ADVERTIMENT. La consulta d'aquesta tesi queda condicionada a l'acceptació de les següents condicions d'ús: La difusió d'aquesta tesi per mitjà del servei TDX (www.tesisenxarxa.net) ha estat autoritzada pels titulars dels drets de propietat intel·lectual únicament per a usos privats emmarcats en activitats d'investigació i docència. No s'autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d'un lloc aliè al servei TDX. No s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

ADVERTENCIA. La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del servicio TDR (www.tesisenred.net) ha sido autorizada por los titulares de los derechos de propiedad intelectual únicamente para usos privados enmarcados en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio TDR. No se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

WARNING. On having consulted this thesis you're accepting the following use conditions: Spreading this thesis by the TDX (www.tesisenxarxa.net) service has been authorized by the titular of the intellectual property rights only for private uses placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading and availability from a site foreign to the TDX service. Introducing its content in a window or frame foreign to the TDX service is not authorized (framing). This rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it's obliged to indicate the name of the author

UNIVERSITAT POLITÈCNICA DE CATALUNYA
Departament d'Arquitectura de Computadors

Low-Cost And Efficient Fault Detection And Diagnosis Schemes for Modern Cores

---

A thesis submitted in fulfillment of
the requirements for the degree of
DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC
by

Javier Sebastián Carretero Casado
Barcelona, 2015

Advisor: **Xavier Vera Rivera** (Intel Corporation)
Tutor: **Antonio González Colás** (UPC)

Mientras iba de tu mano hacia la montaña,
unos días eran fuego y otros eran llamas.
Dentro del espejo donde no me reflejaba,
la promesa que en la cima nos aguardaba.
Pero una vez allí las nubes no nos dejaban ver el suelo
y una sensación que tuve fue miedo.

El camino de bajada era más estrecho,
se podría decir una bajada a los infiernos.
Te pedí que me guiaras cuando estaba ciego,
la montaña fue quien respondió con eco,
un eco que reproducía exactamente mis lamentos,
los sueños que una vez tenía y ya no tengo.

San Juan de la Cruz - Los Planetas

All these saints that I move without
I lose without in vain
All these saints, they move without
They moved without again
Well, all these places will lose without
They lose without a name

St. Apollonia - Beirut

Abstract

Continuous improvements in transistor scaling together with microarchitectural advances have made possible the widespread adoption of high-performance processors across all market segments. However, the growing reliability threats induced by technology scaling and by the complexity of designs are challenging the production of cheap yet robust systems.

Soft error trends are haunting, especially for combinational logic, and parity and ECC codes are therefore becoming insufficient as combinational logic turns into the dominant source of soft errors. Furthermore, experts are warning about the need to also address intermittent and permanent faults during processor runtime, as increasing temperatures and device variations will accelerate inherent aging phenomena.

These challenges specially threaten the commodity segments, which impose requirements that existing fault tolerance mechanisms cannot offer. Current techniques based on redundant execution were devised in a time when high penalties were assumed for the sake of high reliability levels. Novel light-weight techniques are therefore needed to enable fault protection in the mass market segments.

The complexity of designs is making post-silicon validation extremely expensive. Validation costs exceed design costs, and the number of discovered bugs is growing, both during validation and once products hit the market. Fault localization and diagnosis are the biggest bottlenecks, magnified by huge detection latencies, limited internal observability, and costly server farms to generate test outputs.

This thesis explores two directions to address some of the critical challenges introduced by unreliable technologies and by the limitations of current validation approaches.

We first explore mechanisms for comprehensively detecting multiple sources of failures in modern processors during their lifetime (including transient, intermittent, permanent and also design bugs). Our solutions embrace a paradigm where fault tolerance is built based on exploiting high-level microarchitectural invariants that are reusable across designs, rather than relying on re-execution or ad-hoc block-

level protection. To do so, we decompose the basic functionalities of processors into high-level tasks and propose three novel runtime verification solutions that combined enable global error detection: a computation/register dataflow checker, a memory dataflow checker, and a control flow checker. The techniques use the concept of end-to-end signatures and allow designers to adjust the fault coverage to their needs, by trading-off area, power and performance. Our fault injection studies reveal that our methods provide high coverage levels while causing significantly lower performance, power and area costs than existing techniques.

Then, this thesis extends the applicability of the proposed error detection schemes to the validation phases. We present a fault localization and diagnosis solution for the memory dataflow by combining our error detection mechanism, a new low-cost logging mechanism and a diagnosis program. Selected internal activity is continuously traced and kept in a memory-resident log whose capacity can be expanded to suite validation needs. The solution can catch undiscovered bugs, reducing the dependence on simulation farms that compute golden outputs. Upon error detection, the diagnosis algorithm analyzes the log to automatically locate the bug, and also to determine its root cause. Our evaluations show that very high localization coverage and diagnosis accuracy can be obtained at very low performance and area costs. The net result is a simplification of current debugging practices, which are extremely manual, time consuming and cumbersome.

Altogether, the integrated solutions proposed in this thesis capacitate the industry to deliver more reliable and correct processors as technology evolves into more complex designs and more vulnerable transistors.

Acknowledgements

Esta tesis no hubiera visto la luz sin la ayuda y el apoyo constante de varias personas.

Quiero empezar estos agradecimientos con Xavier Vera: director de tesis, jefe y amigo. Xavi es quien me ha iniciado en el mundo de la investigación, me ha enseñado, guíado, aconsejado y quien me ha discutido ideas con una paciencia inacabable. De él he aprendido la importancia de desarrollar el pensamiento crítico, la autoconfianza y el espíritu de luchador. Gràcies!

Junto con Xavi, Jaume Abella ha sido como un segundo director. Gran parte de lo que sé se lo debo a él. Ojalá algún día tenga su capacidad de saber plantear las preguntas adecuadas, de saber esquivar y meter goles con argumentos, o de perseguir la simplicidad y la novedad en las soluciones. Aquesta tesi és teva també.

A Antonio González tengo que agradecerle la confianza que tuvo en mí, desde el día que me contrató en Intel, así como cuando me sugirió la posibilidad de realizar esta tesis. Gracias por estas oportunidades.

Mil gracias también a los compañeros de Intel y UPC por las risas, coñas, viajes y buenos momentos compartidos. En especial a los 'Rancis', a Matteo, Rakesh R., Demos, Juan F., Kyriakos, Gaurang U., Qiong, Pepe y a Pedro M., entre otros.

Fuera del laboratorio, quiero dar las gracias a mis amigos/as de Barcelona y de Barahona. Me habéis ayudado a desconectar de las dificultades de la tesis y del trabajo. Sois una parte importante de mi vida, y con vosotros he podido respirar aire fresco y ver las cosas con una perspectiva más amplia. Ya no hay más excusas para no veros más a menudo.

Para acabar, me gustaría dedicar unas palabras a mi familia. A todos ellos, gracias por hacerme sentir querido y tener fe en mí. En especial, a mis hermanos (Charo, María José y Patxi) por su apoyo constante. A mis cuñados y a los bichejos por hacer al clan aún más divertido. Y sobre todo a mis padres, a quien va dedicada esta tesis: nunca podré apreciar ni llegar al nivel de vuestra dedicación.

Contents

Abstract v

Acknowledgements vii

Contents ix

List of Figures xiii

List of Tables xvii

Glossary of Acronyms and Abbreviations xix

1 Introduction 1

1.1 Motivation: Reliability Challenges . . . 2

1.1.1 Impact of Transistor Scaling on Lifetime Reliability . . . 2

1.1.2 Growing Design Complexity and Validation Costs . . . 6

1.2 Problem Statement . . . 7

1.2.1 Lifetime Reliability Mechanisms for Multiple Sources of Failures 7

1.2.2 Overheads of Error Detection Solutions . . . 8

1.2.3 Tackling Observability and Reproducibility During Post-Silicon Validation . . . 9

1.2.4 System-Level Simulation for Error Discovery and Diagnosis . . 10

1.3 Thesis Approach . . . 11

1.4 Thesis Contributions . . . 12

1.5 Thesis Organization . . . 14

2 Background **17**

2.1 Economic Costs of Hardware Reliability . . . 17

2.2 Reliability Concepts and Metrics . . . 18

2.2.1 Basic Terminology and Classification of Errors . . . 18

2.2.2 Fault Tolerance Metrics 20
2.3 Hardware Failure Phenomena: How Electronics Fail 21
2.4 Aspects of Fault Tolerance 24
2.5 Validation and Debugging: Background 26
2.5.1 Pre-Silicon Validation 26
2.5.2 Post-Silicon Validation 26
2.5.3 Runtime Validation 27

3 Related Work **29**
3.1 Re-execution- Based Hardware Techniques 30
3.2 Error Coding Techniques 35
3.3 Circuit-Level Techniques 38
3.4 Software-Level and Hybrid Techniques 40
3.5 Industrial Validation Techniques 43

4 Evaluation Framework **47**
4.1 Benchmarks, Tools and Simulators 47
4.1.1 Benchmarks 47
4.1.2 Timing Simulator 48
4.1.3 Fault Coverage Evaluation Methodology 51
4.1.4 Area, Power and Delay Evaluation Methodology 54
4.2 RAS Features in the Baseline Processor 58

5 Register Dataflow Validation **61**
5.1 Introduction 61
5.2 Register Dataflow Failures 62
5.3 End-to-End Dataflow Validation 64
5.3.1 Signature-Based Protection: General Idea 64
5.3.2 Failure Recovery 68
5.3.3 Microarchitectural Changes 69
5.4 End-to-End Register Value and Dataflow Validation 70
5.4.1 Implementing End-to-End Residue Checking 70
5.4.2 Integrating Signatures with Residues 73
5.4.3 Microarchitectural Changes 75
5.4.4 Examples 77
5.5 Signature Generation Policies 80
5.5.1 Round-Robin Policies 81

5.5.2 Minimum In-Flight Use Policy . . . 83
5.5.3 Physical Register Policy . . . 83
5.5.4 Static Policy . . . 84
5.5.5 Enhanced Static Policy . . . 86
5.6 Evaluation . . . 90
5.6.1 Coverage Results . . . 91
5.6.2 Overheads . . . 95
5.7 Related Work . . . 101
5.8 Conclusions . . . 105

6 Control Flow Recovery Validation 107
6.1 Introduction . . . 107
6.2 Control Flow Recovery in Modern OoO Processors: Overview . . . 108
6.3 Control Flow Recovery Failures . . . 110
6.4 End-To-End Validation of RAT State Recovery . . . 113
6.4.1 RAT State Signature Tracking . . . 113
6.4.2 RAT State Signature Validation . . . 118
6.4.3 Microarchitectural Changes . . . 119
6.5 End-To-End Validation of Instruction Squashing . . . 120
6.5.1 Bogus Region Tracking . . . 120
6.5.2 Bogus Region Validation . . . 122
6.5.3 Microarchitectural Changes . . . 124
6.6 Evaluation . . . 124
6.6.1 Coverage Results . . . 124
6.6.2 Overheads . . . 128
6.7 Conclusions . . . 132

7 Memory Flow Validation 135
7.1 Introduction . . . 135
7.2 Load-Store Queue: Overview . . . 136
7.3 Load-Store Queue Failures . . . 138
7.4 LSQ Memory Ordering Tracking and Validation: General Idea . . . 140
7.4.1 Microarchitectural Changes . . . 141
7.4.2 LSQ Memory Ordering Tracking . . . 142
7.4.3 LSQ Memory Order Validation . . . 143
7.4.4 Failure Recovery . . . 145
7.5 Design #1: *MOVT* Access at Execute . . . 145

7.6 Design #2: Minimal *prodID* Acquisition 148
7.7 Design #3: *MOVT* Access at Allocate 150
7.8 Evaluation 153
7.8.1 Fault Coverage Methodology 153
7.8.2 Area Overheads 154
7.8.3 Evaluation of Design #1: *MOVT* Access at Execute 155
7.8.4 Evaluation of Design #2: Minimal *prodID* Acquisition 158
7.8.5 Evaluation of Design #3: *MOVT* Access at Allocate 159
7.9 Conclusions 163

