Energy to Failure - MWPUETF

For the evaluation of the temporal and spatial duplication mechanism, the processor is used in a static configuration and the 2, 4, and 8-issue configurations are assessed. In Section 6.4.2, the dynamic behavior of the core will be further explored.

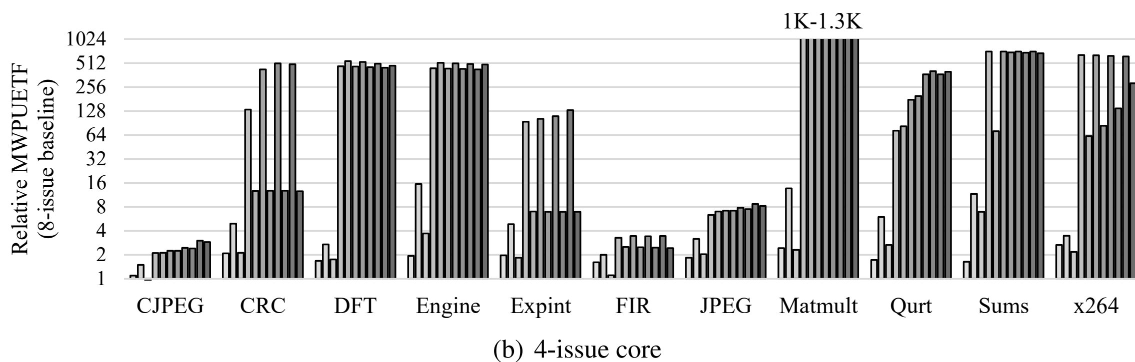
Figure 6.6 shows the MWPUETF for each issue-width (2, 4, and 8). In this and in the next figures, *SD* means Spatial Duplication, *PG* Power Gating, *ILPc* ILP control, and *TD* Temporal Duplication. In addition, it has the 8-issue as the baseline, in order to evaluate the relation between different issue-widths.

For each configuration of the temporal duplication, there are two extra options: enable or disable the power gating, and the buffer size. One more configuration was also studied, a *timeout* mechanism, which would discard those instructions that are waiting for too long in the buffer, in order to free space for newer instructions. On the other hand, this mechanism did not present many improvements when compared to not having this mechanism, which means that using low values for the timeout would imply in too much duplicated instructions being discarded (therefore, reducing the reliability), and a high value for the timeout would result in a similar behavior when the timeout is not applied. In rare cases this mechanism presented an improvement in fault tolerance: by not duplicating an instruction that was still pending to be executed, the bundle could now be committed and another bundle could be stored for duplication in the buffer. However,

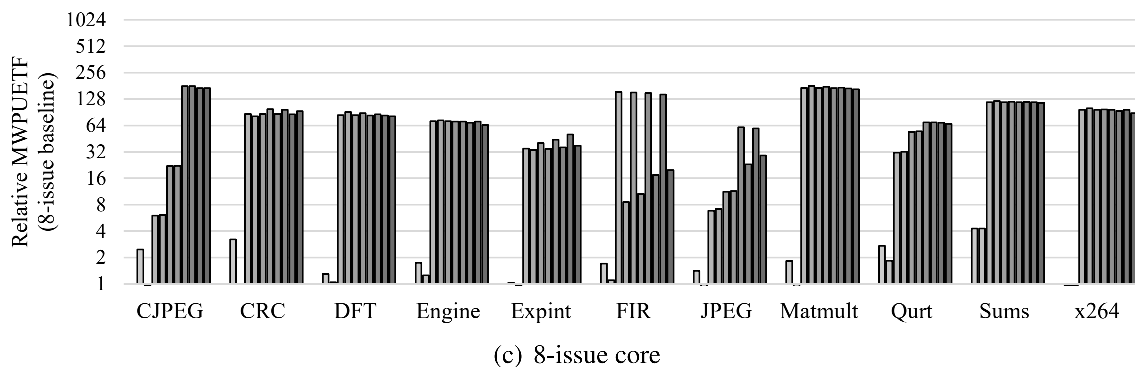
Figure 6.6: MWPUETF Comparison



(a) 2-issue core



(b) 4-issue core



(c) 8-issue core

Source: The Author

as these results were not significant, they were omitted from this section. In the current work, every instruction is given the same criticality, which means that there is no priority when scheduling the duplicated instructions, the scheduling is done in a FIFO fashion. As future work, the sensitivity of the instructions will be assessed, so the timeout mechanism may become more important as one can discard low priority instructions in order to free space to duplicate instructions that are more critical.

In this figure (Figure 6.6 - note that the Y axis is in logarithmic scale), the first observation is that it is not possible to find a single configuration (in terms of issue-width)

that will always be better than the others considering all axes and benchmarks. For instance, for the *CJPEG*, the 8-issue delivers the best MWPUETF when the temporal duplication mechanism is applied, the same goes for the *CRC* with the 4-issue and the *DFT* with the 2-issue. In addition, for most benchmarks, all configurations of the temporal duplication are able to deliver a better result than when only spatial duplication is applied, because it is able to exploit those slots that the spatial duplication is not able to use for duplication, even though the power dissipation of the temporal duplication approach is higher because of the additional buffers and control logic. By applying temporal duplication, the MWPUETF can be improved by up to 2K times (*Sums* with a buffer of 128) when compared to the unprotected processor.

By increasing the buffer size, more instructions can be stored for duplication, however, more power will be dissipated by these buffers. Therefore, each benchmark will have an ideal buffer size considering its behavior, increasing the buffer after the point where the benchmark effectively uses it will only result in increased power dissipation, while reducing the size of the buffer may affect the reliability as fewer instructions will be stored for the temporal duplication. A similar reasoning can be applied to the PG mechanism, which will turn off more hardware, at the cost of reducing the number of duplicated instructions in some cases.

Table 6.9 depicts the best configuration for each benchmark considering the MWPUETF metric. Each application has an ideal configuration and issue-width that needs to be applied in order to provide the best trade-off among all axes. For these benchmarks, the 8-issue is the best one for three benchmarks with a buffer size of 64 (with and without PG) and 16 (without PG). The 4-issue is the best configuration for four benchmarks with a buffer size varying from 16 to 128. Finally, the 2-issue is the best alternative for four benchmarks with a buffer size of 128. When compared to the other issue-widths, the 2-issue has fewer empty slots (NOPs), because the compiler is able to find instructions that are not independent of each other in several cases. When the issue-width is increased to 8, the compiler often inserts NOPs to fill the slots that could not be used, as already discussed. Therefore, the 2-issue ends up using more positions from the buffer, while in the 8-issue a buffer of 16 is able to deliver the best outcome for the *FIR* benchmark.

6.4.1.2 Duplication ratio

Tables 6.10, 6.11, and 6.12 present the percentage of instructions that were duplicated in each of the considered configurations. In Table 6.10, the 2-issue requires a large

Table 6.9: Best Configuration for Each Benchmark

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True								4-issue			
	32	False											
		True											
	64	False		4-issue					8-issue				
		True	8-issue										
	128	False					4-issue						
		True			2-issue	2-issue					2-issue	2-issue	

Source: The Author

Table 6.10: Duplication Ratio - 2-issue Processor

	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
SD-PG	5%	16%	21%	13%	4%	2%	17%	22%	16%	29%	12%
SD-PG-ILPc	1%	0%	0%	1%	0%	0%	0%	0%	6%	18%	0%
TD-PG(False)-Buffer(16)	42%	61%	100%	75%	9%	86%	67%	70%	94%	76%	100%
TD-PG(True)-Buffer(16)	42%	61%	100%	75%	9%	86%	67%	70%	94%	76%	100%
TD-PG(False)-Buffer(32)	57%	82%	100%	90%	10%	92%	75%	78%	98%	100%	100%
TD-PG(True)-Buffer(32)	57%	82%	100%	90%	10%	92%	75%	78%	98%	100%	100%
TD-PG(False)-Buffer(64)	87%	100%	100%	98%	10%	99%	86%	94%	99%	100%	100%
TD-PG(True)-Buffer(64)	87%	100%	100%	98%	10%	99%	86%	94%	99%	100%	100%
TD-PG(False)-Buffer(128)	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
TD-PG(True)-Buffer(128)	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%

Source: The Author

buffer for most benchmarks in order to duplicate all, or almost all instructions. Also, the buffer of 128 is large enough to provide duplication for all instructions in these benchmarks. When compared to the spatial duplication, the temporal mechanism is able to duplicate much more instructions as it exploits the empty slots in different time frames and while the processor is waiting for the memory.

The 4-issue core (Table 6.11), presents elevated duplication ratio with an intermediate buffer size, while the 8-issue (Table 6.12) is able to duplicate most instructions with small buffers, due to the increased amount of NOPs in the code generated by the compiler and because it is able to execute eight instructions in each cycle on a cache miss (compared to only two instructions per cycle in the 2-issue). Note that even for applications that are able to duplicate 100% of the instructions, the checker still remains vulnerable to faults and such vulnerability is considered in the reliability evaluation.

Table 6.11: Duplication Ratio - 4-issue Processor

	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
SD-PG	14%	39%	23%	78%	48%	10%	28%	75%	50%	76%	12%
SD-PG-ILPc	7%	8%	5%	26%	2%	0%	12%	5%	21%	63%	1%
TD-PG(False)-Buffer(16)	46%	100%	100%	100%	100%	50%	66%	100%	97%	100%	100%
TD-PG(True)-Buffer(16)	46%	94%	100%	100%	76%	31%	66%	100%	97%	100%	100%
TD-PG(False)-Buffer(32)	64%	100%	100%	100%	100%	100%	70%	100%	99%	100%	100%
TD-PG(True)-Buffer(32)	64%	100%	100%	100%	93%	90%	72%	100%	99%	100%	100%
TD-PG(False)-Buffer(64)	97%	100%	100%	100%	100%	100%	76%	100%	100%	100%	100%
TD-PG(True)-Buffer(64)	97%	100%	100%	100%	100%	97%	87%	100%	100%	100%	100%
TD-PG(False)-Buffer(128)	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
TD-PG(True)-Buffer(128)	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%

Source: The Author

Table 6.12: Duplication Ratio - 8-issue Processor

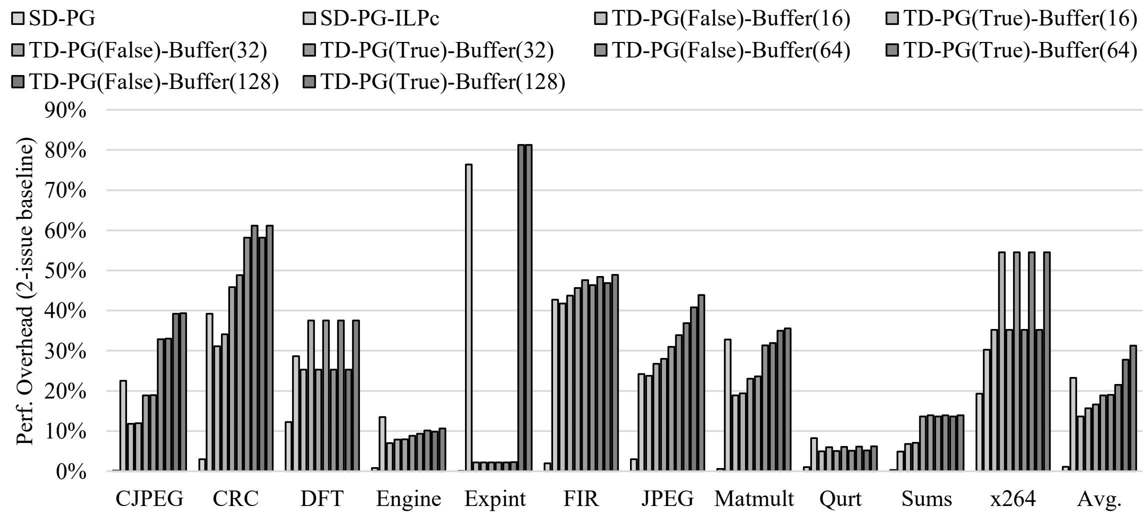
	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
SD-PG	44%	59%	18%	19%	4%	29%	23%	37%	44%	64%	1%
SD-PG-ILPc	15%	3%	7%	0%	2%	11%	2%	0%	30%	64%	1%
TD-PG(False)-Buffer(16)	85%	100%	100%	100%	98%	100%	84%	100%	97%	100%	100%
TD-PG(True)-Buffer(16)	85%	100%	100%	100%	98%	91%	84%	100%	97%	100%	100%
TD-PG(False)-Buffer(32)	99%	100%	100%	100%	100%	100%	92%	100%	99%	100%	100%
TD-PG(True)-Buffer(32)	99%	100%	100%	100%	100%	100%	92%	100%	99%	100%	100%
TD-PG(False)-Buffer(64)	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
TD-PG(True)-Buffer(64)	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
TD-PG(False)-Buffer(128)	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
TD-PG(True)-Buffer(128)	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%

Source: The Author

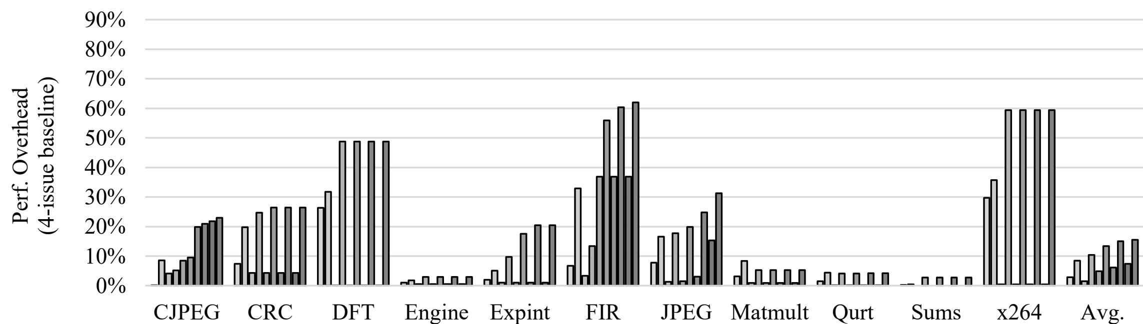
6.4.1.3 Performance

Figure 6.7 presents the performance overhead for each configuration when running the benchmarks in each issue-width. The average performance overhead for the spatial duplication with PG is of 1.10%, 2.89%, and 3.10% for the 2, 4, and 8-issue respectively. When the ILP control is applied to this technique, the overhead is increased to 23.25%, 8.43%, and 5.76%. When the Temporal duplication is added without applying PG and with a buffer size of 128, the average overhead is of: 27.80%, 7.39%, and 1.25% for the 2, 4, and 8-issue. Finally, when the PG is turned on, the overhead goes to 31.29%, 15.47%, and 12.78%. Considering only the spatial duplication, the PG results in more overhead in the 8-issue configuration because it has more opportunity to apply the power gating, which means that it will be applied more frequently and consequently it will incur in more overhead when the required FU needs to be turned on. On the other hand, the 2-issue is only able to apply PG to a reduced number of phases, because the pipelanes are

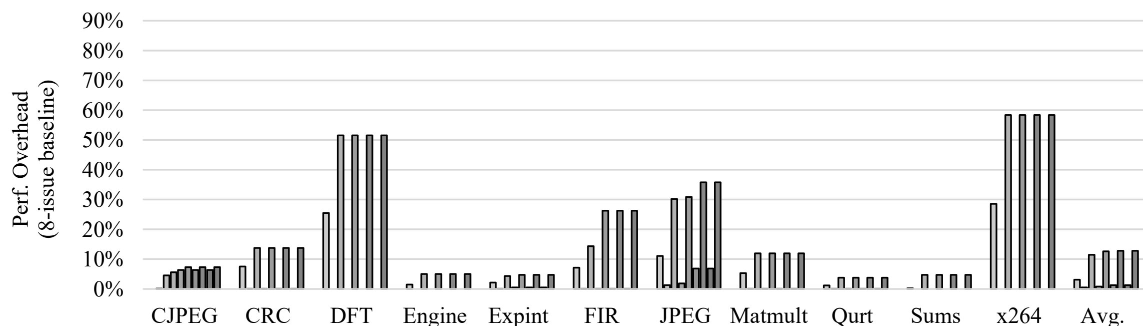
Figure 6.7: Performance Overhead Comparison



(a) 2-issue core



(b) 4-issue core



(c) 8-issue core

Source: The Author

occupied with program instructions most of the time.