8 Automated Fault Localization and Diagnosis 165
8.1 Introduction 165
8.2 Automated Fault Localization and Diagnosis: Proposed System Overview 166
8.3 Event Generation 167
8.4 Diagnosis Algorithm 170
8.5 Logging System Implementation 176
8.5.1 Microarchitectural Changes 177
8.5.2 System-Level Interaction 180
8.6 Evaluation 182
8.6.1 Diagnosis Coverage Results 182
8.6.2 Overheads 186
8.7 Related Work 188
8.8 Conclusions 191

9 Conclusions 193
9.1 Publications 194
9.2 Open Research Directions 195

Appendix:

A Baseline Processor Microarchitecture 197
A.1 Processor Frontend 197
A.2 Processor Backend 199

Bibliography 209

List of Figures

1.1 SER trends for SRAM cells, latches and combinational logic 3
1.2 Chip-level SER trends for caches and logic 4
1.3 Wear-out failure phenomena FIT contribution breakdown 6

2.1 Classification of faults effects 19
2.2 Vendors fault tolerance metrics 21
2.3 Particle strike causing current disturbance 22
2.4 Physical wear-out phenomena, open and short creation 23
2.5 Validation domains and characteristics 25

4.1 Global structure of our evaluation framework 48
4.2 Baseline processor microarchitecture 50

5.1 Register signature assignment among dependent instructions: an example 64
5.2 End-to-end signature checking: extensions in the core dataflow 65
5.3 End-to-end signature checking: extensions in the backend logic 66
5.4 Concurrent error detection with residue codes 71
5.5 End-to-end residue checking: extensions in the backend logic 72
5.6 Combined end-to-end signature and residue checking scheme: extensions in the backend logic 74
5.7 End-to-end signatures and residues operation: fault-free scenario example 77
5.8 End-to-end signatures and residues operation: *Selection of wrong inputs* example 79
5.9 End-to-end signatures and residues operation: *Wrong Register File Access* example 80
5.10 Distribution of usage for the different logical registers across all benchmarks 85

5.11 Signature masking enhancement to boost coverage for *'Selection of wrong inputs'* case: extensions in the backend logic 88
5.12 Coverage results for all policies and error scenarios for 2-bit signatures 92
5.13 Coverage results for all policies and error scenarios for 3-bit signatures 94

6.1 Failure scenarios related to RAT state history reconstruction 111
6.2 Failure scenarios related to identification of control-flow dependent instructions 112
6.3 RAT state signature generation: extensions in the rename logic 114
6.4 f and f^{-1} blocks implementation 117
6.5 RAT state signature reconstruction: extensions in the commit logic . 118
6.6 BCT mechanism: extensions for bogus region tracking and validation . 121
6.7 BCT mechanism: extensions in the commit logic 123
6.8 Coverage for end-to-end RAT state signatures 125
6.9 Breakdown of number of younger resolved bogus regions for each mispredicted branch 126
6.10 Coverage for identification of control-flow dependent instructions (1 to 4 BCT entries) 127

7.1 A typical LSQ configuration 136
7.2 Failure scenarios related to LSQ operation 138
7.3 *MOVT* hardware template 141
7.4 Memory ordering tracking and validation: an example 144
7.5 *MOVT* hardware for design #1: prodID *acquisition* at execute time . 146
7.6 *MOVT* hardware for design #2: minimal prodID *acquisition* 149
7.7 *MOVT* hardware for design #3: prodID *acquisition* at allocate time . 151
7.8 Coverage and slowdown for different fully-associative *MOVT*s based on *prodID* acquisition at execute time 156
7.9 Detailed evaluation of a 16 entries, 4-way *MOVT* based on *prodID* acquisition at execute time 157
7.10 Coverage and slowdown for different *minimalist MOVT*s configurations 159
7.11 Flushed loads for different *minimalist MOVT*s configurations 160
7.12 Coverage and slowdown for different *MOVT*s based on prodID *acquisition* at allocate time 161
7.13 Breakdown of *speculative prodID* comparisons for a 32-entries fully-associative *MOVT* based on prodID *acquisition* at allocate time . . . 162

7.14 Breakdown of pipeline flushes for different *MOVT*s based on prodID *acquisition* at allocate time 162

8.1 Event driving latches: extensions in the processor 167
8.2 Diagnosis algorithm showing failure type determination: high-level code 173
8.3 Log of a LSQ failure: an example 174
8.4 Accumulated diagnosis coverage versus log size 175
8.5 Address hashing undistinguishable failure scenarios: an example . . . 176
8.6 Activity logging mechanism: hardware design and integration 178
8.7 Log buffer: hardware organization 179
8.8 Breakdown of number of LSQ log events generated per cycle 183
8.9 Diagnosis coverage and dropped events for different '*LOG buffer*' configurations. '`Xwr,Yrows`' stands for number of writable events per cycle, total number of '*event rows*' 184
8.10 Diagnosis coverage for a '`5wr,12rows` *LOG buffer*' configuration . . . 185
8.11 Slowdown induced by a '`5wr,12rows` *LOG buffer*' configuration . . . 187

List of Tables

3.1 Comparison of hardware-level global re-execution techniques 32
3.2 Comparison of error coding techniques 36
3.3 Comparison of circuit-level techniques 40
3.4 Comparison of software and hybrid techniques 41

4.1 Benchmarks used to evaluate our solutions 49
4.2 Simulator configuration 52

5.1 Register signature mismatches corresponding to real register dataflow errors 68
5.2 Values, residues, signatures and combined residues-signatures for fault-free example 78
5.3 Mask table for a processor with two execution ports (*P0* and *P1*), two bypass levels (*BL0* and *BL1*) and the write-back port (*WB*) 89
5.4 Values of the masks set up at every bypass level and execution port . 89
5.5 Area and power overheads for the different signature generation policies when end-to-end residue is absent. 97
5.6 Area and power overheads for the different signature generation policies when end-to-end residue is implemented. 99
5.7 Overheads summary of implementing end-to-end signature checking and end-to-end residue checking. 100
5.8 Comparative table of techniques that detect errors in the register dataflow 101
5.9 Blocks and logic protection for register dataflow validation techniques 103

6.1 Commit time assertion checks for instruction squashing verification . 123
6.2 Area and power overheads. `n`*b* SGN stands for `n`-bits RAT state signatures. 129
6.3 Area and power overheads. `n`*e* `BCT` stands for `n` `BCT` entries. 130

6.4 Area and power overheads. nb SGN stands for n-bits RAT state signatures and ne BCT for n BCT entries. 131

7.1 Protocol when loads hit the *MOVT* at commit time (Design #1) . . 147
7.2 Protocol when loads miss the *MOVT* at commit time (Design #1) . . 147
7.3 Protocol when loads hit the *MOVT* at commit time (Design #2) . . 149
7.4 Protocol when loads miss the *MOVT* at commit time (Design #2) . . 150
7.5 Protocol when loads hit the *MOVT* at commit time (Design #3) . . 152
7.6 Protocol when loads miss the *MOVT* at commit time (Design #3) . . 152
7.7 Area overhead w.r.t. the LSQ, for different *MOVT* sizes 155
7.8 Coverage, slowdown and area cost for different *MOVT* configurations 155

8.1 Diagnosable LSQ failure scenarios: descriptions and required analysis window size 172
8.2 Area, peak dynamic power and cycle time overhead for different '*LOGGING systems*' 188
8.3 Comparative table for fault localization, logging and diagnosis techniques 190

Glossary of Acronyms and Abbreviations

ACE	Architecturally Correct Execution.
AGEN	Address Generation.
AGU	Address Generation Unit.
ALLOC	Allocation.
ALU	Arithmetic and Logic Unit.
AR-SMT	Active and Redundant Simultaneous Multi-Threading.
ATE	Automatic Test Equipment.
ATPG	Automatic Test Pattern Generation.
AVF	Architecture Vulnerability Factor.
BCS	Beta Core Solution.
BCT	Bogus Check Table.
BER	Backward Error Recovery.
BICS	Built-In Current Sensor.
BIST	Built-In Self Test.
BTB	Branch Target Buffer.
CAM	Content Addressable Memory.
CFCSS	Control Flow Checking by Software Signatures.
CFG	Control Flow Graph.
ch RAT	Checkpoint RAT.
CMOS	Complementary Metal-Oxide Semiconductor.
CMP	Chip Multiprocessor.
CPU	Central Processing Unit.
CRAFT	CompileR Assisted Fault Tolerance.
CRC	Cyclic Redundancy Code.
CRT	Chip-level Redundant Threading.
CRTR	Chip-level Redundant Threading with Recovery.
CSM	Continuous Signature Monitoring.
D$	Data Cache.
DDFV	Dynamic DataFlow Verification.
DDR3 SDRAM	Double Data Rate type three Synchronous Dynamic Random-Access Memories.
DEC-TED	Double Error Correction Triple Error Detection.

DFCM Differential Finite Context Method.
DFG Data Flow Graph.
DFT Design for Testability.
DIVA Dynamic Implementation Verification Architecture.
DMR Dual Modular Redundancy.
DRAM Dynamic Random-Access Memory.
DRR Double Round Robin signature generation policy.
DUE Detected Unrecoverable Error.
DVFS Dynamic Voltage and Frequency Scaling.
ECC Error Correcting Code.
EDDI Error Detection by Duplicated Instructions.
EM Electromigration.
FER Forward Error Recovery.
FIFO First In First Out.
FIT Failure In Time.
FL Free List.
FO4 Fan-out of 4.
FP Floating Point.
FRITS Functional Random Instruction Testing at Speed.
FWD Forwarding.
GDXC Generic Debug eXternal Connection.
HCI Hot Carrier Injection.
HPC High-Performance Computing.
I$ Instruction Cache.
IC Integrated Circuit.
IFRA Instruction Footprint Recording and Analysis.
INT Integer.
I/O Input/Output.
IQ Issue Queue.
ISA Instruction Set Architecture.
JEU Jump Execution Unit.
L2$ Second Level Cache.
LD Load.
LDEXEC Load Execution.
LdQ Load Queue.
LEA Load Effective Address.
LLC Last Level Cache.
LRR Logical Round Robin signature generation policy.
LRU Least Recently Used.
LSQ Load/Store Queue.
MCA Machine Check Architecture.
MIN Minimum signature generation policy.
MOB Memory Order Buffer.
MOD Modulo signature generation policy.

MOVT Memory Order Validation Table.
MRR Minimum Round Robin signature generation policy.
MSHR Miss Status Holding Register.
MTBF Mean Time Between Failures.
MTTF Mean Time To Failure.
MTTR Mean Time To Repair.
MUX Multiplexor.
NBTI Negative Bias Temperature Instability.
NMOS N-type Metal Oxide Semiconductor.
OS Operating System.
PBTI Positive Bias Temperature Instability.
PC Program Counter.
pdst Physical Register Destination.
PHT Pattern History Table.
PLA Programmable Logic Arrays.
PMOS P-type Metal Oxide Semiconductor.
PRF Physical Register File.
PSA Path Signature Analysis.
PSMI Periodic State Management Interrupt.
QRR Quad Round Robin signature generation policy.
RAN Random signature generation policy.
RAS Reliability-Availability-Serviceability.
RAT Register Alias Table.
RF Register File.
RMT Redundant Multi Threading.
RNA Register Name Authentication.
ROB Re-Order Buffer.
RR Round Robing signature generation policy.
RTL Register Transfer Language.
SBU Single Bit Upset.
SDC Silent Data Corruption.
SEC-DED Single Error Correction Double Error Detection.
SelR Selective Replication.
SER Soft Error Rate.
SEU Single Event Upset.
SGN Signature.
SHREC SHared REsource Checker.
SIMD Single-Instruction Multiple Data.
SIS Signatured Instruction Streams.
SlicK Slice-Based Locality Exploitation for Efficient Redundant Multithreading.
SMT Simultaneous Multi Threading.
SpecIV Speculative Instruction Validation.
SRAM Static Random-Access Memory.
SRMT Software-based Redundant Multithreading.

SRT Simultaneously and Redundantly Threading.
SRTR Simultaneously and Redundantly Threaded with Recovery.
ST Store.
StQ Store Queue.
SW Software.
SWIFT Software Implemented Fault Tolerance.
TAC Timestamp-based Assertion Checking.
TBFD Trace-Based Fault Diagnosis.
TDDB Time Dependent Dielectric Breakdown.
TDP Thermal Design Power.
TLB Translation Lookaside Buffer.
TMR Triple Modular Redundancy.
TRUMP Triple Redundancy Multiplication Protection.
TTF Time To Failure.
TVF Time Vulnerability Factor.
VLIW Very Long Instruction Word.
XOR Exclusive Or.

CHAPTER 1

INTRODUCTION

Historically, fault tolerant designs have been applied to niche safety-critical and mission-critical segments to provide high reliability levels against hardware faults.

However, the increasing transistor miniaturization and subsequent supply voltage reductions, together with growing design complexities are amplifying the susceptibility of all computing systems to runtime errors: reliability is becoming a concern for products ranging from mobile devices up to mainstream computers.

Depending on the target market segments, processors are designed considering certain error rate specifications. However, whereas these specifications stay constant across product generations, the inherent error rates due to transient, permanent and design errors do increase. Meeting error rate specifications requires trading off performance, power or cost to implement fault tolerance techniques or to improve the validation phases.

Traditional fault tolerance solutions based on re-execution were devised in a time when high performance and power penalties were assumed for the sake of high reliability levels. However, commodity segments are extremely customer-sensitive and impose requirements that existing approaches cannot offer. Resorting to error detection-correction codes to protect memory arrays in commodity segments will soon not suffice, as combinational logic turns into the dominant source of failures. In the absence of new fault tolerance solutions, traditional online error detection methods would counteract the benefits of technology scaling and would offset the actual growth of the microprocessor industry.

At the same time, the increasing design complexity together with shortening time-to-market schedules are imposing important challenges in guaranteeing that processors are error-free before shipment. The costs of post-silicon validation currently

overpass the costs of the design phases, and the number of bugs found in the validation phases and once products hit the market is projected to rapidly increase.

In this thesis we focus on the error detection, error localization and diagnosis aspects of fault tolerance. Error detection is a pre-requisite to support other aspects of fault tolerance, whereas bug localization and debugging dominate validation efforts.

This thesis explores two directions to address some of the critical challenges introduced by unreliable technologies and by the limitations of current validation approaches. We first explore low-cost effective solutions to detect multiple sources of failures in commodity processors during their lifetime. Then, we explore post-silicon approaches that target the problems of bug detection, localization and diagnosis by relying on the features of our error detection mechanisms.

This chapter is organized as follows. Section 1.1 provides a detailed motivation for this work by presenting the main reliability challenges and trends targeted in this thesis. In Section 1.2 we state the research problems and the main objectives targeted in this thesis. Section 1.3 describes the new approaches that thesis proposes to effectively address the research objectives. Section 1.4 enumerates the main contributions of our work, and finally we describe the organization of this document in Section 1.5.

1.1 Motivation: Reliability Challenges

There are several reliability challenges brought by technology scaling and by the complexity of designs that have driven the research of this thesis. We describe them in the following two sections.

1.1.1 Impact of Transistor Scaling on Lifetime Reliability

The everlasting transistor miniaturization is radicalizing the error rates caused by particle strikes, also known as soft errors [129]. Furthermore, other sources of failure that were considered as a manufacturing problem in the past, are now emerging as a threat to processor lifetime reliability. We describe for these failure mechanisms the error rate trends introduced by technology scaling.

Increasing Cell and Control Logic Susceptibility to Soft Errors

Prediction of soft errors scaling trends for CMOS are haunting. On one hand, as SRAM cells shrink their soft error rate (SER) is expected to decrease due to a reduction in the susceptible area. On the other hand, SRAM cells will operate with

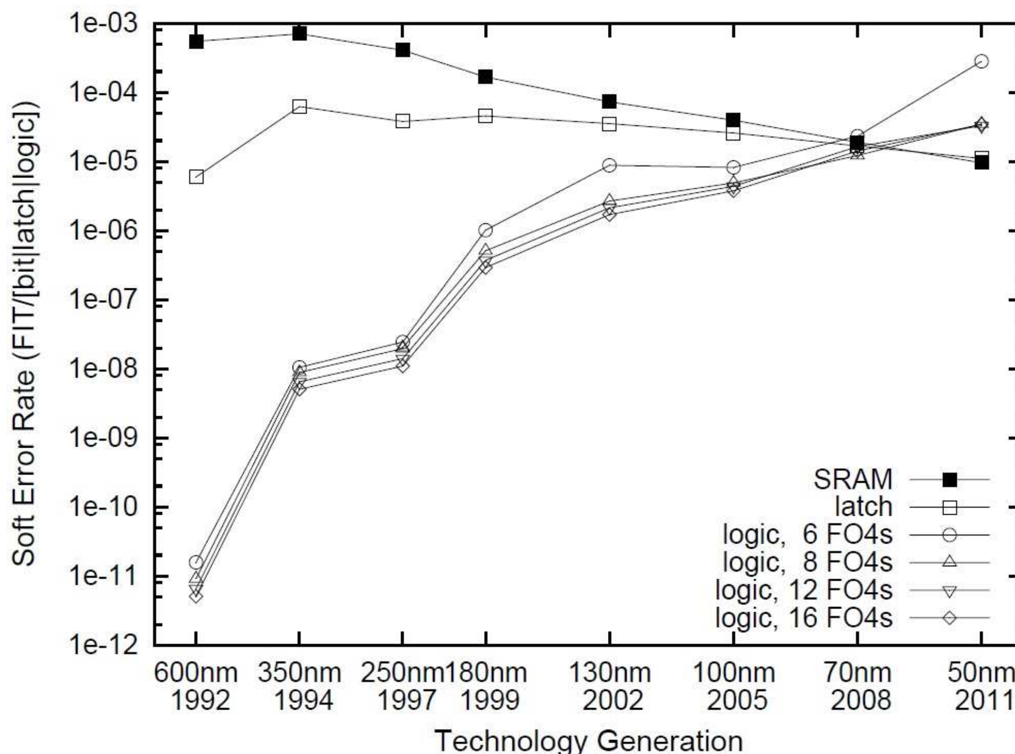


Fig. 1.1: SER trends for SRAM cells, latches and combinational logic [175]

lower charges and the Q_{crit} will decrease, amplifying the range of harmful particles. The former factor has dominated and offset the latter one in the past. However, this fact no longer seems to persists for SRAM cells. Alan Wood et al. [51] have reported that Oracle's technology scaling from 180nm to 65nm in the past caused a significant reduction in the SER error rates per bit, but from 40nm onwards to 28nm there was a reversal of this trend. Intel and Oracle [51, 133] indicated that the SER/bit for SRAM cells was slightly reducing after 45nm, and becoming almost constant.

The vulnerability for control logic has been traditionally lower than for SRAM cells or latches due to logical, electrical and timing masking effects [174]. However, Shivakumar et al. [175] estimated that the SER from logic would rise exponentially. Their study indicated that the effect of latching-window masking would be reduced drastically as transistors would shrink in size or frequency would increase (or pipe stages would decrease in length) [174, 175].

Figure 1.1 shows the soft error rate estimations for SRAM cells, latches and combinational logic for different pipeline depths and technologies[1]. It can be observed that the error rate trends for combinational logic was projected to increase at an

[1]Error rates are expressed in FITs, that stands for Failures In Time. One FIT corresponds to one failure in one billion (10^9) hours.

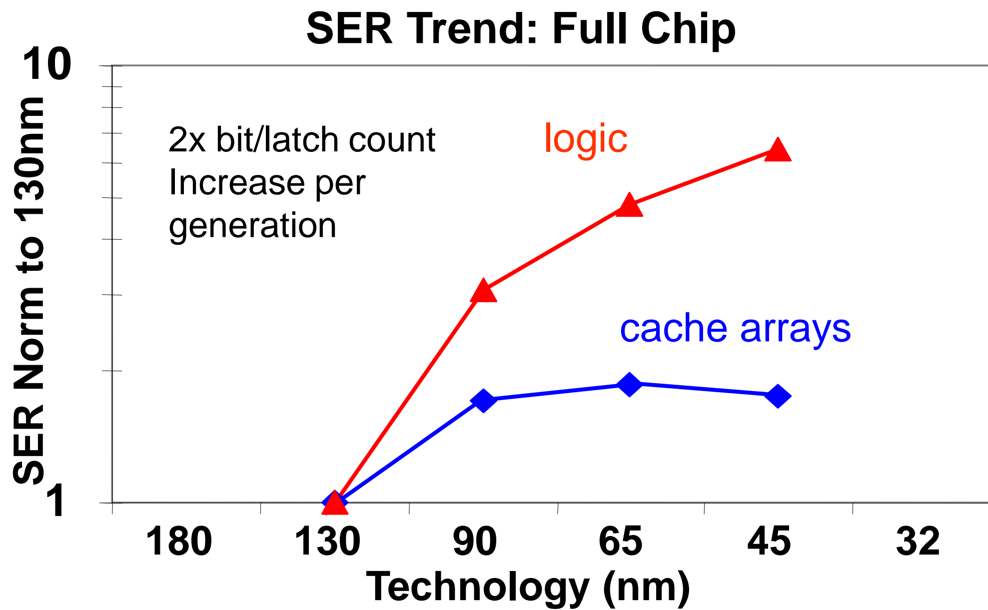


Fig. 1.2: Chip-level SER trends for caches and logic [133]

exponential rate for all pipeline depths. Also, from 50nm onwards the SER of a 6/8/12/16 FO4 logic and the SER per latch were projected to be higher than the SER per SRAM cell. The study also estimated that logic would become the dominant source of soft errors and the SER contribution of unprotected SRAM cells would stay relatively constant (in absolute terms) [175].

This SER trend for combinational logic has been confirmed to be true nowadays [15]. In 2014, Intel has reported [133] that the contribution of caches to the chip SER is becoming flat, while at the same time the SER for chip logic continues to increase because the SER per latch is not decreasing fast enough with respect to Moore's Law (Figure 1.2).

Moreover, Wood has observed for 40nm and 28nm technologies that the SER of combinational logic approximately doubles when dropping voltage from 1.25V to 0.7V, and doubles again when reducing from 0.7V to 0.5V [51]. Therefore, designs will be limited in core-count and performance due to voltage limitations, unless efficient fault tolerance techniques (specially for logic) are found.

These results are game-changing: solutions like ECC or parity to reduce the SRAM SER will quickly become insufficient as logic becomes the dominant source of soft errors. Therefore, new efficient methods for protecting combinational logic are becoming essential to construct reliable systems.

New Sources of Failures Affecting Lifetime Reliability

Wear-out vulnerability, permanent faults and variations have traditionally been exclusively dealt by circuit and process engineers, because it has been considered as pure manufacturing problems, and not problems to be handled during runtime (i.e. not lifetime reliability problems). However, supply voltages are not scaling accordingly to transistor scaling[2], resulting in increasing power densities, which at the same time is accelerating the problems of aging phenomena[3] and affecting processor lifetime reliability at a higher pace for new technologies.

Srinivasan et. al [191] have determined that the failure rate of a 65nm POWER4-like processor is 300% higher than the 180nm version of the processor. Time-dependent dielectric breakdown (TDDB) and electromigration (EM) represent the most damaging failure mechanisms, as Figure 1.3 shows. Failure rates caused by these phenomena will become more frequent and will radicalize due to their exponential dependence on temperature and because of decreasing interconnect dimensions.

With shrinking geometries, interconnects and transistors are becoming more vulnerable to the impact of variations introducing during the fabrication process. The thickness of the layers varies over the die area and as a consequence wear-out phenomena like electromigration will produce more frequent opens in the narrower portions, or shorts between neighboring or crossing conductors [42]. High-frequency circuits with minimal frequency guardband will be more prone to suffer from delay faults because of increased resistance induced by wearout, or because of the static variations of transistors parameters [23]. Similarly, bridging faults will appear due to short-circuit scenarios caused by electromigration.