For the temporal duplication, when the PG is applied, the buffers are stressed due to the reduction of the empty slots (slots that were going to be used to execute duplicated instructions are now turned off). In addition, there is also the overhead of the PG mechanism. When the temporal duplication is applied, the 2-issue results in a higher performance overhead than the 4 and 8-issue because there are fewer empty slots for the execution of the duplicated instructions and because it is able to execute only two instruc-

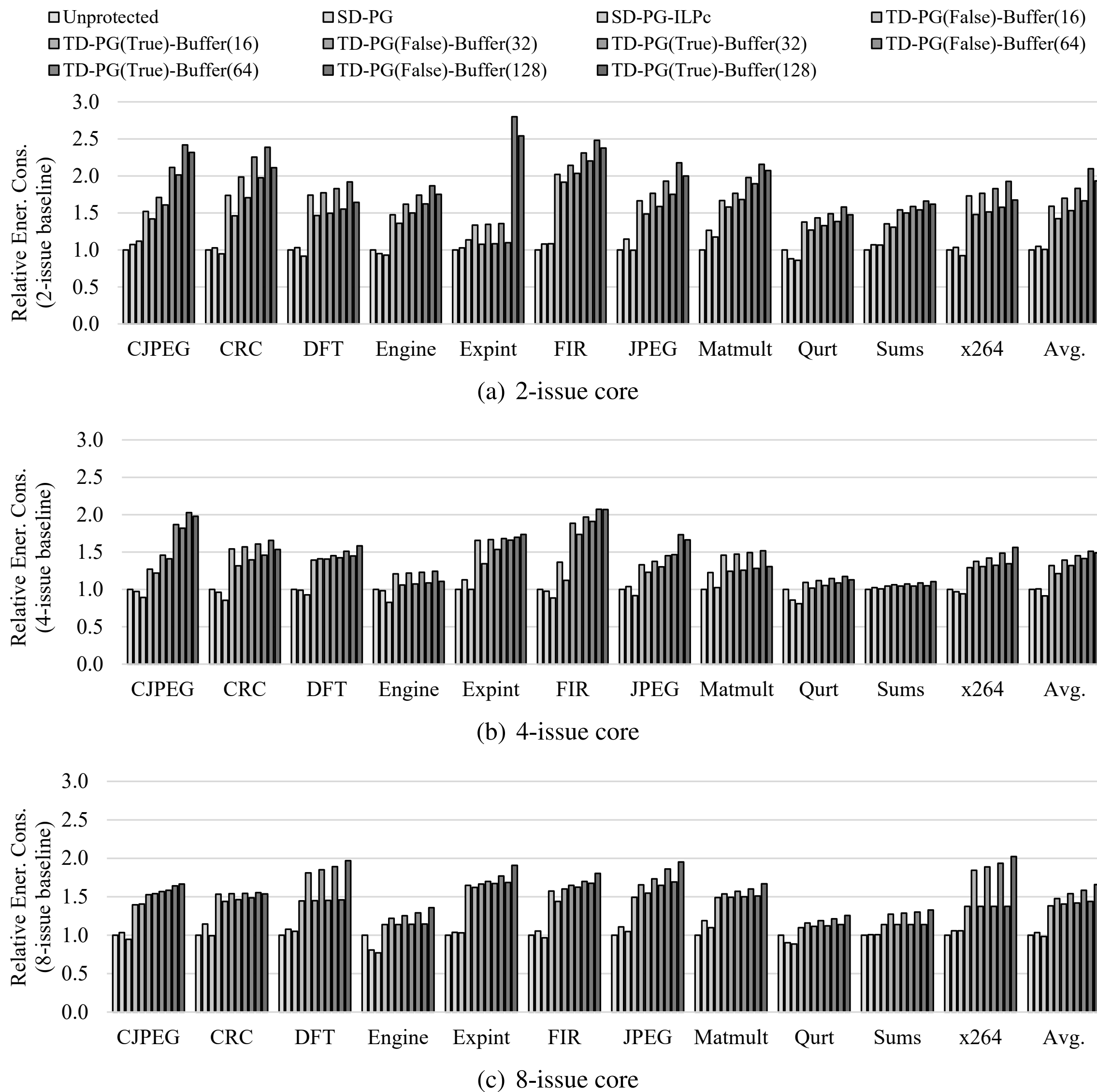
tions per cycle when a cache miss occurs, as aforementioned. The maximum overhead is of 81% for the *Expint* benchmark with a buffer size of 128, note that this benchmark was able to duplicate only 9-10% of the instructions with a small buffer size, but when the buffer is increased to 128, it is able to duplicate 100% of the instructions (resulting in more performance overhead).

6.4.1.4 Energy and Area

Figure 6.8 depicts the relative energy consumption when applying a protection technique to each issue-width. As the temporal duplication increases considerably the number of instructions that are duplicated, the energy consumption is also increased because the modules will have more switching activity, thus more power dissipation. As the energy consumption is also relative to the execution time, the 2-issue core presents a higher overhead in energy due to its reduced performance. The maximum energy overhead appears in the *Expint* benchmark with 128 buffer and without PG, being of 2.8 times. This overhead is due to the execution of more (duplicated) instructions, reduction in performance, and the additional power dissipation of the modules, buffers, and memories from the temporal duplication mechanism. On average, the temporal duplication increases the energy consumption by 65.04%, 30.17%, and 43.30% for the 2, 4, and 8-issue. The 4-issue presents the lowest energy overhead because the 2-issue has high performance overhead, while the 8-issue has elevated power dissipation, both influencing the energy consumption.

Table 6.13 presents the area of each configuration compared to the unprotected processor that includes cache memories. Even though these results are for the polymorphic processor, in this table, the results are presented for each static configuration in order to allow the comparison of different issue-widths. Increasing the buffer size naturally increases the area, so as applying PG. The cache configuration is the same from the polymorphic core, that is, the 8-issue has a 16KB cache, the 4-issue 8KB, and the 2-issue 4KB. The maximum overhead is of 13.83% on the 8-issue temporal duplication with PG and a buffer size of 128, and it can be as low as 1.56% on the 2-issue with a buffer size of 16, without PG. The area for the whole polymorphic processor, including the configuration memory and the polymorphic buffers will be discussed in the next subsection.

Figure 6.8: Relative Energy Consumption Comparison



(a) 2-issue core

(b) 4-issue core

(c) 8-issue core

Source: The Author

6.4.2 Optimization Algorithm

In this subsection, first, the number of steps required by the polymorphic processor to find the best configuration for each application will be evaluated. Then, its ability to cope with different applications' behavior will be evaluated through the MWPUETF metric. Finally, the performance and energy to execute all applications will be compared.

6.4.2.1 Number of Steps to Find the Best Configuration

Considering that we are evaluating a total of 27 possible configurations (unprotected: 3 issue-widths, and temporal duplication: 4 buffer sizes, 3 issue-widths, and enable or disable the PG), the maximum number of steps that the optimization algorithm

Table 6.13: Area Comparison - Temporal Duplication Mechanism

	Area (mm^2)			Area Overhead		
	2-issue	4-issue	8-issue	2-issue	4-issue	8-issue
Unprotected	0.530	0.686	1.038	0.00%	0.00%	0.00%
SD-PG	0.548	0.721	1.097	3.37%	5.00%	5.68%
SD-PG-ILPc	0.548	0.721	1.097	3.37%	5.00%	5.68%
TD-PG(False)-Buffer(16)	0.539	0.703	1.086	1.56%	2.38%	4.63%
TD-PG(True)-Buffer(16)	0.549	0.722	1.119	3.41%	5.13%	7.76%
TD-PG(False)-Buffer(32)	0.542	0.708	1.096	2.20%	3.13%	5.60%
TD-PG(True)-Buffer(32)	0.552	0.727	1.129	4.04%	5.88%	8.73%
TD-PG(False)-Buffer(64)	0.549	0.719	1.112	3.41%	4.79%	7.14%
TD-PG(True)-Buffer(64)	0.558	0.738	1.145	5.26%	7.54%	10.27%
TD-PG(False)-Buffer(128)	0.562	0.737	1.149	5.90%	7.38%	10.70%
TD-PG(True)-Buffer(128)	0.572	0.756	1.182	7.75%	10.13%	13.83%

Source: The Author

Table 6.14: Number of Kernel Executions and Number of Steps to Find the Best Configuration

	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Num. of steps	19	19	23	19	21	17	19	19	19	19	23
Num. of iterations	215500	400	44440	200	40000	600	200	250	2000	138500	1000
Tests-Iterations ratio	0.01%	4.75%	0.05%	9.50%	0.05%	2.83%	9.50%	7.60%	0.95%	0.01%	2.30%

Source: The Author

will perform is 27. Table 6.14 presents the number of executions for each kernel (as previously discussed in Section 6.1.1), and the number of steps required to find the best configuration for each application. For these benchmarks, the number of tests varies from 17 to 23. Thus, the optimization algorithm is able to find the best configuration without evaluating all possible scenarios. The average number of tests is 19.65.

This table also shows the ratio between the number of tests and iterations, which vary from 0.01% to 9.5%, the latter occurs in the *Engine* and *JPEG* applications: 19 out of 200 kernel executions are performed by the learning phase, then, the other 181 iterations are executed in the best configuration. Each execution is already an iteration of the kernel and the learning phase contributes to the final result of the application, a given iteration is not executed twice after finding the best configuration. There are two main behaviors regarding the performance of the tests, for instance, if a given benchmark has the 4-issue as the best issue-width, it will start testing on the 2-issue mode, which will incur in a small performance overhead in the first 8 iterations (which is a small part of the application). The opposite can also occur, if the application has the 2-issue as the best configuration,

the algorithm will also test some configurations of the 4-issue to make sure that it found the best configuration for that benchmark. In this case, the benchmark will improve its performance while the 4-issue testing is being performed. In addition, all overheads for the configuration switching are taken into account as discussed in Section 6.1, which are lower than 0.003% in terms of performance for all benchmarks.

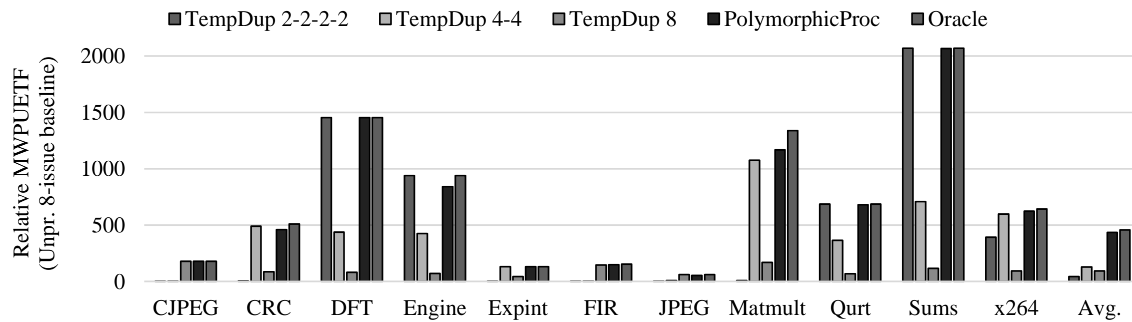
6.4.2.2 MWPUETF Comparison with the Same Weight for All Axes

In order to evaluate the proposed polymorphic processor, we compared it to the unprotected versions and to the best static configuration *on average* from the configurations evaluated in Section 6.4.1, for each issue-width. In addition, each static configuration is able to execute multiple applications concurrently (four 2-issue applications, two 4-issue, or one 8-issue). An Oracle processor is used for comparison and each of these configurations is detailed next.

- ***Unprotected 2-2-2-2*** : Unprotected dynamic processor with four contexts, each one executing one application in 2-issue mode.
- ***Unprotected 4-4*** : Unprotected processor with two contexts, 4-issue each.
- ***Unprotected 8*** : Unprotected processor with a single 8-issue context.
- ***Temporal Duplication 2-2-2-2*** : On average, the best 2-issue configuration is the 128 buffer with PG. Also, four applications can be executed in parallel in this configuration.
- ***Temporal Duplication 4-4*** : The best configuration for the 4-issue is also the 128 buffer with PG and it runs two contexts.
- ***Temporal Duplication 8*** : The 8-issue has a 64 buffer without PG as the best configuration, running a single context.
- ***Polymorphic Processor*** : The polymorphic processor is able to switch between different issue-widths and configurations considering fault tolerance, energy optimization, and performance. Each benchmark is dynamically tested and the best configuration is found according to the aforementioned optimization algorithm (Algorithm 1).
- ***Oracle Processor*** : The oracle processor executes all applications in the best possible configuration for each one, providing an upper bound for comparison.

Figure 6.9 presents the MWPUETF of each benchmark when executed with the best (average) configuration, and it is compared to the proposed polymorphic processor and the oracle processor. The results are normalized to the ***Unprotected 8*** and they

Figure 6.9: MWPUETF Comparison - Polymorphic Processor



Source: The Author

are presented for each application individually, the performance and energy consumption when the applications are scheduled and executed in parallel will be evaluated in the next subsection.