Current testing approaches based on screening out processor infant mortality through temperature and voltage induce aging are therefore becoming obsolete and inadequate [23, 204]. As a consequence, an increasing number of faults due to weaker, variable transistors or due to latent manufacturing defects will manifest once the processor has been shipped and will cause failures before the target lifetime.

These facts call for new runtime solutions that are able to expand their error detection capabilities beyond soft errors, as new failure phenomena are becoming a problem for lifetime reliability.

---

[2]Due to cell stability issues, leakage power hazards, etc.

[3]Aging phenomena includes electromigration, stress migration, gate oxide or time dependent dielectric breakdown, thermal cycling [192], negative-bias temperature instability [6], among others.

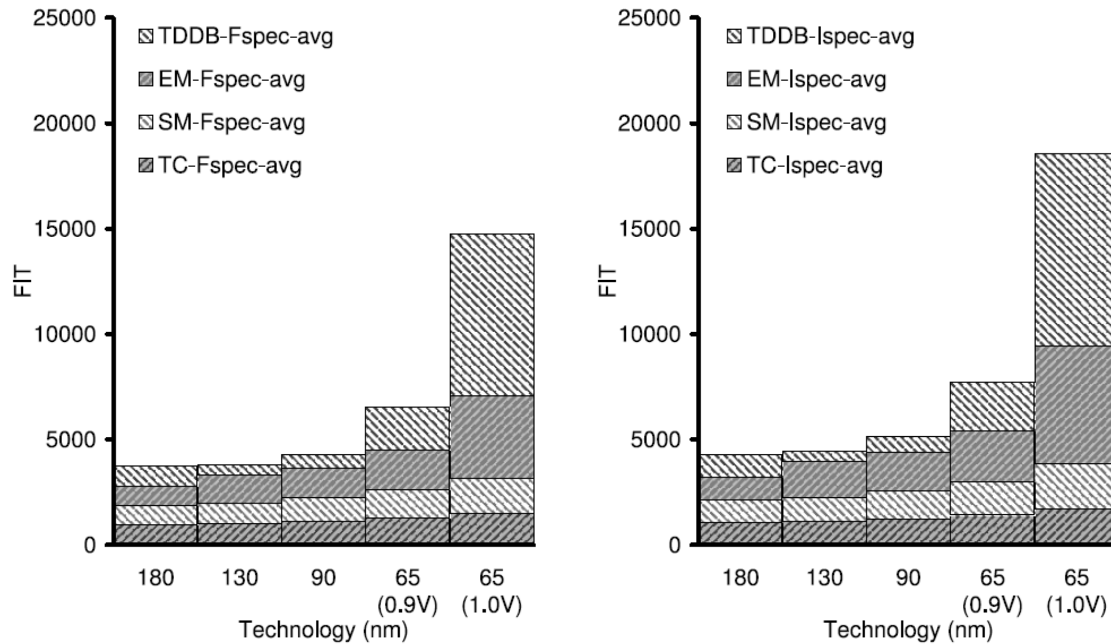


Fig. 1.3: Wear-out failure phenomena FIT contribution breakdown [192]

1.1.2 Growing Design Complexity and Validation Costs

Technology scaling has enabled performance breakthroughs through increasing transistor densities. On the other hand, a system FIT rate is rapidly increasing due to Moore's Law exponential pace: there is a higher probability that at least one of them suffers from faults during their lifetime or during fabrication. Therefore, the cost and complexity to keep current reliability levels for future technology where billions of devices are guaranteed to work during several years with a low failure rate is going to be huge.

However, growing transistor density is also materializing as an increasing processor design complexity, which directly puts a tremendous pressure in the processor validation phases. This growing complexity has fueled the importance of post-Silicon validation and debug phases during the production cycle of a processor [87, 121]. The validation phases currently overpass in cost the design phases. Around 35% of the product cost are spent on them, and it has been reported [148] that microprocessor companies staff their teams with three verification engineers per designer. At the same time, Intel has reported that the number of bugs found in the validation phases is increasing at a 3x to 4x rate for each generation, and this trend is proportional to the number of lines of structural RTL [88, 201]. The growing design complexity and the shrinking timelines for product delivery are aggravating these facts.

Even though post-silicon validation can leverage real silicon speeds to achieve high

coverage for subtle component interactions, it poses problems to error detection and diagnosis.

Sometimes bugs elude the validation phases and end up in the market, potentially causing massive financial and reputation impacts. The number of escaped bugs is increasing at a high pace: as an example, for Core 2 Duo designs researchers have reported a discovery rate that is 3 times larger than that of the Pentium 4 [43]. Under this scenario, the number of bugs debbugged in the validation phases is projected to rapidly increase, as well as the speed in which they are discovered once products hit the market (affecting millions of purchases).

These facts are calling for research advances in novel techniques and tools to improve the post-silicon validation phases, as well as in runtime verification approaches that provide processor lifetime correctness under undiscovered bugs. According to the ITRS [80, 81, 82], "without major breakthroughs, verification will be a non-scalable, show-stopping barrier for further progress in the semiconductor industry".

1.2 Problem Statement

The described challenges brought by the increasing vulnerability of silicon technologies and by the inefficiency of existing post-silicon validation methods, introduce several problems that we address in this thesis. In the following subsections we discuss them, we critically analyze the short-comings of some existing work, and we state the high-level research objectives that this thesis addresses to alleviate these problems.

1.2.1 Lifetime Reliability Mechanisms for Multiple Sources of Failures

Reliability trends show that multiple wear-out and permanent sources of failure are emerging as important contributors to microprocessors failure rates, rendering soft errors not the only reliability concern to be taken care of during product lifetime. At the same time, design complexity is causing an increase in design bugs eluding the post-silicon validation phases and impacting processor lifetime reliability.

As it will be thoroughly analyzed in Chapter 3 (Related Work), most state-of-art error detection solutions are designed for a specific error type, or for a few of them. For pure hardware reexecution-based techniques (Section 3.1), permanent faults cannot be targeted by solutions relying on temporal redundancy [64, 93, 143, 157, 162, 183, 196, 205, 207], whereas design bugs cannot be detected by solutions based on spatial (and design) redundancy [63, 127, 182, 197, 198]. Software-implemented redundant execution approaches (Section 3.4) also fail to detect multiple sources of failures

for the same reasons: they can either detect soft errors [138, 158, 160] or cannot comprehensively detect design bugs [34, 211]. Circuit-level techniques (Section 3.3) are limited to soft error mitigation [61], soft error detection detection [161, 202], or cannot detect permanent fault or design bugs in a cost-effective manner [47, 203]. On the other hand, built-in self-test circuits [2] cannot detect soft errors. Traditional error coding techniques like parity, ECC or CRC (Section 3.2), can detect soft and hard errors but just target data protection [70, 90, 210] and not combinational logic (an important contributor to processors failure rates).

Therefore, one major goal of this thesis is to:

- Explore and evaluate novel on-line mechanisms for comprehensively detecting multiple sources of failures in modern microprocessor cores during their lifetime (including transient, intermittent, permanent faults and design bugs). We look for unified mechanisms that can deal with all these sources of failure at the same time.

1.2.2 Overheads of Error Detection Solutions

The radical increase in raw error rates will pervade and threaten all commodity market segments. These segments impose challenging requirements to fault tolerance mechanisms that existing ones do not offer. Most of the error detection mechanism were devised for high-end segments where extreme reliability levels were targeted, in spite of severely weighing down global performance. However, reliability is not a primary design goal in commodity systems and some amount of fault coverage can be traded-off as long as processor performance, power and area are not severely impacted by runtime error detection solutions.

As it is described in Chapter 3, state-of-art error detection solutions are generally not suitable from a performance, power or area perspective when dealing with multiple sources of failures. Reexecution-based techniques covering soft and hard errors [63, 127, 198] suffer extreme power and power performance overheads because they redo at every microarchitectural block all the state and internal activity that constitute a computation. Reexecution-based techniques exploiting loose synchronization [182] or ineffectual instruction removal [197] to minimize performance slowdowns, still incur high power overheads and sacrifice a hardware thread context from another core to execute redundant computations. Advanced solutions exploiting both spatial redundancy and design heterogeneity [10] protect against soft errors, hard errors and design bugs. However, their power overheads and area costs are not affordable.

Software-implemented redundant execution approaches targeting soft and hard errors [34, 211] suffer from the same performance and power problems, even though

they require minimal area overheads. Compiler support has also been exploited by hybrid software-hardware solutions to avoid re-execution and to compute the expected microarchitectural activity to be observed during an error-free execution [131, 171, 219]. However, these techniques can only detect failures for the fetch and decode logic, and require extending the processor instruction set. Finally, error coding techniques implemented as self-checking circuits [12, 17, 136, 152] can detect soft errors, hard errors and design bugs with tolerable power and area overheads while causing no slowdown, though they are designed to detect errors in data and functional units.

Globally, existing solutions based on re-execution cannot strategically protect selected critical blocks or functionalities in a cost-effective and targeted way: they are global all-or-nothing approaches. Furthermore, these solutions do not offer flexibility to processor designers who may prefer modulating error coverage and power, performance and area overheads.

Hence, this thesis also aims at:

- Satisfying the needs for efficient reliability solutions with minimal costs in performance, power and area, while at the same time reaching similar reliability levels of traditional defect tolerance techniques.
- Exploring alternatives to reexecution-based techniques that can provide a more flexible trade-off between coverage and overheads, and that are also designed to be more modular for targeting specific blocks or functionalities.

1.2.3 Tackling Observability and Reproducibility During Post-Silicon Validation

The increasing design complexity and transistor integration is posing critical problems to error detection, localization and diagnosis during the post-silicon validation phases.

Processor are like black boxes where observing internal state or activity is extremely difficult. Common techniques like scan chains [2], hold-scan flip-flops [94] and cycle breakpoints [18] allow high-speed state inspection at a given execution moment. However, these techniques are prone-to error and require long iterative non-automated trial-and-error processes to hunt down the moment when the fault is exercised (as their use is extremely dependent on the experience of validators). Modern solutions based on on-chip embedded trace buffers [1, 103, 220] can continuously sample the internal state for a given time period, by storing traced data into dedicated memory. They are however limited by the limited capacity of on-chip storage buffers and the pin I/O bandwidth to extract them. On-chip trace buffering have fixed and limited capacity: these solutions fail at capturing the internal activity for common scenarios where errors manifest thousands of cycles after faults are exercised. In case

of a failure, the log may have been overflown with traces without information about the real cause. Furthermore, on-chip trace buffers [1, 103] require important area overheads. Hardware features added for post-silicon validation purposes are costly and useless to the user once a product goes into production. Therefore, companies normally rely on scan-based techniques to increase the internal observability.

A big problem found during post-silicon validation are non-reproducible errors, which are important contributors to the high cost of current post-silicon approaches [84]. Existing tracing solutions aggravate the reproducibility problem: when attempting to reproduce an error, frequent and time-consuming scan chain and external logic analyzer operations can introduce interferences and non-determinism into the normal program timing, potentially hiding the error. Independently of the interference caused by current state acquisition methods, many bugs are non-reproducible in nature because of the unique conditions that are needed for them to manifest (such as temperature, voltage fluctuations, etc).

To enhance the post-silicon validation phases, in this thesis we also:

- Pursue advancements in system observability through microarchitectural logging technologies that can enable bigger and more flexible buffering capacities, while at the same time have a very low area impact (hardware cost).
- Look for new validation approaches that can extend coverage to non-reproducible errors and that minimally interfere with system performance and operation.