When comparing only the static versions, each configuration is the most appropriate for a subset of applications. On average, the ***Temporal Duplication 4-4*** achieves an improvement of 129 times when compared to the unprotected version. However, this value is still low when compared to the ***Oracle Processor***, that is able to achieve a factor of 458 times, on average. Therefore, there is not one specific configuration or issue-width that is able to deliver the best result when the axes of fault tolerance, energy consumption, and performance are considered, supporting the motivation that a dynamic mechanism is required to provide a processor that is able to efficiently adapt itself according to the application at hand.

The results for the ***Polymorphic Processor*** in Figure 6.9 depict that it is able to choose the best configuration dynamically after a brief period of learning and it is able to adapt the processor to cope with all applications' behavior, delivering a MWPUETF improvement close to the oracle. Table 6.15 presents the results normalized to the ***Oracle Processor***. The ***Polymorphic Processor*** is able to get from 86.01% to 99.99% of the oracle's result, having an average of 94.88%. This means that the ***Polymorphic Processor*** is able to deliver almost the same result as the ***Oracle Processor***, but in a completely dynamic manner. On the other hand, the best static configuration (***Temporal Duplication 4-4***) is only able to achieve 28.24% of the result from the ***Oracle Processor***. In addition, if one decides to restrict the tests to configurations that are most likely to result in higher improvements, for instance, by eliminating the test of the unprotected configurations, the ***Polymorphic Processor*** can get even closer to the ***Oracle Processor***, as fewer tests will be performed to find the best configuration.

Table 6.15: MWPUETF Normalized to the Oracle Processor

	Temporal duplication			Polymorphic
	2-2-2-2	4-4	8	
CJPEG	0.71%	1.68%	99.14%	99.99%
CRC	0.80%	96.40%	16.79%	90.36%
DFT	100.00%	30.06%	5.61%	99.95%
Engine	100.00%	45.12%	7.54%	89.81%
Expint	2.45%	99.65%	33.69%	99.90%
FIR	1.27%	2.19%	95.57%	96.24%
JPEG	6.36%	14.06%	98.89%	86.01%
Matmult	0.59%	80.28%	12.67%	87.21%
Qurt	100.00%	52.98%	10.00%	99.10%
Sums	100.00%	34.25%	5.63%	99.98%
x264	61.02%	92.85%	14.56%	96.78%
Average	9.21%	28.24%	20.24%	94.88%

Source: The Author

Finally, the ***Polymorphic Processor*** was able to find the best configuration for all benchmarks, running most of the application in such configuration (Table 6.9). The small decrease in the final MWPUETF is due to the learning phase, which executes the application in a sub-optimal configuration, but still delivers a result that is close to the ***Oracle Processor*** in all benchmarks. On average, the ***Polymorphic Processor*** is able to improve the MWPUETF by 434 times, when compared to the ***Unprotected 8***. All modules for the polymorphic processor, including the fault tolerance and PG mechanisms, result in an area overhead of 26.82% when compared to the unprotected processor.

6.4.2.3 MWPUETF Comparison with Different Weights for Each Axis

In this subsection, we evaluate the MWPUETF metric with *different* weights for each axis, prioritizing a specific axis. The complete comparison was included to Appendix B, and here we will focus on the most significant configuration for each axis, which was using a factor of 0.8 to each of the axes. Therefore, the a, b, and c from Equation (4.4) are set to all combinations of 0.8, 0.1, and 0.1, prioritizing one axis at a time.

The static configurations are chosen in the same way as the ones from Section 6.4.2.2. That is, the unprotected versions are evaluated with 2, 4, and 8 issue-slots, and the protected configurations are based on the best configuration *on average* for all benchmarks. Table 6.16 depicts the best static configuration for each issue-width when prioritizing each of the axes.

Table 6.16: Best Static Configuration When Prioritizing Each of the Axes

	Temporal Duplication					
	2-2-2-2		4-4		8	
Priority	Buffer	PG	Buffer	PG	Buffer	PG
Energy Consumption	128	True	32	True	64	False
Performance	128	True	128	False	64	False
Fault Tolerance	128	True	128	False	128	False

Source: The Author

Table 6.17 presents the best configuration for each benchmark when the energy consumption axis is prioritized. As we increase the weight of energy consumption, the 2 and 4-issue are prioritized, as well as smaller buffer sizes and low overhead configurations, because they consume less energy. Table 6.18 presents the results from the static and polymorphic configurations when normalized to the ***Oracle Processor***. As most benchmarks tend to use similar configurations when the energy is prioritized (i.e., small issue-width), the static versions are able to get closer to the oracle. The ***Temporal Duplication 2-2-2-2*** achieves 85.95% of the oracle's result on average, the 4-issue 90.77%, and the 8-issue 59.82%. The best unprotected configuration achieves 84.33% (***Unprotected 2-2-2-2***). Nonetheless, the ***Polymorphic Processor*** still is able to provide better adaptation to the changing applications, achieving 95.35%.

Table 6.17: Best Configuration - Priority: Energy (0.8)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False	2-issue				2-issue	2-issue	2-issue				
TempDup	16	False											4-issue
		True				4-issue				4-issue			
	32	False		4-issue									
		True											
	64	False											
		True									4-issue		
	128	False											
		True			2-issue							2-issue	

Source: The Author

Table 6.18: MWPUETF Normalized to the Oracle Processor - Priority: Energy Consumption

	Unprotected			Temporal duplication			Polymorphic
	2-2-2-2	4-4	8	2-2-2-2	4-4	8	
CJPEG	100.00%	92.65%	69.20%	71.51%	87.38%	88.49%	99.99%
CRC	96.27%	79.90%	54.89%	89.40%	87.85%	62.91%	92.63%
DFT	68.84%	59.04%	41.13%	100.00%	90.75%	48.50%	99.96%
Engine	73.46%	58.14%	37.54%	91.11%	99.13%	52.16%	87.74%
Expint	100.00%	79.15%	54.22%	94.08%	77.50%	55.33%	99.96%
FIR	100.00%	94.13%	66.73%	69.83%	94.30%	77.46%	96.32%
JPEG	100.00%	89.47%	64.09%	83.28%	92.79%	69.25%	89.46%
Matmult	76.32%	61.26%	40.26%	60.36%	99.03%	50.48%	87.88%
Qurt	70.96%	58.87%	40.64%	98.80%	95.44%	56.75%	98.58%
Sums	68.72%	49.96%	35.51%	100.00%	89.95%	51.93%	99.99%
x264	84.64%	69.08%	45.66%	99.53%	86.55%	56.43%	97.68%
Average	84.33%	70.41%	48.64%	85.95%	90.77%	59.82%	95.35%

Source: The Author

Table 6.19 depicts the best configurations when prioritizing performance. As we increase the weight of performance, the 4 and 8-issue are prioritized, as they deliver better performance than the 2-issue. The 8-issue does not greatly improve the performance when compared to the 4-issue, so even when prioritizing the axis of performance, it does not outweigh the extra energy consumption of the 8-issue core for most benchmarks. Table 6.20 shows the results normalized to the ***Oracle Processor***. From the static configurations, the ***Temporal Duplication 8*** achieves 87.85%, followed by the 4-issue with 87.56%, and the 2-issue with 69.52%. The same reasoning of the energy prioritization can be applied to the performance one, it narrows the set of possible configurations for the static processor as benchmarks tend to use higher issue-widths when more performance is desired. Thus, being able to achieve better results when compared to the scenario in which all axes have the same weight. The ***Polymorphic Processor*** gets to 96.97% compared to the oracle.

Finally, Table 6.21 shows the configurations with focus on the fault tolerance axis. Prioritizing the fault tolerance axis does not have a clear trend when its weight is increased because such behavior highly depends on the application behavior and the number of instructions that each configuration is able to duplicate. That is, applications with low ILP can duplicate most instructions with a small issue-width, while applications with high ILP require larger issue-widths and buffer sizes. Table 6.22 presents the results when normalized to the ***Oracle Processor***, prioritizing fault tolerance. The ***Temporal Duplication 8***, on average, gets to 35.45% of the oracle's result, the 4-issue 34.96%, and the 2-issue 14.06%. As expected, the results for the unprotected versions when prioritizing fault tol-

Table 6.19: Best Configuration - Priority: Performance (0.8)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True			4-issue					4-issue			
	32	False											
		True											
	64	False		4-issue					8-issue				
		True	8-issue								4-issue		
	128	False					4-issue						
		True				2-issue						2-issue	

Source: The Author

Table 6.20: MWPUETF Normalized to the Oracle Processor - Priority: Performance

	Unprotected			Temporal duplication			Polymorphic
	2-2-2-2	4-4	8	2-2-2-2	4-4	8	
CJPEG	36.39%	51.77%	59.50%	35.86%	56.66%	99.91%	99.99%
CRC	52.20%	57.72%	53.69%	54.68%	99.63%	83.76%	93.72%
DFT	51.47%	56.12%	56.09%	99.34%	97.84%	87.11%	99.96%
Engine	56.08%	56.03%	52.91%	100.00%	96.02%	80.99%	94.65%
Expint	68.38%	65.73%	62.57%	68.78%	99.96%	91.37%	99.94%
FIR	46.91%	60.55%	60.44%	48.79%	65.16%	99.55%	97.23%
JPEG	54.84%	66.73%	66.31%	60.29%	77.78%	99.89%	91.67%
Matmult	52.47%	53.21%	48.69%	57.50%	97.83%	81.35%	92.84%
Qurt	54.87%	57.93%	56.72%	97.28%	98.93%	86.57%	99.04%
Sums	50.85%	48.98%	47.93%	100.00%	89.84%	77.13%	99.99%
x264	52.70%	57.77%	52.41%	83.21%	99.26%	82.53%	98.16%
Average	51.95%	57.28%	55.85%	69.52%	87.56%	87.85%	96.97%

Source: The Author

erance are extremely low (it does not have any fault tolerance mechanism), being 0.99% at most on the ***Unprotected 2-2-2-2***. As in all the other comparisons, the ***Polymorphic Processor*** is able to adapt the processor configuration to the application at hand, both when all axes have the same weight or when a given axis is prioritized, and it achieves 95.37% when the priority is fault tolerance.

Table 6.21: Best Configuration - Priority: Fault tolerance (0.8)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True								4-issue			
	32	False											
		True											
	64	False	8-issue	4-issue					8-issue				
		True											
	128	False					4-issue						
		True			2-issue	2-issue					2-issue	2-issue	

Source: The Author

Table 6.22: MWPUETF Normalized to the Oracle Processor - Priority: Fault Tolerance

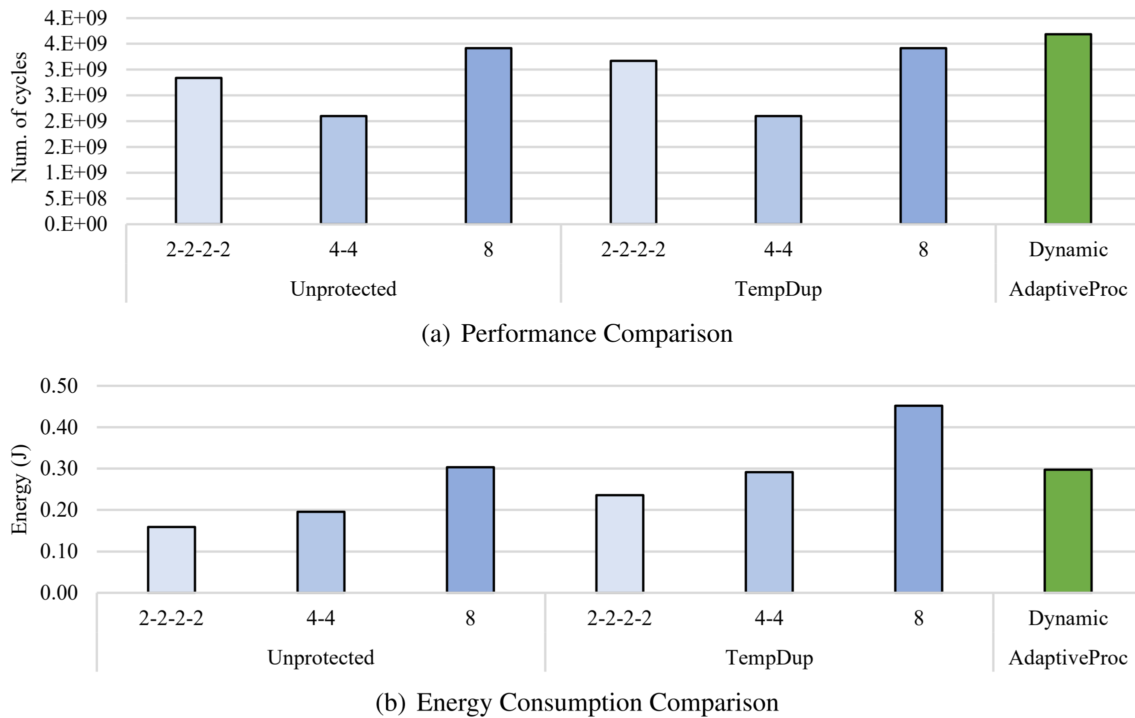
	Unprotected			Temporal duplication			Polymorphic
	2-2-2-2	4-4	8	2-2-2-2	4-4	8	
CJPEG	1.21%	1.13%	1.20%	2.46%	3.58%	99.46%	99.99%
CRC	0.98%	0.89%	0.67%	1.64%	99.63%	31.84%	92.28%
DFT	0.39%	0.35%	0.30%	100.00%	39.08%	13.19%	99.96%
Engine	0.68%	0.58%	0.49%	100.00%	50.15%	16.25%	90.67%
Expint	2.82%	2.42%	1.88%	3.16%	99.96%	62.44%	99.93%
FIR	1.47%	1.48%	1.29%	3.00%	3.50%	99.25%	95.71%
JPEG	3.14%	3.56%	2.70%	8.85%	16.09%	99.62%	85.42%
Matmult	0.59%	0.56%	0.38%	1.69%	97.83%	30.83%	89.93%
Qurt	0.80%	0.71%	0.61%	100.00%	57.70%	19.55%	99.33%
Sums	0.33%	0.32%	0.28%	100.00%	43.23%	14.06%	99.98%
x264	1.42%	1.04%	0.65%	73.68%	99.26%	31.27%	97.30%
Average	0.99%	0.91%	0.73%	14.06%	34.96%	35.45%	95.37%

Source: The Author

6.4.2.4 System Performance and Energy Consumption

Figure 6.10 presents the comparison in terms of performance and energy of a system that will schedule and execute the 11 applications that are being evaluated, for the unprotected, temporal duplication, and polymorphic designs. Considering the performance (Figure 6.10(a)) of the unprotected versions, the ***Unprotected 4-4*** is able to execute the 11 benchmarks faster than the 2- and 8-issue versions, this is because the 4-issue is able to execute two applications in parallel, while the performance improvement of the 8-issue does not compensate the fact that it can only execute a single context. The ***Unprotected 2-2-2-2*** is also faster than the 8-issue, even though each application takes more time to execute, four applications can be executed in parallel.