1.2.4 System-Level Simulation for Error Discovery and Diagnosis

The limited internal observability is drifting validation towards methodologies based on rooting errors once an architectural state mismatch is found. Post-silicon validation is principally driven by software tests that are run during a massive number of cycles on real silicon samples. These software tests are generated by specific applications [146], whereas RTL processor models are used to to compute the expected error-free architectural results. As a consequence, big server farms are needed to keep in pace with the validation flow. The biggest issue of these approaches is that catching errors by means of architectural state mismatches incurs huge detection latencies, which ultimately leads to extremely time consuming and complex debugging processes to narrow down the time interval when the fault is exercised.

Once a reproducible error is discovered, methods to transfer and synchronize the silicon state to the RTL simulator [178] are used as a means to debug it. The objective is to help validators to understand the wrong system behavior, to reason about the error-free behavior and to locate the fault. System-level simulation of

RTL models is an inestimable and powerful tool, but it comes at a high price: it is generally 7-8 orders of magnitude slower than actual silicon [16]. Furthermore, when facing design bugs, RTL system-level simulation cannot help much because the bug may probably be present in the models. In addition, debugging the location and root cause behinds errors is a complex and manual step that requires a deep understanding of the microarchitecture.

These facts indicate several research objectives. In this thesis we also aim to:

- Explore alternatives to error discovery methods that rely on golden state generation and slow architectural-state mismatch sighting.
- Enrich the debugging practices with new methods to automatize the error localization and error diagnosis steps. We specially want to minimize the dependence on costly system-level failure simulations. Given the increasing design complexity, we also want to provide tools to help rooting the cause of errors (wrong system behavior and the expected error-free one).

1.3 Thesis Approach

This thesis explores several approaches to address the mentioned research objectives.

We embrace a paradigm shift where error detection is built based on dynamically checking microarchitectural invariants, rather than relying on performance-oppressive redundant execution, or limited fine-grain circuit-level approaches.

Our solutions are end-to-end in nature: instead of individually checking low-level microarchitectural blocks, end-to-end approaches allow verifying high-level functionalities whose implementation is scattered across many components, in a uniform and implementation-independent manner. An end-to-end scheme involves generating a protection code at a source point, and checking for errors only at the end of the path, where information is ultimately consumed.

This thesis approaches the problem by first decomposing the functionalities of a modern processor into high-level tasks that can be verifiable in a cost-effective manner and that when combined together can ensure the correctness of almost the whole core. Specifically, we propose *three* new approaches to detect errors during runtime, which encompass the following high-level functionalities of modern out-of-order cores: *computation-register dataflow, control flow recovery* and *memory dataflow*. This thesis proposes error detection mechanisms for these functionalities because of several reasons. First, the involved blocks are difficult to protect in a cost-effective manner. Second, these functionalities represent around 78.35% of the area of the baseline pro-

cessor described in Chapter 4 (excluding protected structures like caches, TLBs, and queues). And third, according to our previous studies [205], it contributes to around 94% of the SDC SER FIT [4] (excluding protected structures, too).

Then, this thesis addresses the problems of current validation methodologies. We begin by adding value to our error detection mechanisms by extending their applicability to the post-silicon validation phases. Since our error detection methods can catch design bugs (and transient, permanent and intermittent), we minimize the need for slow system-level simulation to perform bug discovery. We also advocate that new transparent continuous logging techniques combined with flexible on-chip buffer capacities allows debugging non-reproducible errors and reduces the dependence on costly external tools.

1.4 Thesis Contributions

The main contributions of this dissertation span two different areas: targeted lightweight runtime error detection and cost-effective post-silicon fault localization and diagnosis.

The key results related to run-time fault detection are as follows:

1. *Register dataflow logic* runtime validation is first deeply studied. We propose a novel runtime technique to detect errors in the register dataflow logic. The solution introduces a novel concept called *signature checking* that detects errors by attaching a token to each produced register value and by matching consumed signatures against source signatures. We show through fault injection campaigns that the rename tables, wake-up logic, select logic, bypass control, operand read and write-back, register free list, register release, register allocation, and the load replay logic are protected with high coverage. The approach is shown to be very effective in detecting faults, and allows designers to choose the coverage ratio by amplifying the signature size.

   We also propose nine different signature allocation policies with different area and power requirements. We show that in-flight signature distribution can be controlled to increase coverage for different register dataflow failure scenarios.

2. We introduce a new microarchitecture that combines register dataflow checking and register value checking. We particularly show how to improve our register dataflow checking technique by integrating it with an end-to-end residue checking scheme. Our evaluations show that a significant amount of power and area

[4] Silent Data Corruption failure rate caused by soft errors, as described in Chapter 2.

can be amortized by combining both solutions, while at the same time protection is extended to the functional units, load-store queue data and addresses, bypass values and register file values.

3. Efficient *control flow logic* runtime validation is then studied. Even though a myriad of targeted solutions exist to detect faults in the instruction sequencing (fetch, decode and allocate logic), none of them can check the complex logic involved in implementing efficient control flow recovery. We propose two techniques to validate the rename state recovery and the squashing functionalities of high-performance out-of-order cores. The proposal uses end-to-end rename state signature checking and tracking of squashed regions to detect faults in the ROB, the rename state recovery logic, the checkpoint rename tables, and in the instruction squashing mechanism. Our evaluations demonstrate the effectiveness of our approach: very high failure reduction rates are achieved with minor power and area overheads.

4. Finally, we target the runtime validation of the *memory dataflow logic* implemented by the load-store queue. Our proposed solution (MOVT), relies on a tiny cache-like structure that keeps the last producer id's for tracked addresses. At commit time, loads are checked to have obtained the data from the youngest older producing store. We have shown that by exploiting the fact that most forwarding store-load pairs are close to each other, coverage can be increased for small set-associative MOVTs by conservatively flushing the pipeline and restarting execution under some scenarios. Three different implementations of the technique with different trade-offs are proposed and evaluated. The solution presents very high fault coverage with attractive area and performance overheads. Moreover, MOVT can be used to solve the vulnerability hole inherent to redundant multi-threading designs where the load-store queue activity is not replicated across threads.

The key results related to cost-effective fault localization and automated diagnosis are the following:

5. Existing tracing solutions are constrained by the capacity and area of on-chip logs. A new software-hardware logging system to increase the internal observability is proposed to alleviate these issues. First, we show that by sequestering physical memory pages from the application being run and re-purposing them to store activity logs we can increase observability by means of logs that can be sized to suite validation needs and without requiring big hardware structures. We then propose a hardware structure that temporally buffers internal activity

at full speed and connects with the data cache to access the log pages. We study its efficiency and show that by offloading the buffer during idle cache cycles and by letting the application allocate lines as needed, performance is not critically impacted.

6. We show how to combine our error detection mechanisms together with the described logging system to construct a novel post-silicon validation methodology. As a practical example, we particularly focus on the memory dataflow logic implemented by the load-store queue. By using our runtime bug-detection mechanisms together with the proposed non-intrusive logging system, we eliminate the simulation steps required to generate golden outputs for test programs and we extend coverage to non-reproducible errors without any intervention to orchestrate the activity logging.

7. Current debugging practices are manual and cumbersome. We present a diagnosis algorithm that analyzes the log produced by our validation system and automatically localizes and diagnoses errors in the load-store queue. Not only the fault location is determined, but also the wrong behavior and the failure-free expected one. We evaluate its efficiency and show that a very high percentage of errors can be automatically diagnosed for different precision levels.

1.5 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 presents basic background information and the basic concepts necessary for proper understanding of this document. Chapter 3 contains a survey and a critical analysis of related work in the area of fault tolerance and post-silicon validation. Chapter 4 describes the evaluation framework. This chapter details the processor performance simulator, the benchmarks, and the area/power/delay/error coverage models that have been used in this thesis. We also describe the baseline error detection mechanisms that our processor model incorporates.

We distribute the main contributions of this thesis in the next four chapters:

- In Chapter 5 we present a runtime solution to detect errors in the register dataflow logic, register data values and computation. Several implementations with different coverage, area and power trade-offs are also studied.

- In Chapter 6 we detail two runtime solutions to detect errors in the control flow recovery logic of modern out-of-order processors. The first one implements an

end-to-end validation of the rename state recovery, whereas the second one is an end-to-end solution for validating the instruction squashing.

- In Chapter 7 we describe a proposal that targets the runtime validation of the memory dataflow logic implemented by the load-store queue. We present three different implementations, with varying degrees of error detection coverage, performance costs and design complexity.
- In Chapter 8 we detail a novel hardware-software solution to discover, locate and diagnose errors during post-silicon validation. To show the potential of our approach, we focus on how to apply it to the memory dataflow logic implemented by the load-store queue.

Finally, Chapter 9 draws the main conclusions of this dissertation and points out some ideas for future work. The microarchitecture description of our baseline processor is found in Appendix A.

CHAPTER 2

BACKGROUND

The scope of this Chapter is to present some basic concepts that are relevant to the general topic of this thesis. In Section 2.1 we describe the economic impact of hardware vulnerability in computing systems. Section 2.2 defines some basic concepts related to faults, errors and metrics. Hardware failure mechanisms are categorized in Section 2.3. Some relevant fault tolerance fields are introduced in Section 2.4. We finally include in Section 2.5 an overview of the different processor validation phases.

2.1 Economic Costs of Hardware Reliability

Technology reliability problems are already making an impact in the industry across all sectors that rely on information systems. Despite being a sensitive problem for microprocessor companies, several reliability issues have been notoriously public.

Sun Microsystems was one of the first affected companies affected by technology scaling issues. In 2000 it was reported that their UltraSparc II servers were crashing at an alarming rate. After arduous works to understand the reason behind this behavior and significant dissatisfaction from their clients, Sun found that the problem was caused by bit flips in their insufficiently protected cache memories [3, 14]. In 2004, Cyprus reported an incident where soft errors brought down an automotive factory every month [129]. In 2005, HP revealed that a CPU farm in Los Alamos National Laboratory crashed frequently due to particle striking several cache tag arrays [117].

Other examples of reliability hazards related to design bugs have also hit the microprocessor industry. In 1994 it was discovered that the Intel Pentium microprocessor made occasional errors in floating point divisions [150]. Intel ended up offering to replace all flawed Pentium processors, and despite a small fraction of consumers

requested a replacement, a cost of 475 million dollars was incurred. In 1997, it was found a bug affecting the 6x86 Cyrix microprocessors series. The bug allowed to build unprivileged programs that would halt the system in an infinite loop. Fortunately, a workaround at the OS level allowed avoiding the recall of all products [218]. In 2007, a flaw was discovered in the TLB of several AMD Phenom processors series that could cause a system lock-up in rare circumstances. Initial BIOS and software workarounds disabled the buggy TLB, incurring in performance degradations of 10%-15% on average. The bug put a temporary stop to the production and severely harmed AMD reputation [180].

Regarding degradation errors, Intel announced in 2011 a problem in the 3Gbps SATA ports of its Cougar Point chipset famility. The problem was located in a faulty and leaky transistor that caused the degradation of the IO link over time. Despite Intel performed stress burn-in tests before releasing the product, the problem was detected by OEMs. Intel halted the shipment of affected products, recalled the faulty ones and incurred an estimated cost of 700 million dollars [95].

The impact of faulty hardware is spreading across to the whole spectrum of companies. Standish Group Research reports that hardware failures are involved in around 21% of companies unplanned outages [53]. The operations of current companies is increasingly linked to their computing infrastructure, and any system downtime directly hampers the productivity of a company. Despite planned downtime represents 90% of downtime [185], the unexpected nature of outages are more damaging for a company. It has been reported [185] that on average, businesses lose between 1400 and 1800 dollars for every minute of information technology (IT) system downtime. Furthermore, market segments like financial services, telecommunications, manufacturing and energy are more susceptible to a high rate of revenue loss during IT downtime. A brokerage firm was estimated to lose around 108.000 dollars per downtime minute (6.5 million dollars per hour) on average in 2008 [53]. Furthermore, the economic consequences of outages do not take into account the change in the client fidelity or the opportunity loss when the computing infrastructure was not available.

2.2 Reliability Concepts and Metrics

This section reviews basic concepts related to faults, errors and metrics.