Figure 6.10: System Performance and Energy Consumption Comparison



(a) Performance Comparison

(b) Energy Consumption Comparison

Source: The Author

The temporal duplication designs follow the same trend as the unprotected ones, with an overhead for the temporal duplication and PG mechanisms, which is of 21% for the ***Temporal Duplication 2-2-2-2***. The ***Polymorphic Processor*** will execute each application in its best configuration (after the learning phase), so the issue-width is not fixed, and the dispatcher needs to fill the pipelanes with up to four applications in parallel. As discussed in Chapter 4, the dispatcher prioritizes the benchmarks with longer execution time to be scheduled first and then tries to schedule as many benchmarks in parallel as possible, in order to increase the flexibility of scheduling the remaining applications. The ***Polymorphic Processor*** presents a similar performance to the ***Temporal Duplication 8***, because it has to schedule applications with different issue-width and shut down the slots that cannot be used in a given instant of the execution.

Considering energy consumption (Figure 6.10(b)), the 8 and 4-issue version consume more energy due to the increased power dissipation, even though the performance of the 4-4 is better than the 2-2-2-2. The ***Polymorphic Processor*** is able to achieve 34% less energy consumption when compared to the ***Temporal Duplication 8***, 2% more than the ***Temporal Duplication 4-4*** and 26% more than the ***Temporal Duplication 2-2-2-2***.

Therefore, the proposed ***Polymorphic Processor*** is able to dynamically choose the most appropriate processor configuration for each application with low overhead, and it

achieves results close to the ***Oracle Processor*** when the trade-off among fault tolerance, energy consumption, and performance is considered.

7 CONCLUSION AND FUTURE WORK

7.1 Summary of Contributions

In this work, a polymorphic and adaptive processor was developed to trade-off the axes of fault tolerance, energy consumption, and performance, by implementing specific mechanisms for each of these axes. In addition, these mechanisms together with the processor configuration are tuned during runtime to deliver the best trade-off between these axes by a multi-objective optimization algorithm.

The fault tolerance and ILP control mechanisms are able to provide a significant reduction in the failure rate with modest hardware cost. The PG mechanism provides support for reducing the energy consumption of the processor, reducing the energy overhead that is created when executing duplicated instructions or reducing the base energy consumption when it is used to shut down idle hardware. In addition, simulators were developed to mimic and exploit the dynamic version of the ρ-VEX processor to allow the configuration of the processor to be changed during runtime, and the proposed mechanisms were also developed with dynamically reconfigurable features such as the buffer sizes and the ability to enable or disable the PG mechanism.

For the polymorphic processor, the multi-objective optimization algorithm dynamically tests the applications that are being executed and chooses the best configuration of the processor to maximize the MWPUETF, which is able to achieve results within 94.88% of the results of an oracle, on average. This demonstrates that the optimization algorithm can quickly and accurately select the optimal configuration for each benchmark and it allows the weight of each axis to be tuned during design time, so the polymorphic processor can choose, during runtime, the best configuration based on system requirements. In addition, the application dispatcher maximizes the number of applications executing in parallel to exploit the available hardware.

Finally, a hybrid fault injector was proposed to speed up the fault injection campaign by simulating the target design in two different modes: RTL and gate-level. Therefore, executing most of the application in the accelerated mode (RTL) and automatically switching to the gate-level simulation when the fault is going to be injected, so the accuracy of the fault injection is maintained.

7.2 Future Work

As future work, some possible extensions to the polymorphic processor are discussed next.

7.2.1 Additional Techniques for Energy Optimization

Additional techniques energy optimization may be integrated into the proposed polymorphic processor. For instance, DVFS can be evaluated as it is used in most of the current processors, but having in mind that it may affect reliability when operating in low voltage levels. In addition, PG can be applied to the register file as well, reducing the energy consumption when the application does not use all the registers that are available in the design.

7.2.2 Support for Multithreaded Applications

ρ-VEX support for OpenMP is currently being developed. Therefore, the proposed polymorphic processor can be extended to support multithreaded applications in addition to multiple (single-threaded) applications running concurrently. Possibly minor changes in the proposed design would be required for such extension, as they can be applied to individual threads in the same way they are applied to different applications. Therefore, each thread can be evaluated individually and the best configuration for each thread can be applied.

7.2.3 Criticality-aware Duplication

Instructions have different criticality and different probabilities of being affected by a energetic particle. For example, a memory operation that is affect by a energy particle will likely result in wrong computation as the data or the address of the operation will be incorrect. On the other hand, an *AND* operation with one input in *'0'* will result in the same result regardless of the other (possibly faulty) input. In addition, the probability of an instruction being affected by an energetic particle also depends on the hardware area that such instruction requires. For instance, a multiplication operation will likely be more

affected than an *ADD* (as the multiplier occupies more area than the adder). Therefore, the proposed processor can be extended to consider these different probabilities and criticality to prioritize the duplication of critical instructions.

7.2.4 ILP Control for the Polymorphic Processor

As previously discussed, the ILP control used in the adaptive version (which was implemented in the static version of the ρ-VEX) was not used in the polymorphic version because the temporal duplication provides the flexibility of duplicating instructions in different clock cycles, so if the same ILP control mechanism was to be implemented, additional performance overhead would be created while those instructions could be duplicated in a cache miss, for instance. Therefore, a more complex ILP control mechanism is required, which considers the temporal duplication behavior to decide when to reduce the ILP of the application.

7.2.5 Improved Application Dispatcher

In this work, a simple mechanism was implemented to schedule the applications, sorting them by execution time in order to increase the flexibility of the scheduling. However, more complex approaches can be evaluated in order to further reduce the total execution time and perform a scheduling closer to the optimal.

7.3 Publications

The publications that were achieved by the author during the thesis period are listed next.

7.3.1 Journals

- Dynamic Trade-off among Fault Tolerance, Energy Consumption, and Performance on a Multiple-issue VLIW Processor, **IEEE Transactions on Multi-Scale Computing Systems**, (SARTOR et al., 2017).

- Exploiting Idle Hardware to Provide Low Overhead Fault Tolerance for VLIW Processors, **ACM Journal on Emerging Technologies in Computing Systems (JETC)**, (SARTOR et al., 2017).
- Multi-architecture profiler for Android, **International Journal of High Performance Systems Architecture (IJHPSA)**, (SARTOR; BECK, 2017).

7.3.2 Conferences and Workshops

- Adaptive and Polymorphic VLIW Processor to Optimize Fault Tolerance, Energy Consumption, and Performance, **ACM International Conference on Computing Frontiers**, (SARTOR et al., 2018)
- ISA-DTMR: Selective Protection in Configurable Heterogeneous Multicores, **International Symposium on Applied Reconfigurable Computing (ARC)**, (ERICHSEN et al., 2018).
- A Low-Cost BRAM-based Function Reuse for Configurable Soft-Core Processors in FPGAs, **International Symposium on Applied Reconfigurable Computing (ARC)**, (BECKER et al., 2018).
- DIM-VEX: Exploiting Design Time Configurability and Runtime Reconfigurability, **International Symposium on Applied Reconfigurable Computing (ARC)**, (SOUZA et al., 2018).
- Simbah-FI: Simulation-Based Hybrid Fault Injector, **Brazilian Symposium on Computing Systems Engineering (SBESC)**, (SARTOR; BECKER; BECK, 2017).
- Adaptive ILP control to increase fault tolerance for VLIW processors, **International Conference on Application-specific Systems, Architectures and Processors (ASAP)**, (SARTOR; WONG; BECK, 2016).
- Run-time Phase Prediction for a Reconfigurable VLIW Processor, **Design, Automation and Test in Europe Conference and Exhibition (DATE)**, (GUO et al., 2016)
- How Programming Languages and Paradigms Affect Performance and Energy in Multithreaded Applications, **Brazilian Symposium on Computing Systems Engineering (SBESC)**, (MAGALHAES et al., 2016).
- A Novel Phase-Based Low Overhead Fault Tolerance Approach for VLIW Processors, **IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**, (SAR-

TOR et al., 2015).

- The Impact of Virtual Machines on Embedded Systems, **IEEE Computer Software and Applications Conference (COMPSAC)**, (SARTOR; LORENZON; BECK, 2015).
- Evaluation of energy savings on a VLIW processor through dynamic issue-width adaptation, **International Symposium on Rapid System Prototyping (RSP)**, (GIRALDO et al., 2015).
- Evaluation of Failures Masking Across the Software Stack, **MEDIAN Workshop**, (SANTINI et al., 2015).
- The Influence of Parallel Programming Interfaces on Multicore Embedded Systems, **IEEE Computer Software and Applications Conference (COMPSAC)**, (LORENZON et al., 2015b).
- Optimized Use of Parallel Programming Interfaces in Multithreaded Embedded Architectures, **IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**, (LORENZON et al., 2015a).
- A sparse VLIW instruction encoding scheme compatible with generic binaries, **International Conference on ReConFigurable Computing and FPGAs (ReConFig)**, (BRANDON et al., 2015).
- A Transparent Multiple-ISA MPSoC Architecture, **Workshop on SoCs, Heterogeneous Architectures and Workloads**, (SARTOR et al., 2014).

REFERENCES

ADITYA, S.; MAHLKE, S. A.; RAU, B. R. Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, ACM, v. 5, n. 4, p. 752–773, 2000.

ANJAM, F.; NADEEM, M.; WONG, S. Targeting code diversity with run-time adjustable issue-slots in a chip multiprocessor. In: IEEE. **Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2011. p. 1–6.

ANJAM, F.; WONG, S. Configurable fault-tolerance for a configurable VLIW processor. In: **Reconfigurable Computing: Architectures, Tools and Applications**. [S.l.]: Springer, 2013. p. 167–178.

ASSAYAD, I.; GIRAULT, A.; KALLA, H. Tradeoff exploration between reliability, power consumption, and execution time. In: **Computer Safety, Reliability, and Security**. [S.l.]: Springer, 2011. p. 437–451.

AUSTIN, T. M. DIVA: A reliable substrate for deep submicron microarchitecture design. In: IEEE. **Microarchitecture. Annual International Symposium on**. [S.l.], 1999. p. 196–207.

BECKER, P. H. E. et al. A Low-Cost BRAM-based Function Reuse for Configurable Soft-Core Processors in FPGAs. In: **International Symposium on Applied Reconfigurable Computing (ARC)**. [S.l.: s.n.], 2018.

BERG, M. et al. Effectiveness of internal vs. external SEU scrubbing mitigation strategies in a Xilinx FPGA: Design, test, and analysis. 2008.

BHATT, S.; CHEN, M.; LIN, C. Y.; LIU, P. Abstractions for parallel N-body simulations. In: **Proceedings Scalable High Performance Computing Conference SHPCC.** [S.l.: s.n.], 1992. p. 38–45.

BINKERT, N. et al. The Gem5 Simulator. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 39, n. 2, p. 1–7, 2011. ISSN 0163-5964.

BINKERT, N. L. et al. The M5 Simulator: Modeling Networked Systems. **IEEE Micro**, v. 26, n. 4, p. 52–60, 2006. ISSN 0272-1732.

BOLCHINI, C. A software methodology for detecting hardware faults in VLIW data paths. **Reliability, IEEE Transactions on**, IEEE, v. 52, n. 4, p. 458–468, 2003.

BOLCHINI, C.; MIELE, A.; SANDIONIGI, C. Increasing autonomous fault-tolerant FPGA-based systems' lifetime. In: IEEE. **IEEE European Test Symposium (ETS)**. [S.l.], 2012. p. 1–6.

BOLCHINI, C.; SANDIONIGI, C. Fault classification for SRAM-Based FPGAs in the space environment for fault mitigation. **IEEE Embedded Systems Letters**, IEEE, v. 2, n. 4, p. 107–110, 2010.

BORKAR, S. Design challenges of technology scaling. **IEEE micro**, IEEE, v. 19, n. 4, p. 23–29, 1999.

BORNSTEIN, B.; ESTLIN, T.; CLEMENT, B.; SPRINGER, P. Using a multicore processor for rover autonomous science. In: **Aerospace Conference**. [S.l.: s.n.], 2011. p. 1–9. ISSN 1095-323X.

BOYER, M.; TARJAN, D.; SKADRON, K. Federation: Boosting per-thread performance of throughput-oriented manycore architectures. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM, v. 7, n. 4, p. 19, 2010.

BRANDON, A. et al. A Sparse VLIW Instruction Encoding Scheme Compatible with Generic Binaries. In: **Reconfigurable Computing and FPGAs (ReConFig), International Conference on**. [S.l.: s.n.], 2015.

BRANDON, A.; HOOZEMANS, J.; STRATEN, J. van; WONG, S. Exploring ILP and TLP on a Polymorphic VLIW Processor. In: ____. **Architecture of Computing Systems (ARCS), International Conference**. [S.l.]: Springer International Publishing, 2017. p. 177–189. ISBN 978-3-319-54999-6.

BRANDON, A.; WONG, S. Support for dynamic issue width in VLIW processors using generic binaries. In: EDA CONSORTIUM. **Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2013. p. 827–832.

BRÜNINK, M. et al. Aaron: An adaptable execution environment. In: IEEE. **Dependable Systems & Networks (DSN), IEEE/IFIP International Conference on**. [S.l.], 2011. p. 411–421.

BUTENHOF, D. R. **Programming with POSIX threads**. [S.l.]: Addison-Wesley Professional, 1997.

CARMICHAEL, C.; CAFFREY, M.; SALAZAR, A. Correcting single-event upsets through Virtex partial configuration. **Xilinx Application Note, XAPP216 (v1. 0)**, 2000.

CHANDRAMOWLISHWARAN, A.; KNOBE, K.; VUDUC, R. Performance evaluation of concurrent collections on high-performance multicore computing systems. In: IEEE. **Parallel & Distributed Processing (IPDPS), IEEE International Symposium on**. [S.l.], 2010. p. 1–12.

CHAPMAN, B.; JOST, G.; Van Der Pas, R. **Using OpenMP: portable shared memory parallel programming**. [S.l.]: MIT press, 2008.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: **IEEE International Symposium on Workload Characterization (IISWC)**. [S.l.: s.n.], 2009. p. 44–54.

CHEN, Y.-Y.; LEU, K.-L. Reliable data path design of VLIW processor cores with comprehensive error-coverage assessment. **Microprocessors and Microsystems**, Elsevier, v. 34, n. 1, p. 49–61, 2010.

CHO, H.; CHER, C.-Y.; SHEPHERD, T.; MITRA, S. Understanding Soft Errors in Uncore Components. In: **Proceedings of the Annual Design Automation Conference (DAC)**. New York, NY, USA: ACM, 2015. (DAC), p. 89:1—-89:6. ISBN 978-1-4503-3520-1.

CHO, H.; MIRKHANI, S.; CHER, C. Y.; ABRAHAM, J. A.; MITRA, S. Quantitative evaluation of soft error injection techniques for robust system design. In: **ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2013. p. 1–10. ISSN 0738-100X.

CLABES, J. et al. Design and implementation of the POWER5^{TM} microprocessor. In: ACM. **Annual Design Automation Conference (DAC)**. [S.l.], 2004. p. 670–672.

CLARK, L. T. et al. An embedded 32-b microprocessor core for low-power and high-performance applications. **Solid-State Circuits, IEEE Journal of**, IEEE, v. 36, n. 11, p. 1599–1608, 2001.

COLWELL, R. P.; O'DONNELL, J.; PAPWORTH, D. B.; RODMAN, P. K. **Instruction storage method with a compressed format using a mask word**. [S.l.]: US Patent 5057837, 1991.

CONTE, T. M.; BANERJIA, S.; LARIN, S. Y.; MENEZES, K. N.; SATHAYE, S. W. Instruction fetch mechanisms for VLIW architectures with compressed encodings. In: IEEE. **Microarchitecture (MICRO). IEEE/ACM International Symposium on**. [S.l.], 1996. p. 201–211.

de Lima Kastensmidt, F. G.; NEUBERGER, G.; HENTSCHKE, R. F.; CARRO, L.; REIS, R. Designing fault-tolerant techniques for SRAM-based FPGAs. **IEEE Design & Test of Computers**, IEEE, n. 6, p. 552–562, 2004.

DE, V.; BORKAR, S. Technology and design challenges for low power and high performance. In: ACM. **Low power electronics and design, international symposium on**. [S.l.], 1999. p. 163–168.

DONGARRA, J.; HEROUX, M. A.; LUSZCZEK, P. HPCG benchmark: a new metric for ranking high performance computing systems. **Knoxville, Tennessee**, 2015.

EIJNDHOVEN, J. T. J. van et al. TriMedia CPU64 architecture. In: IEEE. **Computer Design (ICCD), International Conference on**. [S.l.], 1999. p. 586–592.

EJLALI, A.; MIREMADI, S. G.; ZARANDI, H.; ASADI, G.; SARMADI, S. B. A hybrid fault injection approach based on simulation and emulation co-operation. In: **Dependable Systems and Networks. International Conference on**. [S.l.: s.n.], 2003. p. 479–488.

ERICHSEN, A. G. et al. ISA-DTMR: Selective Protection in Configurable Heterogeneous Multicores. In: **International Symposium on Applied Reconfigurable Computing (ARC)**. [S.l.: s.n.], 2018.

FAY, D.; SHYE, A.; BHATTACHARYA, S.; CONNORS, D. A.; WICHMANN, S. An adaptive fault-tolerant memory system for FPGA-based architectures in the space environment. In: IEEE. **Adaptive Hardware and Systems, NASA/ESA Conference on**. [S.l.], 2007. p. 250–257.

FISHER, J. A.; FARABOSCHI, P.; YOUNG, C. **Embedded computing: a VLIW approach to architecture, compilers and tools**. [S.l.]: Elsevier, 2005.

GAISLER, J. Evaluation of a 32-bit microprocessor with built-in concurrent error-detection. In: IEEE. **Fault-Tolerant Computing (FTCS). International Symposium on**. [S.l.], 1997. p. 42–46.

GAO, C. et al. A study of thread level parallelism on mobile devices. In: IEEE. **Performance Analysis of Systems and Software (ISPASS), IEEE International Symposium on**. [S.l.], 2014. p. 126–127.

GEUSKENS, B.; ROSE, K. **Modeling microprocessor performance**. [S.l.]: Springer Science & Business Media, 2012.

GIBSON, D.; WOOD, D. A. Forwardflow: a scalable core for power-constrained CMPs. In: ACM. **ACM SIGARCH Computer Architecture News**. [S.l.], 2010. v. 38, n. 3, p. 14–25.

GIRALDO, J. S. P.; SARTOR, A. L.; CARRO, L.; WONG, S.; BECK, A. C. S. Evaluation of energy savings on a VLIW processor through dynamic issue-width adaptation. In: IEEE. **Rapid System Prototyping (RSP), International Symposium on**. [S.l.], 2015. p. 11–17.

GIRALDO, J. S. P.; WONG, S.; BECK, A. C. S. Leveraging Compiler Support on VLIW processors for Efficient Power Gating. In: **VLSI (ISVLSI), IEEE Computer Society Annual Symposium on**. [S.l.: s.n.], 2016.

GOMAA, M. A.; VIJAYKUMAR, T. N. Opportunistic transient-fault detection. In: IEEE. **Computer Architecture (ISCA), International Symposium on**. [S.l.], 2005. p. 172–183.

GOSWAMI, K. K. DEPEND: a simulation-based environment for system level dependability analysis. **IEEE Transactions on Computers**, v. 46, n. 1, p. 60–74, jan 1997. ISSN 0018-9340.

GUO, Q. et al. Run-time phase prediction for a reconfigurable VLIW processor. In: IEEE. **Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2016. p. 1634–1639.

GUPTA, S.; FENG, S.; ANSARI, A.; MAHLKE, S. Erasing core boundaries for robust and configurable performance. In: IEEE COMPUTER SOCIETY. **Microarchitecture, IEEE/ACM International Symposium on**. [S.l.], 2010. p. 325–336.

GUSTAFSSON, J.; BETTS, A.; ERMEDAHL, A.; LISPER, B. The Mälardalen WCET Benchmarks: Past, Present And Future. **WCET**, v. 15, p. 136–146, 2010.

HARI, S. K. S.; ADVE, S. V.; NAEIMI, H.; RAMACHANDRAN, P. Relyzer: Exploiting Application-level Fault Equivalence to Analyze Application Resiliency to Transient Faults. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 47, n. 4, p. 123–134, 2012. ISSN 0362-1340.

HAUSER, J. Berkeley SoftFloat. 2002. Available from Internet: <http://www.jhauser.us/arithmetic/SoftFloat.html>. Accessed in: 17 Feb. 2018.

HAZUCHA, P.; SVENSSON, C. Impact of CMOS technology scaling on the atmospheric neutron soft error rate. **Nuclear Science, IEEE Transactions on**, IEEE, v. 47, n. 6, p. 2586–2594, 2000.

HAZUCHA, P.; SVENSSON, C.; WENDER, S. A. Cosmic-ray soft error rate characterization of a standard 0.6-/spl mu/m cmos process. **Solid-State Circuits, IEEE Journal of**, IEEE, v. 35, n. 10, p. 1422–1429, 2000.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. [S.l.]: Elsevier Science, 2017. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 9780128119068.

HILL, M. D.; MARTY, M. R. Amdahl's law in the multicore era. **Computer**, IEEE, n. 7, p. 33–38, 2008.

HOOZEMANS, J.; JOHANSEN, J.; STRATEN, J. V.; BRANDON, A.; WONG, S. Multiple contexts in a multi-ported VLIW register file implementation. In: **International Conference on ReConFigurable Computing and FPGAs (ReConFig)**. [S.l.: s.n.], 2015. p. 1–6.

HOOZEMANS, J.; STRATEN, J. van; WONG, S. Using a polymorphic VLIW processor to improve schedulability and performance for mixed-criticality systems. In: **IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)**. [S.l.: s.n.], 2017. p. 1–9.

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

HU, J. et al. Compiler-assisted soft error detection under performance and energy constraints in embedded systems. **ACM Transactions on Embedded Computing Systems (TECS)**, ACM, v. 8, n. 4, p. 27, 2009.

HU, J. S. et al. Compiler-directed instruction duplication for soft error detection. In: IEEE COMPUTER SOCIETY. **Design, Automation and Test in Europe (DATE)**. [S.l.], 2005. p. 1056–1057.

HU, Z. et al. Microarchitectural techniques for power gating of execution units. In: ACM. **Low power electronics and design, international symposium on**. [S.l.], 2004. p. 32–37.

HUBENER, B.; SIEVERS, G.; JUNGEBLUT, T.; PORRMANN, M.; RUCKERT, U. CoreVA: A Configurable Resource-Efficient VLIW Processor Architecture. In: IEEE. **Embedded and Ubiquitous Computing (EUC), IEEE International Conference on**. [S.l.], 2014. p. 9–16.

HUCK, J. et al. Introducing the IA-64 architecture. **IEEE Micro**, v. 20, n. 5, p. 12–23, 2000. ISSN 0272-1732.

IBTESHAM, D.; DEBONIS, D.; ARNOLD, D.; FERREIRA, K. B. Coarse-Grained Energy Modeling of Rollback/Recovery Mechanisms. In: IEEE. **Dependable Systems and Networks (DSN), IEEE/IFIP International Conference on**. [S.l.], 2014. p. 708–713.

INSTRUMENTS, T. TMS320C6745/C6747 DSP technical reference manual. **SPRUH91A, Texas Instruments Inc**, 2011.

IPEK, E.; KIRMAN, M.; KIRMAN, N.; MARTINEZ, J. F. Core fusion: accommodating software diversity in chip multiprocessors. **SIGARCH Comput. Archit. News**, v. 35, n. 2, p. 186–197, 2007. ISSN 0163-5964.

JACOBS, A.; CIESLEWSKI, G.; GEORGE, A. D.; GORDON-ROSS, A.; LAM, H. Reconfigurable fault tolerance: A comprehensive framework for reliable and adaptive FPGA-based space computing. **ACM Transactions on Reconfigurable Technology and Systems (TRETS)**, ACM, v. 5, n. 4, p. 21, 2012.

JEE, S.; PALANIAPPAN, K. Performance evaluation for a compressed-VLIW processor. In: ACM. **Applied computing, ACM symposium on**. [S.l.], 2002. p. 913–917.

JEONG, K.; KAHNG, A. B.; KANG, S.; ROSING, T. S.; STRONG, R. MAPG: Memory Access Power Gating. In: **Proceedings of the Conference on Design, Automation and Test in Europe (DATE)**. San Jose, CA, USA: EDA Consortium, 2012. (DATE '12), p. 1054–1059. ISBN 978-3-9810801-8-6.

JOST, T. T.; NAZAR, G. L.; CARRO, L. Scalable memory architecture for soft-core processors. In: **IEEE International Conference on Computer Design (ICCD)**. [S.l.: s.n.], 2016. p. 396–399.

KALBARCZYK, Z. et al. Hierarchical Simulation Approach to Accurate Fault Modeling for System Dependability Evaluation. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 25, n. 5, p. 619–632, 1999. ISSN 0098-5589.

KALIORAKIS, M.; TSELONIS, S.; CHATZIDIMITRIOU, A.; GIZOPOULOS, D. Accelerated microarchitectural Fault Injection-based reliability assessment. In: **IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)**. [S.l.: s.n.], 2015. p. 47–52. ISSN 1550-5774.

KAXIRAS, S.; MARTONOSI, M. Computer architecture techniques for power-efficiency. **Synthesis Lectures on Computer Architecture**, Morgan & Claypool Publishers, v. 3, n. 1, p. 1–207, 2008.

KHUBAIB, K.; SULEMAN, M. A.; HASHEMI, M.; WILKERSON, C.; PATT, Y. N. Morphcore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP. In: IEEE. **Microarchitecture (MICRO), IEEE/ACM International Symposium on**. [S.l.], 2012. p. 305–316.

KIM, C. et al. Composable lightweight processors. In: IEEE. **Microarchitecture, IEEE/ACM International Symposium on**. [S.l.], 2007. p. 381–394.

KOBAYASHI, H. et al. Comparison between neutron-induced system-SER and accelerated-SER in SRAMs. In: IEEE. **Reliability Physics Symposium, IEEE International**. [S.l.], 2004. p. 288–293.

KOOLI, M.; NATALE, G. D.; BOSIO, A. Cache-aware reliability evaluation through LLVM-based analysis and fault injection. In: **IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)**. [S.l.: s.n.], 2016. p. 19–22.

LATTNER, C.; ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In: IEEE COMPUTER SOCIETY. **Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization**. [S.l.], 2004. p. 75.

LESEA, A.; DRIMER, S.; FABULA, J. J.; CARMICHAEL, C.; ALFKE, P. The rosetta experiment: atmospheric soft error rate testing in differing technology FPGAs. **Device and Materials Reliability, IEEE Transactions on**, IEEE, v. 5, n. 3, p. 317–328, 2005.

LI, M. L.; RAMACHANDRAN, P.; KARPUZCU, U. R.; HARI, S. K. S.; ADVE, S. V. Accurate microarchitecture-level fault modeling for studying hardware faults. In: **IEEE International Symposium on High Performance Computer Architecture**. [S.l.: s.n.], 2009. p. 105–116. ISSN 1530-0897.

LI, S.; CHEN, K.; AHN, J. H.; BROCKMAN, J. B.; JOUPPI, N. P. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In: **ICCAD: International Conference on Computer-Aided Design**. [S.l.: s.n.], 2011. p. 694–701.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy. **Journal of Parallel and Distributed Computing**, Elsevier, v. 95, p. 107–123, 2016.

LORENZON, A. F.; SARTOR, A. L.; CERA, M. C.; Schneider Beck, A. C. Optimized Use of Parallel Programming Interfaces in Multithreaded Embedded Architectures. In: IEEE. **VLSI (ISVLSI), IEEE Computer Society Annual Symposium on**. [S.l.], 2015. p. 410–415.

LORENZON, A. F.; SARTOR, A. L.; CERA, M. C.; BECK, A. C. S. The Influence of Parallel Programming Interfaces on Multicore Embedded Systems. In: IEEE. **Computer Software and Applications Conference (COMPSAC), IEEE Annual**. [S.l.], 2015. v. 2, p. 617–625.

MADAN, N.; BALASUBRAMONIAN, R. Power efficient approaches to redundant multithreading. **Parallel and Distributed Systems, IEEE Transactions on**, IEEE, v. 18, n. 8, p. 1066–1079, 2007.

MAGALHAES, G. G.; SARTOR, A. L.; LORENZON, A. F.; NAVAUX, P. O. A.; BECK, A. C. S. How Programming Languages and Paradigms Affect Performance and Energy in Multithreaded Applications. In: **Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.: s.n.], 2016. p. 71–78.

MAGNUSSON, P. S. et al. Simics: A full system simulation platform. **Computer**, v. 35, n. 2, p. 50–58, 2002. ISSN 0018-9162.