2.2.1 Basic Terminology and Classification of Errors

A **fault** is defined as an undesired event or physical imperfection, such as a design bug, a manufacturing defect or a bit flip due to cosmic particle strikes.

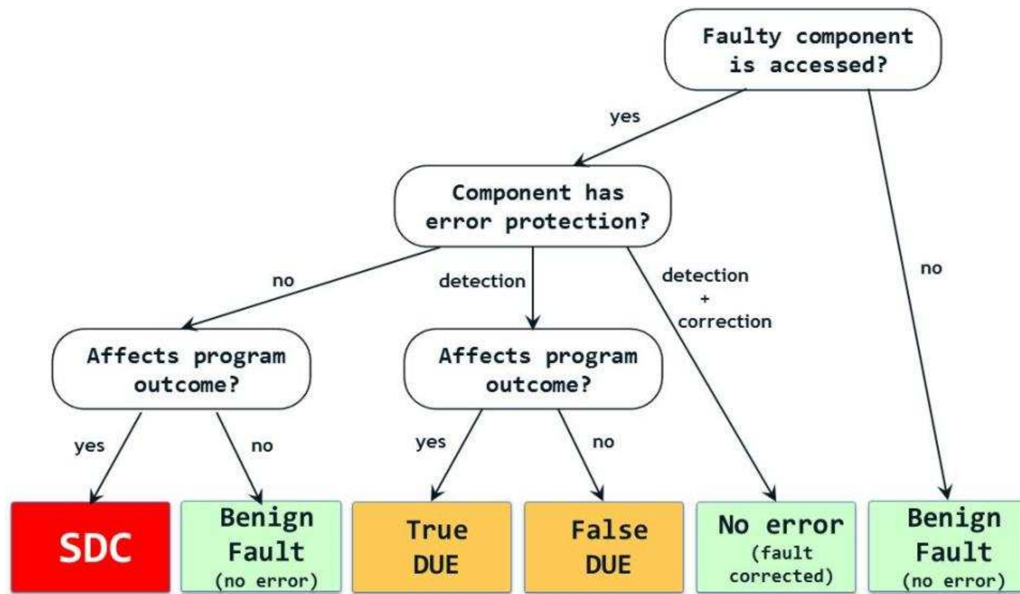


Fig. 2.1: Classification of faults effects

The activation of a fault is termed as **error**, and can propagate up to higher levels (e.g. circuit, microarchitecture and application). However, **masking** effects can avoid an error from manifesting to the upper levels whenever the output of the affected blocks continue being correct [126, 128, 175]. As an example, a transient error in a mispredicted instruction will not affect the application output, or at the application level, a bit flip in the result of a dead instruction will be harmless.

A non-masked and non-corrected error can result in a **failure**: an unexpected behavior that is visible at the architectural or user level (wrong register values, memory or I/O updates). Hence, a failure is a special case of an error [129], and both terms are generally interchangeably used. Failures include data corruption, system crashes and hangs.

Architecturally, errors are further classified into two groups, as shown in Figure 2.1. Faults that manifest and are undetected are termed as **silent data corruption errors (SDC)** if they are visible at the user level. The most insidious manifestation of SDC errors is the invisible alteration of user sensitive data. On the other hand, faults that manifest but are simply detected are called **detected unrecoverable errors (DUE)**. Depending on if the error would have manifested as a failure or not, DUE errors are further classified as True DUEs or False DUEs, respectively. Faults that are not exercised or faults that do not affect the program outcome are simply benign faults and are not considered as errors.

2.2.2 Fault Tolerance Metrics

Microprocessor designers commonly express error rates as **FIT**s units (Failures in Time) because they are additive across the components of a system. One FIT corresponds to one failure in a billion (10^9) hours. The sum of SDC and DUE FIT is usually referred to as the processor FIT value.

The FIT value of every microprocessor component is estimated based on two factors: the **raw device error rate** for different sources of failure (such as soft errors or degradation) and a derating term called **architectural vulnerability factor (AVF)**. AVF is the probability that a fault will end up being a DUE error (DUE AVF, meaning detected) or SDC error (SDC AVF, meaning undetected).

Whereas the raw device error rates depend on process technology, AVF depends on masking effects at higher-level abstractions, including circuit, microarchitecture, architecture and software designs. An important aspect of estimating error rates is considering that not all faults affect the final outcome of a program: (i) at the circuit level, such effects include logical masking, electrical masking and latching-window masking, (ii) at the microarchitectural level masking effects are encountered when errors affect idle, invalid, mispeculated or prediction state/activity, (iii) at the architectural and software level, faults can be masked when exercised by performance enhancing or dynamically dead instructions, or when logically masked [128, 175].

Instead of FIT values, other **reliability** metrics are often used, because they are more meaningful to the end user. A **reliability** metric indicates the probability that a system has been operating correctly since moment 0 until a moment t. Vendors express SDC and DUE error rates to costumers in terms of **Mean Time Between Failures (MTBF)** units, which expresses the mean time elapsed between two errors (either failures or not). **Mean Time To Repair (MTTR)** and **Mean Time to Failure (MTTF)** are also popular metrics. MTTR indicates the mean time required to repair an error once it is detected (either through specific recovery mechanisms or through regular system restart). MTTF captures the mean time to produce the first incorrect output. MTTF is inversely proportional to the FIT value of a system. As Figure 2.2 shows, $MTBF = MTTF + MTTR$. However, since none of these metrics are additive across the components of a system, designers normally work with FIT values.

Reliability is the most well-known term but, it is not very indicative of the fault tolerance of a system. **Availability** is a metric that indicates the probability of a system being correctly operative at a given time, and is computed as $Availability = \frac{MTTF}{MTTF+MTTR} = \frac{MTTF}{MTBF}$. Availability is popularly quantified in 9s. As an example, a system with 99.999% availability (which corresponds to 5 minutes of downtime per

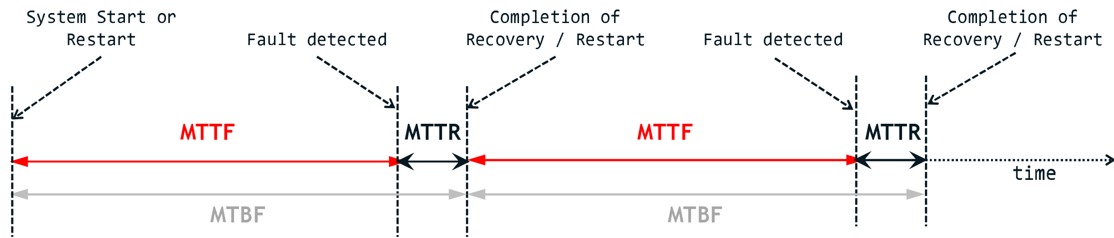


Fig. 2.2: Vendors fault tolerance metrics

year) is said to have five nines of availability.

Also, **serviceability** is a broad qualitative term describing how easily faulty components are identified, diagnosed and/or isolated.

These three related attributes are commonly referred to as the **RAS** (Reliability, Availability, Serviceability) features of a system and are considered when designing, manufacturing, purchasing or using a computer product.

2.3 Hardware Failure Phenomena: How Electronics Fail

Traditionally, hardware errors have been divided into four main categories according to their nature and duration: transient faults, intermittent faults, permanent faults and design bugs.

Transient faults are non-permanent faults caused by several phenomena including voltage fluctuations, electromagnetic interference and electrostatic discharge. However, the major cause is radiation to the chip [129]. High energy cosmic particles interact with atmospheric nuclei and create a cascaded generation reaction of many nucleons such as neutrons, protons, muons, etc. These particles, normally neutrons, strike silicon devices randomly in time and location. When the particles hit the silicon devices they generate electron hole pairs resulting into generation of charge, as Figure 2.3 shows. When this charge exceeds a critical charge (Q_{crit}) [222], they can corrupt a data bit stored in the memory or create a current glitch in any gate in logic. Since the corruption does not harm the transistor structure, the fault will disappear once the cell or transistor output is overwritten. Transient faults manifest as transient errors, also known as soft errors. Whereas packaging radiation and alpha particles can generally be minimized through specific material manufacturing, cosmic rays are unavoidable and their flux increases exponentially with altitude [222]. Transient faults has been considered one of the most predominant source for errors in microarchitectures for current and past silicon technologies [188].

Intermittent faults appear and disappear repeatedly but not continuously in

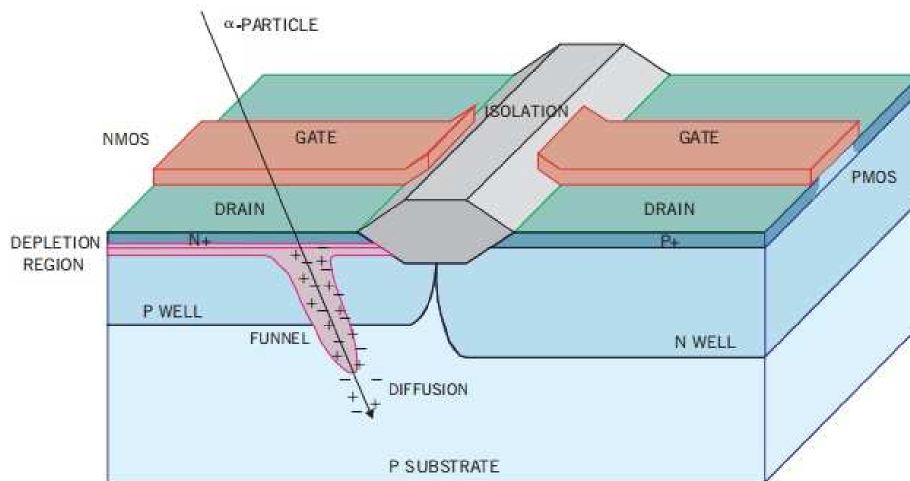


Fig. 2.3: Particle strike causing current disturbance [111]

time. These faults are non-permanent, as in the case of transient faults. As opposed to transient faults, the replacement of the affected device eliminates an intermittent fault. Errors induced by intermittent faults usually occur in bursts when the fault location is exercised. Generally, voltage peaks and falls, as well as temperature fluctuations originate intermittent faults. Intermittent faults often precede the occurrence of permanent faults [42]. High frequency circuits will initially suffer from intermittent delay faults, before open faults occur.

Permanent faults, also known as hard faults, involve errors that are irreversible due to physical changes. These faults are either caused by run-time aging or are originated during the chip fabrication process. Until disabled or repaired, a permanent fault will potentially keep producing erroneous results. There are mainly two sources for permanent faults [186]:

- **Physical wear-out**. Several sources of failures can be classified as aging phenomena. Electromigration [92] refers to the displacement of the metal ions caused by the current density flowing through the conductor. As seen in Figure 2.4, the depletion and accumulation of material creates voids and hillocks, which can lead to open and short faults, respectively. Negative-bias temperature instability [6] (NBTI) breaks progressively silicon-hydrogen bonds at the silicon/oxide interface whenever a negative voltage is applied at the gate of PMOS transistors. The main consequence is a reduction in the maximum operating frequency and an increase in the minimum supply voltage of storage structures to cope for the delay faults. Oxide gate breakdown [194] ultimately manifests as a conduction path from the anode to the cathode through the gate oxide as a result of the reduced dimensions of transistors' gates. Other

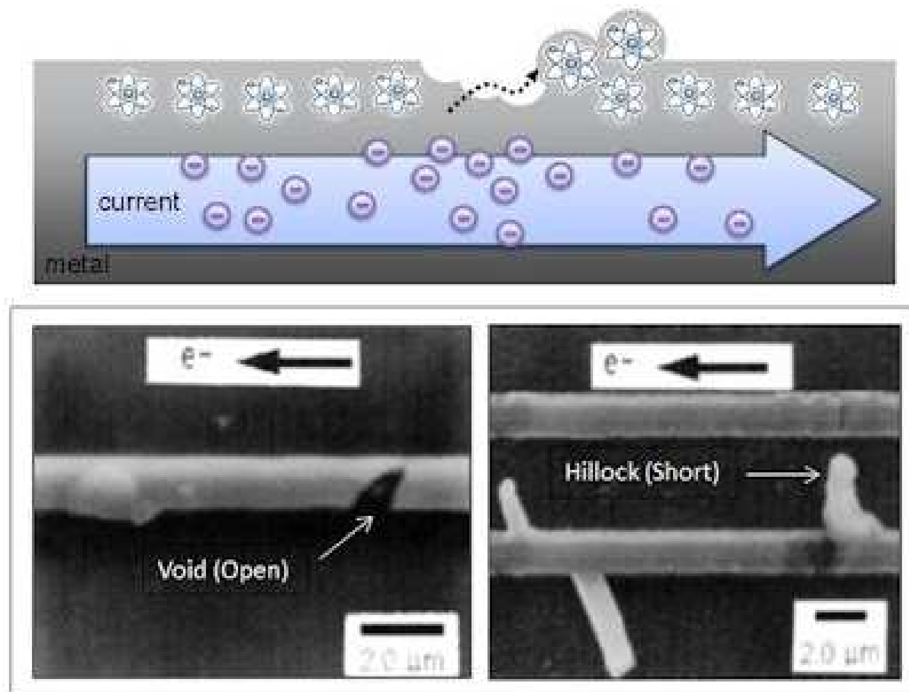


Fig. 2.4: Physical wear-out phenomena, open and short creation [59]

physical events that can reduce the reliability of devices are stress migration for wires, thermal cycling for the package and pins, and hot carrier injection for transistors.