MAHMOOD, A.; MCCLUSKEY, E. J. Concurrent error detection using watchdog processors-a survey. **IEEE Transactions on Computers**, IEEE, v. 37, n. 2, p. 160–174, 1988.

MAIR, H. et al. A 65-nm mobile multimedia applications processor with an adaptive power management scheme to compensate for variations. In: **VLSI Circuits, IEEE Symposium on**. [S.l.: s.n.], 2007.

MARTIN, M. M. K. et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 33, n. 4, p. 92–99, 2005. ISSN 0163-5964.

MCCALPIN, J. D. A survey of memory bandwidth and machine balance in current high performance computers. **IEEE TCCA Newsletter**, v. 19, p. 25, 1995.

MCNAIRY, C.; BHATIA, R. Montecito: A dual-core, dual-thread Itanium processor. **IEEE micro**, IEEE, n. 2, p. 10–20, 2005.

MITRA, S.; HUANG, W.-J.; SAXENA, N. R.; YU, S.-Y.; MCCLUSKEY, E. J. Reconfigurable architecture for autonomous self-repair. **IEEE Design & Test**, IEEE Computer Society Press, v. 21, n. 3, p. 228–240, 2004.

MITROPOULOU, K.; PORPODAS, V.; CINTRA, M. DRIFT: Decoupled CompileR-Based Instruction-Level Fault-Tolerance. In: **Languages and Compilers for Parallel Computing**. [S.l.]: Springer, 2014. p. 217–233.

MORAD, T. Y.; WEISER, U. C.; KOLODNY, A.; VALERO, M.; AYGUADE, E. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. **Computer Architecture Letters**, IEEE, v. 5, n. 1, p. 14–17, 2006.

MUKHERJEE, S. S.; WEAVER, C.; EMER, J.; REINHARDT, S. K.; AUSTIN, T. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: IEEE COMPUTER SOCIETY. **Microarchitecture, IEEE/ACM International Symposium on**. [S.l.], 2003. p. 29.

NAKKA, N.; PATTABIRAMAN, K.; IYER, R. Processor-level selective replication. In: IEEE. **Dependable Systems and Networks, IEEE/IFIP International Conference on**. [S.l.], 2007. p. 544–553.

NAZAR, G. L.; CARRO, L. Fast single-FPGA fault injection platform. In: IEEE. **Defect and fault tolerance in VLSI and nanotechnology systems (DFT), IEEE international symposium on**. [S.l.], 2012. p. 152–157.

NOJI, R.; FUJIE, S.; YOSHIKAWA, Y.; ICHIHARA, H.; INOUE, T. An FPGA-based fail-soft system with adaptive reconfiguration. In: IEEE. **On-Line Testing Symposium, IEEE International**. [S.l.], 2010. p. 127–132.

OH, N.; MCCLUSKEY, E. J. Error detection by selective procedure call duplication for low energy consumption. **Reliability, IEEE Transactions on**, IEEE, v. 51, n. 4, p. 392–402, 2002.

PARASYRIS, K.; TZIANTZOULIS, G.; ANTONOPOULOS, C. D.; BELLAS, N. GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates. In: IEEE. **Dependable Systems and Networks (DSN), IEEE/IFIP International Conference on**. [S.l.], 2014. p. 622–629.

PATEL, A.; AFRAM, F.; CHEN, S.; GHOSE, K. MARSS: A Full System Simulator for Multicore x86 CPUs. In: **Design Automation Conference (DAC)**. New York, NY, USA: ACM, 2011. (DAC '11), p. 1050–1055. ISBN 978-1-4503-0636-2.

PETERSEN, W.; ARBENZ, P. **Introduction to Parallel Computing : A practical guide with examples in C: A practical guide with examples in C**. [S.l.]: OUP Oxford, 2004. (Oxford Texts in Applied and Engineering Mathematics). ISBN 9780191513619.

PONOMAREV, D.; KUCUK, G.; GHOSE, K. Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources. In: **International Symposium on Microarchitecture**. Washington, DC, USA: IEEE Computer Society, 2001. (MICRO), p. 90–101. ISBN 0-7695-1369-7.

POP, P.; POULSEN, K. H.; IZOSIMOV, V.; ELES, P. Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In: ACM. **Hardware/software codesign and system synthesis, IEEE/ACM international conference on**. [S.l.], 2007. p. 233–238.

PRICOPI, M.; MITRA, T. Bahurupi: A polymorphic heterogeneous multi-core architecture. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM, v. 8, n. 4, p. 22, 2012.

PSIAKIS, R.; KRITIKAKOU, A.; SENTIEYS, O. NEDA: NOP Exploitation with Dependency Awareness for Reliable VLIW Processors. In: **IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.: s.n.], 2017. p. 391–396.

PUTNAM, A.; SMITH, A.; BURGER, D. Dynamic vectorization in the E2 dynamic multicore architecture. **ACM SIGARCH Computer Architecture News**, ACM, v. 38, n. 4, p. 27–32, 2011.

QUINN, M. J. **Parallel Programming in C with MPI and OpenMP**. [S.l.]: McGraw-Hill Education Group, 2003. ISBN 0071232656.

RAJE, P. A.; SIU, S. C. **Method and apparatus for sequencing and decoding variable length instructions with an instruction boundary marker within each instruction**. [S.l.]: US Patent 5881260, 1999.

RAMACHANDRAN, P.; KUDVA, P.; KELLINGTON, J.; SCHUMANN, J.; SANDA, P. Statistical Fault Injection. In: **IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)**. [S.l.: s.n.], 2008. p. 122–127. ISSN 1530-0889.

RAMÍREZ, T. et al. Mitigating lower layer failures with adaptive system reconfiguration. In: **Proceedings of the International Conference Mixed Design of Integrated Circuits and Systems (MIXDES)**. [S.l.: s.n.], 2012. p. 109–114.

RAO, R. M.; BURNS, J. L.; DEVGAN, A.; BROWN, R. B. Efficient techniques for gate leakage estimation. In: ACM. **Low power electronics and design, international symposium on**. [S.l.], 2003. p. 100–103.

RASHID, M. W.; TAN, E. J.; HUANG, M. C.; ALBONESI, D. H. Exploiting coarse-grain verification parallelism for power-efficient fault tolerance. In: IEEE. **Parallel Architectures and Compilation Techniques, International Conference on**. [S.l.], 2005. p. 315–325.

RAY, J.; HOE, J. C.; FALSAFI, B. Dual use of superscalar datapath for transient-fault detection and recovery. In: IEEE COMPUTER SOCIETY. **Microarchitecture, ACM/IEEE international symposium on**. [S.l.], 2001. p. 214–224.

REINHARDT, S. K.; MUKHERJEE, S. S. Transient Fault Detection via Simultaneous Multithreading. In: **Computer Architecture (ISCA), International Symposium on**. New York, NY, USA: ACM, 2000. (ISCA '00), p. 25–36. ISBN 1-58113-232-8.

REIS, G. A. et al. Design and evaluation of hybrid fault-detection systems. In: **International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2005. p. 148–159. ISSN 1063-6897.

REIS, G. A.; CHANG, J.; VACHHARAJANI, N.; RANGAN, R.; AUGUST, D. I. SWIFT: Software implemented fault tolerance. In: IEEE COMPUTER SOCIETY. **Code generation and optimization, international symposium on**. [S.l.], 2005. p. 243–254.

ROTENBERG, E. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In: IEEE. **Fault-Tolerant Computing, Twenty-Ninth Annual International Symposium on**. [S.l.], 1999. p. 84–91.

RUSU, S. et al. A 65-nm dual-core multithreaded Xeon{®} processor with 16-MB L3 cache. **Solid-State Circuits, IEEE Journal of**, IEEE, v. 42, n. 1, p. 17–25, 2007.

SANTINI, T. et al. Evaluation of Failures Masking Across the Software Stack. In: **MEDIAN Workshop**. [S.l.: s.n.], 2015.

SARTOR, A. L.; BECK, A. C. S. Multi-architecture profiler for Android. **International Journal of High Performance Systems Architecture**, v. 7, n. 1, p. 41–55, 2017.

SARTOR, A. L.; BECKER, P. H. E.; BECK, A. C. S. Simbah-FI: Simulation-Based Hybrid Fault Injector. In: **Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.: s.n.], 2017. p. 94–101.

SARTOR, A. L.; BECKER, P. H. E.; HOOZEMANS, J.; WONG, S.; BECK, A. C. S. Dynamic Trade-off among Fault Tolerance, Energy Consumption, and Performance on a Multiple-issue VLIW Processor. **IEEE Transactions on Multi-Scale Computing Systems**, PP, n. 99, 2017.

SARTOR, A. L.; CAPELLA, F. M.; BRANDALERO, M.; CARRO, L.; BECK, A. C. S. A Transparent Multiple-ISA MPSoC Architecture. In: **SoCs, Heterogeneous Architectures and Workloads, Workshop on**. [S.l.: s.n.], 2014.

SARTOR, A. L.; LORENZON, A. F.; BECK, A. The impact of virtual machines on embedded systems. In: IEEE. **Computer Software and Applications Conference (COMPSAC), IEEE Annual**. [S.l.], 2015. v. 2, p. 626–631.

SARTOR, A. L. et al. A Novel Phase-Based Low Overhead Fault Tolerance Approach for VLIW Processors. In: IEEE. **VLSI (ISVLSI), IEEE Computer Society Annual Symposium on**. [S.l.], 2015. p. 485–490.

SARTOR, A. L. et al. Exploiting Idle Hardware to Provide Low Overhead Fault Tolerance for VLIW Processors. **ACM Journal on Emerging Technologies in Computing Systems**, ACM, New York, NY, USA, v. 13, n. 2, p. 13:1—-13:21, 2017. ISSN 1550-4832.

SARTOR, A. L.; LORENZON, A. F.; KUNDU, S.; KOREN, I.; BECK, A. C. S. Adaptive and Polymorphic VLIW Processor to Optimize Fault Tolerance, Energy Consumption, and Performance. In: **ACM International Conference on Computing Frontiers**. [S.l.: s.n.], 2018.

SARTOR, A. L.; WONG, S.; BECK, A. C. S. Adaptive ILP Control to increase Fault Tolerance for VLIW Processors. In: **IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)**. [S.l.]: IEEE, 2016.

SATO, T.; FUNAKI, T. Dependability, power, and performance trade-off on a multicore processor. In: **Asia and South Pacific Design Automation Conference (ASPDAC)**. [S.l.: s.n.], 2008. p. 714–719. ISSN 2153-6961.

SCHÖLZEL, M. Reduced Triple Modular redundancy for built-in self-repair in VLIW-processors. In: IEEE. **Signal Processing Algorithms, Architectures, Arrangements and Applications**. [S.l.], 2007. p. 21–26.

SCOTT, J.; LEE, L. H.; ARENDS, J.; MOYER, B. Designing the Low-Power MCORE Architecture. In: **Power driven microarchitecture workshop**. [S.l.: s.n.], 1998. p. 145–150.

SEO, S.; JO, G.; LEE, J. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In: **IEEE International Symposium on Workload Characterization (IISWC)**. [S.l.: s.n.], 2011. p. 137–148.

SHARANGPANI, H.; ARORA, K. Itanium processor microarchitecture. **Micro, IEEE**, IEEE, v. 20, n. 5, p. 24–43, 2000.

SHERWOOD, T.; PERELMAN, E.; HAMERLY, G.; SAIR, S.; CALDER, B. Discovering and exploiting program phases. **Micro, IEEE**, IEEE, v. 23, n. 6, p. 84–93, 2003.

SHERWOOD, T.; SAIR, S.; CALDER, B. Phase Tracking and Prediction. In: **Proceedings of the Annual International Symposium on Computer Architecture**. New York, NY, USA: ACM, 2003. (ISCA ’03), p. 336–349. ISBN 0-7695-1945-8.

SHIVAKUMAR, P.; KISTLER, M.; KECKLER, S.; BURGER, D.; ALVISI, L. Modeling the effect of technology trends on the soft error rate of combinational logic. **Dependable Systems and Networks (DSN), International Conf. on**, p. 389–398, 2002.

SLEGEL, T. J. et al. IBM’s S/390 G5 microprocessor design. **Micro, IEEE**, IEEE, v. 19, n. 2, p. 12–23, 1999.

SMOLENS, J. C.; KIM, J.; HOE, J. C.; FALSAFI, B. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In: IEEE COMPUTER SOCIETY. **Microarchitecture (MICRO), IEEE/ACM International Symposium on**. [S.l.], 2004. p. 257–268.

SNIR, M. **MPI–the Complete Reference: The MPI core**. [S.l.]: MIT press, 1998.

SOUZA, J. D. et al. DIM-VEX: Exploiting Design Time Configurability and Runtime Reconfigurability. In: **International Symposium on Applied Reconfigurable Computing (ARC)**. [S.l.: s.n.], 2018.

SRINIVASAN, S.; KOREN, I.; KUNDU, S. Online mechanism for reliability and power-efficiency management of a dynamically reconfigurable core. In: IEEE. **Computer Design (ICCD), 33rd International Conference on**. [S.l.], 2015. p. 327–334.

SRINIVASAN, S. et al. Toward increasing FPGA lifetime. **Dependable and Secure Computing, IEEE Transactions on**, IEEE, v. 5, n. 2, p. 115–127, 2008.

SUBRAMANYAN, P.; SINGH, V.; SALUJA, K. K.; LARSSON, E. Energy-efficient fault tolerance in chip multiprocessors using critical value forwarding. In: IEEE. **Dependable Systems and Networks (DSN), IEEE/IFIP International Conference on**. [S.l.], 2010. p. 121–130.

SUGA, A.; MATSUNAMI, K. Introducing the FR500 embedded microprocessor. **Micro, IEEE**, IEEE, v. 20, n. 4, p. 21–27, 2000.

SULEMAN, M. A.; MUTLU, O.; QURESHI, M. K.; PATT, Y. N. Accelerating critical section execution with asymmetric multi-core architectures. In: ACM. **ACM SIGARCH Computer Architecture News**. [S.l.], 2009. v. 37, n. 1, p. 253–264.

TANENBAUM, A. S.; WOODHULL, A. S. **Operating systems: design and implementation**. [S.l.]: Prentice-Hall Englewood Cliffs, NJ, 1987.

TREMBLAY, M.; CHAN, J.; CHAUDHRY, S.; CONIGLIARO, A. W.; TSE, S. S. The MAJC architecture: A synthesis of parallelism and scalability. **IEEE Micro**, IEEE, n. 6, p. 12–25, 2000.

TUNE, E.; LIANG, D.; TULLSEN, D. M.; CALDER, B. Dynamic prediction of critical path instructions. In: IEEE. **High-Performance Computer Architecture (HPCA), The Seventh International Symposium on**. [S.l.], 2001. p. 185–195.

VIOLANTE, M. et al. A new hardware/software platform and a new 1/E neutron source for soft error studies: testing FPGAs at the ISIS facility. **Nuclear Science, IEEE Transactions on**, IEEE, v. 54, n. 4, p. 1184–1189, 2007.

WAERDT, J.-W. de et al. The TM3270 media-processor. In: IEEE COMPUTER SOCIETY. **Microarchitecture, IEEE/ACM International Symposium on**. [S.l.], 2005. p. 331–342.

WALI, I.; VIRAZEL, A.; BOSIO, A.; DILILLO, L.; GIRARD, P. An effective hybrid fault-tolerant architecture for pipelined cores. In: IEEE. **Test Symposium (ETS), IEEE European**. [S.l.], 2015. p. 1–6.

WANG, H.; KOREN, I.; KRISHNA, C. M. Utilization-Based Resource Partitioning for Power-Performance Efficiency in SMT Processors. **IEEE Transactions on Parallel and Distributed Systems**, v. 22, n. 7, p. 1150–1163, 2011. ISSN 1045-9219.

WATANABE, Y.; DAVIS, J. D.; WOOD, D. A. WiDGET: Wisconsin decoupled grid execution tiles. In: ACM. **ACM SIGARCH Computer Architecture News**. [S.l.], 2010. v. 38, n. 3, p. 2–13.

WEAVER, C.; EMER, J.; MUKHERJEE, S. S.; REINHARDT, S. K. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In: **International Symposium on Computer Architecture**. Washington, DC, USA: IEEE Computer Society, 2004. (ISCA '04), p. 264—-. ISBN 0-7695-2143-6.

Wind River. **Simics - Supported Targets**. 2017. Available from Internet: <http://www.windriver.com/products/simics/simics-supported-targets.html>. Accessed in: 17 Feb. 2018.

WONG, S.; Van As, T.; BROWN, G. ρ-VEX: A reconfigurable and extensible softcore VLIW processor. In: IEEE. **ICECE Technology, International Conference on**. [S.l.], 2008. p. 369–372.

XIAOGUANG, R. et al. GS-DMR: Low-overhead soft error detection scheme for stencil-based computation. **Parallel Computing**, Elsevier, v. 41, p. 50–65, 2015.

YAHAGI, Y.; SAITO, Y.; TERUNUMA, K.; NUNOMIYA, T.; NAKAMURA, T. Self-consistent integrated system for susceptibility to terrestrial neutron induced soft-error of sub-quarter micron memory devices. In: IEEE. **Integrated Reliability Workshop, IEEE International**. [S.l.], 2002. p. 143–146.

YALCIN, G.; UNSAL, O. S.; CRISTAL, A.; VALERO, M. FIMSIM: A fault injection infrastructure for microarchitectural simulators. In: **IEEE International Conference on Computer Design (ICCD)**. [S.l.: s.n.], 2011. p. 431–432. ISSN 1063-6404.

YANG, J.-M.; KWAK, S. W. A checkpoint scheme with task duplication considering transient and permanent faults. In: IEEE. **Industrial Engineering and Engineering Management (IEEM), IEEE International Conference on**. [S.l.], 2010. p. 606–610.

ZHANG, Y.; CHAKRABARTY, K. Dynamic adaptation for fault tolerance and power management in embedded real-time systems. **ACM Transactions on Embedded Computing Systems (TECS)**, ACM, v. 3, n. 2, p. 336–360, 2004.

ZHU, D.; MELHEM, R.; MOSSÉ, D. The effects of energy management on reliability in real-time embedded systems. In: IEEE. **Computer Aided Design (ICCAD), IEEE/ACM International Conference on**. [S.l.], 2004. p. 35–40.

APPENDIX A — PHASE-CONFIGURABLE DUPLICATION

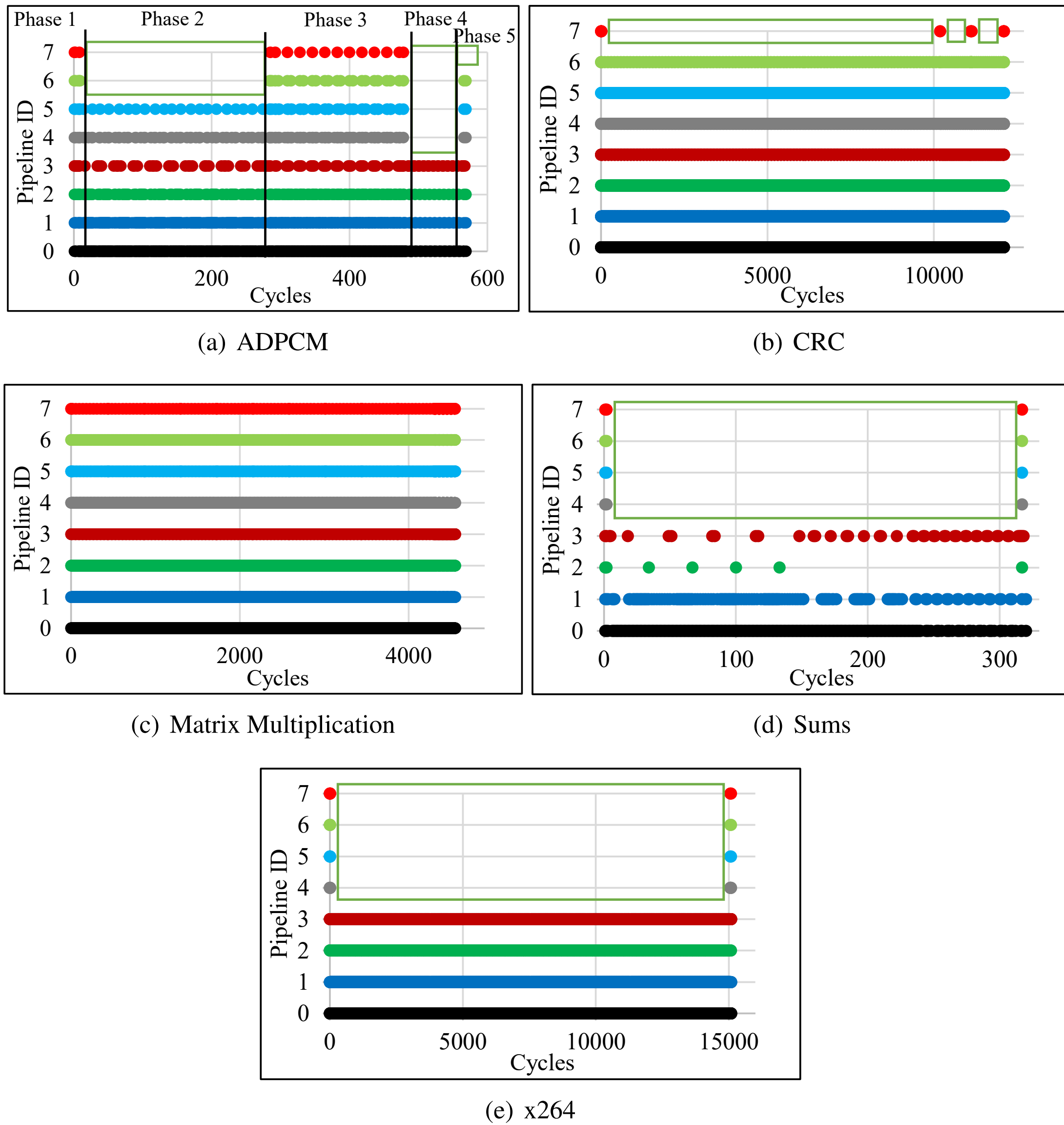
This appendix presents a technique that was also implemented, but it was not integrated to any version of the proposed processor due to its static application profiling requirement.

Many programs have wildly different behavior during its execution. During one part, it can have high Instruction-Level Parallelism (ILP), while in another extremely low ILP (for example, due to memory-intensive operations). Each part is called phase, which can be defined as a set of intervals within a program's execution that have similar behavior, regardless of temporal adjacency (SHERWOOD et al., 2003). Taking advantage of phase information allows the use of several optimization mechanisms to increase fault tolerance or reduce energy consumption.

In this first fault tolerance approach, idle pipelanes during a whole program phase (i.e., a sequence of instructions words that always have No-Operations (NOPs) in specific issue slots) are used to execute duplicated instructions from other pipelanes. The first step for this approach is to profile the application, in order to detect the phases. After that, a table indexed by the program counter (32 bits) and containing the configuration of each application's phase (4 bits) is created. The phase configuration represents the function of each pipelane in a given phase, informing whether each issue slot will execute regular instructions of the application or execute duplicated instructions from another pipelane. Based on this table, the processor will dynamically change the function of the pipelanes and will enable or disable the checkers in each phase.

The profiling was performed for all applications from our benchmark set. The results for five benchmarks are depicted in Figure A.1. The dots demonstrate when a given pipelane, identified by its ID (*Y-axis*), is being used (i.e., executing program instructions) in a given moment of the application's execution (*X-axis*). The profiling for the other five benchmarks has a similar behavior to the one from the *Matrix Multiplication* benchmark (i.e., there are no idle phases). The idle phases that were used to execute duplicated instructions are highlighted in Figure A.1 (empty blocks in Figure A.1(a/b/d/e)). The blank areas of a given pipelane ID represent a period of time in which this pipelane is idle (executing NOPs only). The idle phases are detected when a given pipelane spends more than 5% of the execution time of the application only executing NOPs (note that other values can be chosen to the detection of these phases, but in this work, we restrained ourselves with only this value in order to evaluate other mechanisms and their trade-offs).

Figure A.1: Issue Utilization and Phase-configurable Duplication



(a) ADPCM

(b) CRC

(c) Matrix Multiplication

(d) Sums

(e) x264

Source: The Author

Figure A.2 depicts each phase for the *ADPCM* benchmark: the same representation is used to present the pipelanes (i.e., P_0-P_7 represent the pipelanes *0* to *7*). The pipelanes in white background are executing duplicated instructions from the other pipelane, according to their respective pairs (as discussed in Figure 3.1). The pipelanes in black background are executing main program instructions. In this example, there are phases with full duplication (*phase 4*), partial duplication (*phases 2* and *5*) and no duplication (*phases 1* and *3*).

As it can be noticed, the *ADPCM*, *CRC*, *Sums*, and *x264* benchmarks have phases when some issue slots are not utilized. On the other hand, as the *Matrix Multiplication*, *CJPEG*, *DFT*, *Expint*, *FIR*, and *NDES* benchmarks do not have such phases, they cannot take advantage of the phase-configurable approach, because the modified processor would

Figure A.2: Phase-configurable Duplication for the *ADPCM* Benchmark

Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
P7	P7	P7	P7	P7
P6	P6	P6	P6	P6
P5	P5	P5	P5	P5
P4	P4	P4	P4	P4
P3	P3	P3	P3	P3
P2	P2	P2	P2	P2
P1	P1	P1	P1	P1
P0	P0	P0	P0	P0

Source: The Author

have the same behavior as the unprotected version. Therefore, even though this approach has no costs in terms of performance and negligible power overhead, it can be only used when the application has phases with lower ILP than the processor supports.

A.1 Results

Table A.1 presents the results in terms of performance and failure rate also considering the phase-configurable approach. On average, it achieves a failure rate of 0.45% (considering only benchmarks with phases).

Table A.2 presents the area (both Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (ASIC) versions) and power consumption (ASIC only) for all Very Long Instruction Word (VLIW) configurations. The overhead is almost negligible when one compares the phase-configurable approach with the base 8-issue configuration: 2.6% for the FPGA (in Look-Up Tables (LUTs)) and 2.8% for the ASIC.

Table A.1: Failure Rate and Performance Degradation Comparison

		Unprot.	Prot.	Unprot.	Protected								
		4-issue	Full dup.	8-issue	Phase-config.	Spatial Dup.	Threshold						
ADPCM	Failure rate (%)	6.93	0.06	3.66	0.99	0.66	T = 1	0.59	T = 1.75	0.65	T = 2	0.66
	Exec. Cycles	571	571	568	568	568		633		621		574
CJPEG	Failure rate (%)	9.55	0.02	6.07	6.07	2.33	T = 1	0.79	T = 2.5	2.12		
	Exec. Cycles	508	508	411	411	411		523		426		
CRC	Failure rate (%)	5.20	0.06	2.95	0.64	0.33	T = 1	0.32				
	Exec. Cycles	13289	13289	13270	13270	13270		13616				
DFT	Failure rate (%)	4.63	0.07	2.68	2.68	0.38	T = 1	0.15				
	Exec. Cycles	35072	35072	32575	32575	32575		32979				
Expint	Failure rate (%)	4.21	0.05	2.37	2.37	0.13	T = 1	0.13				
	Exec. Cycles	9341	9341	9097	9097	9097		9257				
FIR	Failure rate (%)	10.94	0.04	5.93	5.93	1.21	T = 1	0.93				
	Exec. Cycles	119392	119392	111769	111769	111769		120095				
Matrix Mul.	Failure rate (%)	9.91	0.08	5.68	5.68	1.30	T = 1	0.17	T = 2	0.53		
	Exec. Cycles	111050	111050	111025	111025	111025		113929		112547		
NDES	Failure rate (%)	3.99	0.04	2.09	2.09	0.42	T = 1	0.24				
	Exec. Cycles	28527	28527	27499	27499	27499		28667				
Sums	Failure rate (%)	5.52	0.04	2.96	0.11	0.37	T = 1	0.37				
	Exec. Cycles	332	332	319	319	319		319				
x264	Failure rate (%)	5.21	0.09	2.94	0.07	0.33	T = 1	0.33				
	Exec. Cycles	15102	15102	15089	15089	15089		15090				

Source: The Author

Table A.2: Area and Power Dissipation Comparison

		FPGA		ASIC	
		Registers	LUTs	Cells	Power dissipation (nW)
Unprotected	4-issue	3,058	16,006	28,041	2,298,962.51
	8-issue	3,974	35,075	66,967	7,484,818.25
Protected	Full. Dup. (4-issue)	4,102	20,819	42,121	3,109,613.33
	Phase-configurable	4,133	35,973	68,849	7,771,568.02
	Spatial Dup.	4,206	36,672	69,305	7,878,161.31
	Spatial with ILP reduction	4,834	41,485	76,407	9,553,048.27

Source: The Author

APPENDIX B — EVALUATION OF DIFFERENT PRIORITIES FOR THE MWPUETF

In this chapter, the results for the best configuration are presented while varying the priority of energy consumption, performance, and fault tolerance in the Mean Work Per Unit of Energy to Failure (MWPUETF) metric. The Equation (B.1) is used to prioritize a specific axis, where $a + b + c = 1$. Results are presented for the following priorities: 0.5, 0.6, 0.7, 0.8, and 0.9. Therefore, if a given axis has an exponent equal to 0.5, the other two axes will have 0.25 each, and so on.

$$MWPUETF = (\frac{core\ utilization}{failure\ rate})^a \times \frac{1}{(exec.\ time)^b \times (energy\ cons.)^c} \tag{B.1}$$

B.1 Priority: Energy Consumption

Tables B.1, B.2, B.3, B.4, and B.5 depict the change in the best configuration as we increase the weight of the energy consumption axis in the MWPUETF metric. The 2 and 4-issue are prioritized, as well as smaller buffer sizes and low overhead configurations. For instance, with a priority of 0.9 to energy, most benchmarks are executed in the unprotected 2-issue processor, and the others are executed configurations with small or intermediate buffer size in the 2 and 4-issue.