- **Fabrication defects**. Chip fabrication is an imperfect process, and product samples can be fabricated with inherent faults. Defects at manufacturing time cause the same problems as wear-out faults but from the very first moment. Plus, it is more likely to have multiple fabrication defects in a chip than multiple wear-out faults manifesting in the field at the same moment. Similarly, tolerable latent fabrication defects can exacerbate during lifetime and lead to intermittent contacts [42].

Design bugs are a special type of permanent faults. Even in an ideal scenario with perfect manufacturing process and total reliability against transient faults, a fabricated microprocessor may not operate correctly in all situations due to a mismatch between the implementation and the specification, or due to an incomplete specification. These kinds of faults are normally referred to as functional faults or design bugs [35, 208].

2.4 Aspects of Fault Tolerance

Dealing with hardware and design faults involves several challenges that constitute in a broad sense the field of fault tolerance research. The fault tolerance area is generally classified into several overlapping fields:

- **Error detection**. The most crucial aspect of fault tolerance is determining whether the system operation was affected by an error or not. To achieve detection capabilities, error detection mechanisms are included into the microprocessor design in order to regularly check the internal state and activity during its lifetime (after the microprocessor has been sold). Adding error detection (but not correction) to a structure eliminates SDC errors, converting those faults to DUE errors. As a consequence, error detection mechanisms allow reducing the SDC FIT. Error detection is the pillar capability that allows enabling other fault tolerance aspects.

- **Error diagnosis**. Error diagnosis has been traditionally conducted during the post-silicon validation phases, as a method to understand the reason behind failures and bugs and guide their correction. However, diagnosis is also used in mission critical segments during their lifetime. Their objective is to guide an adequate higher-level repair or reconfiguration mechanism that can deal with the affecting fault. Since errors can be caused by faults with different nature, error diagnosis is often needed to pinpoint the error type as well as the location of the error. The diagnosis latency is not generally a problem because its cost is paid after an error has been detected. Therefore, software solutions are also attractive and cost-effective.

- **Hardware repair and error reconfiguration**. Once an error has been detected and diagnosed, additional actions are taken in order to avoid that the fault will be exercised again during the processor lifetime. If the fault is permanent or intermittent, repair and reconfiguration can be handled through disabling the faulty parts of the affected component if possible [26, 149]. Repair and reconfiguration can also be conducted at a higher granularity, through physical replacement of the microprocessor, or by means of disabling the faulty core and using a spare one: the ubiquitous chip multiprocessor (CMP) systems makes repair and reconfiguration a realistic and simple approach. Software approaches like software circumvention [116] are a viable solution for single core designs. For transient faults, there is no need for repair or reconfiguration.

- **Error Recovery**. After repair and reconfiguration, the last step is to recover the system state in such a way that no trace of the fault exists anymore and

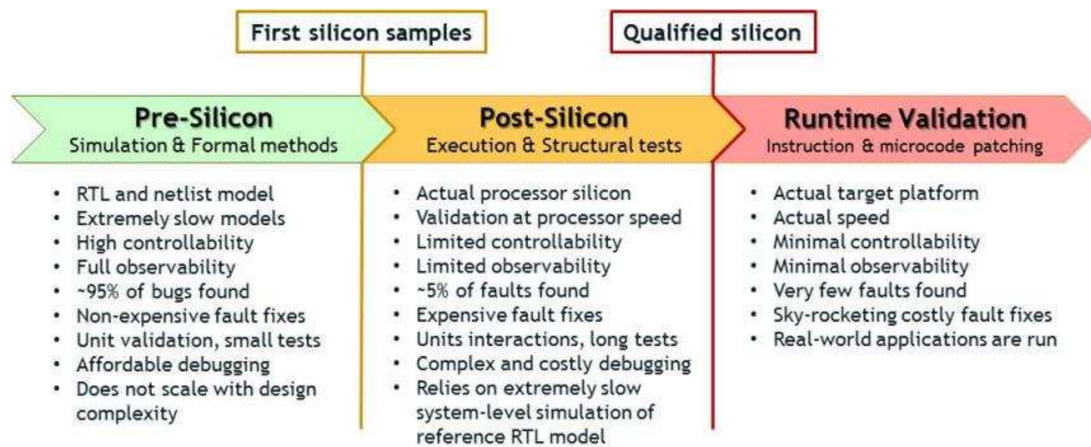


Fig. 2.5: Validation domains and characteristics

normal execution can restart. As a consequence, recovery allows improving the DUE FIT rate. Any state or data possibly corrupted by the fault must be restored and prevented from being visible to the software. A plethora of effective solutions have been proposed for error recovery, a well-studied field [186]. There are two approaches to error recovery: Forward Error Recovery (FER), that corrects the error without reverting to a fault-free state and Backwards Error Recovery (BER), that restores the state to an old known fault-free state.

Multiple efficient BER recovery solutions exist, spanning from pure hardware techniques to pure software approaches. Hardware BER recovery solutions range from simply flushing the speculative state [186] of the microprocessor pipelineto relying in hardware checkpoints [151, 187, 214] / transactional memory [123] for shared-memory multiprocessors. Software BER recovery schemes do not require hardware modifications, because they save a snapshot of the application's state. Software BER schemes have been also proposed for parallel and distributed high-performance computing (HPC) applications [217]. These BER options do not incur severe overheads in terms of performance penalty for saving state. However, BER hardware solutions require important design complexity and area overheads depending on the error detection latency and the confinement capabilities, whereas BER software solutions impact the application design.

FER recovery schemes basically include Triple Modular Redundancy (TMR) systems, which are extremely costly in terms of performance, power and area overheads.

2.5 Validation and Debugging: Background

Microprocessor validation efforts are commonly structured into three domains: pre-silicon, post-silicon and runtime validation. Figure 2.5 summarizes the characteristics of each one of these domains, with their own features.

2.5.1 Pre-Silicon Validation

Pre-silicon validation aims at detecting bugs before silicon prototype are available. A register-transfer level (RTL) model is verified by means of simulation-based and formal techniques to check the equivalence with the reference model.

Simulation-based pre-Si techniques run small [4] tests in the RTL model and compare the outcome to the golden architectural model. Simulation is orders of magnitude slower than real silicon, not exhaustive and severely constrained by limited scope models. Formal verification methods determine the absence of faults, and they are locally applied to small units because of their extreme algorithmic complexity. High controllability and full observability is available to the validators during pre-silicon validation, and most of faults are found in this phase and debugged using affordable methods.

When sufficient validation coverage is obtained, the RTL is synthesized into an optimized circuit netlist [67] that is sent to the fabs (taped-out) to obtain a first prototype. The *post-Silicon validation* phase starts after.

2.5.2 Post-Silicon Validation

Post-silicon validation uncovers most of the faults undetected during the pre-silicon validation phase, but the fixes requires producing new expensive prototype samples. Fabrication defects, electrical faults and design bugs are discovered during *post-silicon validation* through structural testing and functional validation.

Structural Testing Structural testing aims at uncovering faults introduced by the manufacturing process. The netlist is used as the golden model, and is used by Automatic Test Pattern Generation (ATPG) software to infer optimized test sequences and golden outputs that are ultimately probed in the real silicon to expose different types of faults. Engineers do normally incorporate *design for testability* (DFT) features into processors to increase the minimal system observability and controllability. By using DFT features, engineers can inject arbitrary states (tests), freeze the execution or obtain some internal state.

Functional Post-Silicon Validation Functional post-silicon validation aims at mainly debugging design and electrical bugs. High coverage can be attained by executing longer sequences of tests because the full execution performance of the silicon is orders of magnitude higher than RTL simulation.

Functional post-silicon validation is principally driven by random tests and commercial applications that are run at orders of magnitude of higher throughput than during pre-silicon verification. The objective is to exercise the interaction of the components and corner cases by stressing them under a burst of similar stimuli. This step requires a farm of servers running system-level simulation of RTL models to obtain the golden output for the tests being run.

The debug process starts by observing result mismatches, a system crash, deadlock or data corruption. The reproducibility of the bug is first attempted, so that the triggering conditions can be identified. If bug reproducibility is obtained, tracing techniques [1, 103, 178, 220] are used to increase the observability and controllability. These solutions allow capturing the succession of events that lead to the failure.

Validators also attempt to transfer and synchronize the silicon state to the RTL model, as a way to achieve higher observability [178]. System-level RTL simulation is therefore heavily used both for the error discovery process and to help in the the debugging of these errors, even though it is extremely slow.

2.5.3 Runtime Validation

Despite designs are tested extensively before being released to the market, sometimes undetected bugs slip into the final product. *Runtime verification* is a new research topic meant to complement the validation phases so that a higher fault tolerance efficiency is achieved against undiscovered bugs.

Current processors do not incorporate solutions to catch or diagnose undiscovered bugs during their lifetime. These are actually debugged by processor companies, who include in their products solutions such as *instruction patching* and *microcode updates*, that allow fixing non-critical errors once they have been detected in the field. In the case of serious bugs (those related to the computation correctness), nowadays vendors have few solutions other than retiring the shipped products.

CHAPTER 3

RELATED WORK

In this Chapter we summarize the wide spectrum of related work in the area of fault tolerance and post-silicon validation. The objective is providing an overview on how *traditional approaches* have dealt with these aspects and what problems they have exhibited. State-of-art solutions relevantly related to our proposed solutions and to our approach are described in the following Chapters.

In this Chapter we make an overview of different approaches implemented at different levels, ranging from circuit-level up to software-level. We describe the most paradigmatic ones, and we focus on those that are able to detect (or mitigate) faults in the microprocessors hardware. We also cover existing industrial solutions that are used in post-silicon validation. It is important to note that this Chapter does not analyze solutions for recovery, repair or reconfiguration, as these are not objectives of our thesis. Similarly, we do not detail solutions for the memory or uncore.

Section 3.1 analyzes the most important architectural- and microarchitectural-level techniques that rely on redundant execution without software intervention, Section 3.2 describes common coding schemes to protect data (storage) and functional units, Section 3.3 covers some of the existing solutions to detect or mitigate faults at the circuit level and Section 3.4 describes software or hybrid solutions that provide error detection by means of redundant execution. For each one of these approaches, we include a table that summarizes the described solutions and compares their features. For every technique, we also highlight their weakest or not desirable aspects. Finally, in Section 3.5 we cover existing industrial mechanisms, techniques and methodologies that are used during post-silicon validation to increase the observability or controllability of silicon samples, or that are used to increase processor debugability.