Table B.1: Best Configuration - Priority: Energy (0.5)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True								4-issue			
	32	False											
		True											
	64	False		4-issue					8-issue				
		True	8-issue										
	128	False					4-issue						
		True			2-issue	2-issue					2-issue	2-issue	

Source: The Author

Table B.2: Best Configuration - Priority: Energy (0.6)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True								4-issue			
	32	False											
		True											
	64	False		4-issue					8-issue				
		True	8-issue										
	128	False					4-issue						
		True			2-issue	2-issue					2-issue	2-issue	

Source: The Author

Table B.3: Best Configuration - Priority: Energy (0.7)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True				4-issue			4-issue	4-issue			
	32	False											
		True											
	64	False		4-issue									
		True	8-issue										
	128	False					4-issue						
		True			2-issue						2-issue	2-issue	

Source: The Author

Table B.4: Best Configuration - Priority: Energy (0.8)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False	2-issue				2-issue	2-issue	2-issue				
TempDup	16	False											4-issue
		True				4-issue				4-issue			
	32	False		4-issue									
		True											
	64	False											
		True									4-issue		
	128	False											
		True			2-issue							2-issue	

Source: The Author

Table B.5: Best Configuration - Priority: Energy (0.9)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False	2-issue	2-issue			2-issue	2-issue	2-issue	2-issue		2-issue	2-issue
TempDup	16	False											
		True				4-issue							
	32	False											
		True											
	64	False											
		True			2-issue						4-issue		
	128	False											
		True											

Source: The Author

B.2 Priority: Performance

Tables B.6, B.7, B.8, B.9, and B.10 depict the change in the best configuration as we increase the weight of the performance axis in the MWPUETF metric. The 4 and 8-issue are prioritized, as they deliver better performance than the 2-issue. The 8-issue does not greatly improve the performance when compared to the 4-issue, so even when prioritizing the axis of performance, it does not outweigh the extra energy consumption of the 8-issue core for most benchmarks.

Table B.6: Best Configuration - Priority: Performance (0.5)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True								4-issue			
	32	False											
		True											
	64	False		4-issue					8-issue				
		True	8-issue										
	128	False					4-issue						
		True			2-issue	2-issue					2-issue	2-issue	

Source: The Author

Table B.7: Best Configuration - Priority: Performance (0.6)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True								4-issue			
	32	False											
		True											
	64	False		4-issue					8-issue				
		True	8-issue										
	128	False					4-issue						
		True			2-issue	2-issue					2-issue	2-issue	

Source: The Author

Table B.8: Best Configuration - Priority: Performance (0.7)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True								4-issue			
	32	False											
		True											
	64	False		4-issue					8-issue				
		True	8-issue										
	128	False					4-issue						
		True			2-issue	2-issue					2-issue	2-issue	

Source: The Author

Table B.9: Best Configuration - Priority: Performance (0.8)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True			4-issue					4-issue			
	32	False											
		True											
	64	False		4-issue					8-issue				
		True	8-issue								4-issue		
	128	False					4-issue						
		True				2-issue						2-issue	

Source: The Author

Table B.10: Best Configuration - Priority: Performance (0.9)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True			4-issue	4-issue				4-issue			
	32	False											
		True											
	64	False		4-issue					8-issue				
		True	8-issue								4-issue		
	128	False					4-issue						
		True										2-issue	

Source: The Author

B.3 Priority: Fault Tolerance

Tables B.11, B.12, B.13, B.14, and B.15 depict the change in the best configuration as we increase the weight of the fault tolerance axis in the MWPUETF metric. Prioritizing the fault tolerance axis does not have a clear trend when its weight is increased because such behavior highly depends on the application behavior, and the number of instructions that each configuration is able to duplicate. That is, applications with low Instruction-Level Parallelism (ILP) are able to duplicate most instructions with a small issue-width, while applications with high ILP require larger issue-widths and buffer sizes.

Table B.11: Best Configuration - Priority: Fault tolerance (0.5)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True								4-issue			
	32	False											
		True											
	64	False		4-issue					8-issue				
		True	8-issue										
	128	False					4-issue						
		True			2-issue	2-issue					2-issue	2-issue	

Source: The Author

Table B.12: Best Configuration - Priority: Fault tolerance (0.6)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True								4-issue			
	32	False											
		True											
	64	False		4-issue					8-issue				
		True	8-issue										
	128	False					4-issue						
		True			2-issue	2-issue					2-issue	2-issue	

Source: The Author

Table B.13: Best Configuration - Priority: Fault tolerance (0.7)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True								4-issue			
	32	False											
		True											
	64	False	8-issue	4-issue					8-issue				
		True											
	128	False					4-issue						
		True			2-issue	2-issue					2-issue	2-issue	

Source: The Author

Table B.14: Best Configuration - Priority: Fault tolerance (0.8)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True								4-issue			
	32	False											
		True											
	64	False	8-issue	4-issue					8-issue				
		True											
	128	False					4-issue						
		True			2-issue	2-issue					2-issue	2-issue	

Source: The Author

Table B.15: Best Configuration - Priority: Fault tolerance (0.9)

	Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Unprotected		False											
TempDup	16	False						8-issue					4-issue
		True								4-issue			
	32	False											
		True											
	64	False	8-issue	4-issue					8-issue				
		True											
	128	False					4-issue						
		True			2-issue	2-issue					2-issue	2-issue	

Source: The Author

APPENDIX C — RESUMO EM PORTUGUÊS

C.1 Introdução

Ao se projetar um novo processador, o desempenho não é mais o único objetivo de otimização. Reduzir o consumo de energia também é essencial – enquanto a maior parte dos dispositivos embarcados depende fortemente de bateria, processadores de propósito geral (GPPs) são restringidos pelos limites da energia térmica de projeto (TDP – thermal design power). Além disso, devido à evolução da tecnologia, a taxa de falhas transientes vem aumentando nos processadores modernos, o que afeta a confiabilidade de sistemas tanto no espaço quanto no nível do mar.

No entanto, os projetos dos processadores atuais são voltados a no máximo dois desses eixos em decorrência da sua natureza conflitante (ou seja, otimizar um deles provavelmente acarretará um impacto negativo aos outros). Por exemplo, reduzir o consumo de energia provavelmente reduzirá a performance; melhorar a confiabilidade aumentará o consumo de energia e possivelmente reduzirá a performance; e melhorar a performance afetará o consumo de energia e possivelmente a confiabilidade.

Nesse cenário desafiador, este trabalho propõe duas versões de um processador adaptativo e polimórfico capaz de equilibrar desempenho, tolerância a falhas e consumo de energia em tempo de execução. Essa adaptação é feita através do desenvolvimento de técnicas de otimização que se aproveitam das unidades de execução ociosas, que poderão executar instruções replicadas (para tolerância a falhas), ter a sua tensão de alimentação cortada através de power gating (para consumo de energia) ou executar threads adicionais (para o desempenho). Também é possível reduzir artificialmente o paralelismo no nível das instruções (ILP) para liberar unidades de execução ao preço de desempenho, quando mais tolerância a falhas ou menor consumo de energia forem necessários.

Todas essas técnicas foram implementadas no processador VLIW ρ-VEX e utilizadas para adaptar a execução da aplicação com base nas características da aplicação. O processador tem um hardware especial para decidir de forma dinâmica qual técnica de otimização é a mais adequada para ser aplicada em determinado momento, de acordo com uma função de otimização que leva em conta os três eixos.

C.2 Técnicas propostas

A seguir, serão apresentadas as técnicas que foram implementadas nos eixos de tolerância a falhas, consumo de energia e performance.

C.2.1 Tolerância a falhas

C.2.1.1 Duplicação com rollback

A técnica base de todas as técnicas de tolerância a falhas se baseia em um mecanismo de duplicação de instruções com rollback, que reexecuta instruções com falhas para corrigir o erro. A granularidade da duplicação varia conforme a técnica que está sendo utilizada, mas todas exploram as unidades funcionais que não estão sendo usadas. Dessa forma, pipelanes dedicados à execução de instruções duplicadas não precisam ser criados, economizando área e dissipando menos potência. Checkers verificam os resultados das instruções duplicadas e os resultados corretos são commitados.

C.2.1.2 Duplicação baseada em fases

Nessa técnica, é feito o profiling da aplicação com o intuito de identificar suas fases, que são de baixo ou alto ILP. Com base nesse profiling, são criadas configurações, e durante a execução da aplicação, essas configurações são usadas para realizar a duplicação das instruções quando há pipelanes que não estão sendo utilizados. Aplicações que possuem um alto ILP não se beneficiam dessa técnica pois as fases não podem ser claramente exploradas para a duplicação das instruções.

C.2.1.3 Duplicação espacial - duplicação quando possível

Esse mecanismo realiza a duplicação de instruções sempre que há NOPs na palavra VLIW. Ou seja, a cada ciclo, os NOPs são substituídos por instruções duplicadas do mesmo bundle.

C.2.1.4 Duplicação temporal e espacial

Essa técnica leva a anterior um passo adiante, aumentando a flexibilidade para a duplicação das instruções. Instruções que não conseguem ser duplicadas em um mesmo bundle são armazenadas em um buffer que alocará essas instruções duplicadas em um ciclo posterior. Assim, mais instruções são duplicadas, aumentando a confiabilidade. Adicionalmente, a latência da memória é explorada para executar instruções que estão no buffer, enquanto o processador aguarda os dados da memória.

C.2.2 Consumo de energia

O Power Gating é aplicado nas unidades funcionais que não estão sendo utilizadas pela aplicação, para reduzir o consumo de energia. Durante a execução da aplicação, ela é avaliada, e as configurações de PG são salvas para serem usadas na próxima vez em que um determinado trecho de código for executado novamente. Dessa forma, o power gating pode ser aplicado sem qualquer profiling estático.

C.2.3 Performance

O ILP pode ser reduzido dinamicamente para criar artificialmente novos slots e aumentar a duplicação ou maximizar os ganhos do power gating. O controle é feito através de um hardware especializado que divide bundles para serem executados em dois ciclos quando um certo threshold for atingido (definido em tempo de design).

C.3 Processador adaptativo

O processador adaptativo é baseado na versão estática do processador ρ-VEX. Ele utiliza a duplicação espacial, o módulo de controle de ILP e pode ter o módulo de power gating habilitado ou desabilitado, para focar em confiabilidade ou consumo de energia.

C.4 Processador polimórfico

O processador polimórfico explora as técnicas anteriores para prover a reconfiguração do hardware de acordo com a aplicação que está sendo executada. Ele possui o mecanismo de duplicação temporal e espacial no eixo de tolerância a falhas, power gating para o consumo de energia e tem a capacidade de alterar a largura do issue em tempo de execução, possibilitando que até quatro aplicações sejam executadas em paralelo. Todos esses módulos são controlados por um módulo de decisão que leva em conta o trade-off entre os três eixos para escolher a melhor configuração para cada aplicação. Enquanto a aplicação é executada, diferentes configurações do processador são testadas para identificar dinamicamente qual é a configuração ideal para determinada aplicação. Assim, cada vez que um kernel da aplicação for executado, uma configuração diferente

será avaliada, e quando o melhor resultado for encontrado, a fase de testes será encerrada e a aplicação continuará a execução com a melhor configuração possível. Portanto, o processador polimórfico consegue se adaptar a qualquer aplicação após uma breve fase de aprendizado.

C.5 Injetor de falhas híbrido

No escopo desse trabalho também foi proposto um injetor de falhas híbrido, que explora dois níveis de simulação para acelerar a campanha de injeção de falhas, mantendo a precisão de uma injeção em gate-level. A troca entre os dois níveis de simulação é feita de forma automática e transparente ao usuário. A aplicação é executada em uma simulação comportamental até o momento da injeção da falha. Então, o contexto é trocado para a simulação gate-level e a falha é injetada. Após a injeção, a simulação troca o contexto novamente, e a aplicação segue a execução em nível comportamental, acelerando a campanha de injeção de falhas.

C.6 Metodologia

Para obter dados de potência e área, o Cadence Encounter RTL é usado para sintetizar os módulos de hardware em uma tecnologia de 65nm e o CACTI-P (LI et al., 2011) é usado para estimar os custos dos módulos de memória e buffers, que são: memória para as configurações do power gating, buffers da duplicação temporal, memória de configurações para o processador polimórfico, memórias cache e memória principal. Todas as memórias possuem ECC e todos os módulos adicionais são contabilizados em termos de área e potência. O conjunto de benchmarks é composto de 16 aplicações do WCET (GUSTAFSSON et al., 2010) e Powerstone (SCOTT et al., 1998). Simuladores foram desenvolvidos para emular o comportamento da versão dinâmica do ρ-VEX e para incorporar os mecanismos do processador adaptativo.

C.7 Resumo dos resultados

Para o processador adaptativo, múltiplas configurações são avaliadas, e quando a duplicação é aplicada juntamente com o mecanismo de power gating e controle de ILP, o

overhead médio é de 13% em performance, 37,2% em energia e 15,05% em área, e 0,31% de taxa de defeitos.

Considerando uma métrica que abrange os três eixos, o processador polimórfico se mostrou capaz de realizar a adaptação para qualquer aplicação, chegando em 94,88% do resultado de um oráculo, na média (de 86,01% até 99,99%). Em contrapartida, a melhor configuração estática consegue apenas 28,24% do resultado do oráculo. O overhead de área é de 26,82% e a taxa de instruções duplicadas varia de 9 até 100%, considerando todas as configurações testadas.

C.8 Conclusões

Neste trabalho foram propostas diversas técnicas que, juntas, foram exploradas para a implementação de um processador adaptativo e polimórfico que consegue se adaptar a qualquer aplicação de forma dinâmica. Uma métrica que leva em conta os eixos de tolerância a falhas, consumo de energia e performance é usada para realizar a escolha de qual é a melhor configuração para cada aplicação, após um breve período de aprendizagem.