3.1 Re-execution- Based Hardware Techniques

One of the most studied reliability mechanism is to use the existing temporal and/or physical redundancy at the microarchitectural and architectural level. This is the case of the family of techniques that detect faults by comparing the outputs of two redundant executions of a thread without software intervention. The rationale is that a fault will affect just one of the two redundant executions, and hence, a fault can be caught by comparing the outputs at the architectural level, once the fault has propagated to a visible point.

Rotenberg's AR-SMT [162] is a seminal work exploiting the concept of redundant re-execution. Two threads are defined: the A (active) and R (redundant) threads. The A thread always runs ahead of the R thread, and provides R the outputs of its computations through a special buffer. A result match allows the instruction from the R thread to commit its results, hence accumulating a golden architectural state that can be used for recovery in the face of a soft error failure. AR-SMT requires a huge hardware overhead, eliminates the opportunity to execute another non-redundant thread, and it suffers from performance stalls whenever the buffer saturates. Also, high power costs are paid.

The SRT (Simultaneously and Redundantly Threaded) proposal by Reinhardt and Mukherjee [157] introduced the novel concept of *sphere of replication*. All activity within the sphere is replicated. Values crossing the sphere are the outputs and inputs that require comparison and replication, respectively. A large sphere replicates more state; however, updates to that state occur independently in each execution copy, with no need for synchronization. SRT's sphere of replication includes the register file (as opposed to AR-SMT), which avoids checking the result for every instruction, and reduces the communication and synchronization among threads. Checking is performed just for store addresses, store data, and load addresses. However, none of the redundant architectural register files can be used for recovery. The memory space is not replicated: only the leading thread accesses the data cache and forwards the value and address to the trailing thread through a special FIFO. The trailing thread issues loads in program order and non-speculatively to that queue, and performs address checking. The trailing thread does not make use of the load-store queue logic. This fact introduces a vulnerability hole, since any fault affecting the load-store queue state or activity will remain undetected.

Vijaykumar *et al.* [207] adds soft error recovery capabilities to SRT processors having spheres of replication including the register file. SRTR stalls the leading thread from committing instructions until the trailing thread checks the instructions for faults. To reduce stalls due to pressure on the core resources, SRTR checks

the outcomes of an instruction as soon as the trailing instruction completes, rather than at commit time. SRTR uses a special value queue to store register values for redundant checking. The biggest issue is the complexity in hardware required to form the dependence chains and store the leading thread outcome values.

The main constraint on the performance achievable by solutions based on hardware redundant execution is the bottleneck imposed by sharing the issue, the functional unit bandwidth, as well as the ROB [183]. SHREC (SHared REsource Checker) was proposed [183] as a soft error tolerant alternative to alleviate these issues. SHREC uses asymmetric reexecution [10] to relieve that pressure. In asymmetric redundant execution, an instruction is checked by its redundant version after the original instruction has executed and using input operands already available from the first (original) execution. Redundant dependent instructions can be reexecuted simultaneously because each of them will consume the values produced by the original execution. To achievo so, instructions in the ROB that have executed are moved in program order to a small in-order issue queue to perform the checking (functional units are shared between the normal issue queue and the in-order issue queue). Redundant instructions fill the issue bandwidth left idle by the original instructions. Redundant loads obtain their values directly from the data cache. Given that the accesses are done in program order, no need for memory disambiguation is required. This implies that the load-store queue state and logic is protected by SHREC. Recovery can be achieved in the face of transient faults, because no instruction leaves the pipeline without having being checked. SHREC provides soft error protection for the backend of a core, unlike RMT techniques which extend the protection to the frontend.

Replicating all instructions comes at the cost of significant performance degradation and power consumption. Thus, the research community has explored the possibility of replicating only a subset of the instructions.

Gomaa and Vijaykumar's [64] approach attempts to reduce the performance penalty by replicating instructions during low-IPC phases and L2 misses. In order to enable partial explicit redundancy, the technique requires a big structure to continuously communicate the redundant thread the resume-point state. Low soft error coverage is thus achieved for medium and high-IPC application. To alleviate this issue, implicit redundancy is exploited through instruction reuse techniques [184]. Reuse avoids redundant computations (no redundant thread is needed) but at the cost of a loss of coverage that is subsumed by the program inherent value reuse capabilities. However, there is no performance loss because there is no explicit reexecution.

Vera *et al.* propose a Selective Replication [205] (SelR) scheme guided by the vulnerability of the instructions. SelR re-executes instructions that have a significant contribution to the vulnerability, replicating the minimum number of instructions.

Table 3.1: Comparison of hardware-level global re-execution techniques

	Performance, Power Costs	**Recovery**	**Domain**	**Sources Of Failure**	**CPU Featuring**	**HW Cost (complexity)**	**Coverage**	**Full Re-execution**	**Detection Latency**
AR-SMT [162]	Very High	Yes (ECC in RF)	Full	Soft errors	SMT	High	Full	Yes	Bounded
SRT [157]	High	No	Full	Soft errors	SMT	Medium	Full	Yes	Unbounded
SRTR [207]	Very High	Yes	Full	Soft errors	SMT	High	Full	Yes	Bounded
SHREC [183]	Medium	Yes (ECC in RF)	Backend	Soft errors	None	Low	Full at backend	Yes (Asymmetric)	Bounded
Opportunistic [64]	Low	No	Full	Soft errors	SMT	Medium	Low	Partial (low IPC phases)	Unbounded
SelR [205]	Low slowdown, Medium power	Yes (ECC in RF)	Backend	Soft errors	None	Low	Very High at backend	Partial and Asymmetric (AVF prediction)	Bounded
SlicK [143]	Medium slowdown, High power	No	Full	Soft errors	SMT	Very High	Full	Partial (value prediction)	Unbounded
SpecIV [93]	Medium slowdown, High power	No	Full	Soft errors	SMT	Very High	Full	Partial (value prediction)	Unbounded
Slipstream [197]	Speed-up, High power	Yes	Full	Soft + hard errors	CMP	High	Very High	Partial (ineffectual insts. removal)	Bounded
Lockstepping [198]	High	No	Full	Soft + hard errors	CMP	Low	Full	Yes	Bounded
Fingerprinting [182]	Low slowdown, High power	Yes (checkpoints)	Full	Soft + hard errors	Lockstep	Low	Full	Yes	Bounded (huge)
CRT [127]	High	No	Full	Soft + hard errors	CMP + SMT	Medium	Full	Yes	Unbounded
CRTR [63]	Very High	Yes	Full	Soft + hard errors	CMP + SMT	High	Full	Yes	Bounded
DIVA [10]	Low slowdown, Medium power	Yes (ECC in RF)	Backend	Soft + hard errors, bugs	None	High	Full at backend	Yes	Bounded
BCS [196]	Low	Yes	Full	Bugs	None	Medium	Unclear	Partial (signature locality)	Bounded

The AVF of an instruction is estimating by the time it spends residing in the issue queue. SelR can be seen as an evolution of SHREC to deal with its performance overhead, because replicas are placed into the in-order issue queue upon allocation, rather than re-circulated at commit time. This further alleviates the pressure in the ROB.

SlicK (Slice-Based Locality Exploitation for Efficient Redundant Multithreading) [143] also makes use of explicit partial redundancy in the context of an SRT processor. SlicK relies on the use of predictors for values exiting the sphere of replication. The leading thread is executed entirely but it uses a set of predictors to attempt to verify the outputs of the leading thread without re-execution. Instructions that belong to the backward slices of outputs that the predictors were not able to verify are reexecuted by the trailing thread. Slick requires big predictors and complex hardware blocks to perform on-line backward slice extraction.

Speculative Instruction Validation [93] (SpecIV) extends the concept of value prediction in the leading thread to any kind of instruction. SpecIV does not require slice formation and reduces the performance impact of the original SRT implementation. Nevertheless, the technique requires a big area overhead in the form of value predictors, as well as deep modifications to the existing core microarchitecture. Futhermore, a general problem inherent to reliability solutions based on value prediction is the loss of coverage whenever a fault corrupts the leading thread data in such a way that it exactly matches with the predictor's output.

The Slipstream processor [197] was a pioneer reliability solution based on partial replication of the leading thread. The trailing thread is monitored to find ineffectual and highly-confident branch predictions. The future instruction slice instances leading to these ineffectual computations are removed from the A thread. The leading thread is a partial and speculative redundant version of the trailing thread, but is sped up because it has less instructions to process. At the same time, the trailing thread is sped up by the leading thread by warming up the caches and by providing branch outcomes. Slipstream processors provide incomplete fault tolerance because not all instructions are explicitly and redundantly executed. Slipstream poses two main disadvantages: (i) the added hardware is complex and costly in area, and (ii) the detection coverage is partial.

Redundant re-execution has also been studied for other computing processors not implementing SMT. Redundant threads can run on two different cores within the same multicore processor, or on the different cores from two separate processors. We next detail them.

Lockstepping is an example of systems exploiting physical redundancy by integrating two or three different processors on a dual or triple modular redundancy

configuration. The cores are tightly synchronized so that there is cycle-by-cycle input replication and output comparison, as well as fully deterministic execution (same internal activity) [181, 198]. Clearly, design heterogeneity is not possible for lockstepping. Fault detection is guaranteed for transient faults and hard faults, but on the other hand lockstepping is not well suited for market segments other than mission critical. The cost in performance, power and validation is skyrocketing.

Smolens *et al.* [182] evaluate the efficiency of lockstepped systems that create checkpoints of system state and rollback processor execution when a soft or hard error is detected. They observe that no previous lockstepped systems can provide at the same time satisfactory error detection latency and comparison bandwidth. Their solution, called Fingerprinting, allows to alleviate this trade-off. A fingerprint is a hash value computed over the sequence of updates to a processor's architectural state during program execution during a checkpoint interval. Fingerprints are less costly or intrusive than other redundant re-execution schemes that check results on a per-instruction basis, but on the other hand, fingerprints extend the error detection latency. Fingerprints' aliasing probability is low and can be reduced by increasing the hash size. Moreover, given a fingerprint size, its detection capabilities are independent of the number of updates (hash additions) to the code.

Mukherjee *et al.* [127] proposed CRT (Chip-level Redundant Threading), an implementation of SRT under a chip-multiprocessor (CMP). Compared to a SMT implementation, using a CMP for SRT avoids resource contention among the threads and extends the coverage to permanent faults.

Similarly, the concept of SRT and SRTR was expanded to multiprocessors by Gomaa *et al.* Threaded processors with Recovery (CRTR) [63] are the adaption to CMP processors of SRT and SRTR, respectively. Despite the fact they are able to target hard faults, the performance and power overheads are still massive.

New proposals extend the error coverage to design bugs, by exploiting physical redundancy through design heterogeneity: an ISA-compatible core different to the main core is added to work as a checker. We next detail them.

Austin proposed DIVA (Dynamic Implementation Verification Architecture), a ground-breaking work [10, 38, 215]. DIVA uses an incomplete checker in-order core. Instructions arriving to the ROB are moved in program order to the DIVA checker, together with their input operands and still speculative results. For every instruction, the DIVA checker: (i) verifies that the proper result was produced by the main core and (ii) verifies that operand values flow correctly from one instruction to another. After verification, results are committed to the architectural state. For verifying the computation, DIVA exploits asymmetric execution which completely eliminates instruction dependencies. As many checking functional functional units are added as

the main core has, to catch up with the main core IPC. The redundantly computed values are compared against the pre-computed ones. Regarding operand flow checking, the DIVA core verifies that the received source operands match with the ones read from architectural storage. The data cache is also redundantly accessed by loads. All architectural registers and memory are protected with ECC. DIVA offers several advantages: it extends the coverage to hard faults and design errors , and does not require a SMT processor. However, it does not scale well to big high-performance cores and is not suitable for small in-order cores due to its overheads. The operand flow checkers require a huge and messy operand bypass network, which is costly and complex from a design perspective.

A recent work, the Beta Core Solution (BCS) [196] reduces the power cost of full re-execution. BCS uses a minimal complete in-order checker core, as opposed to DIVA. For every bundle of instructions waiting for retirement, a signature is generated by incorporating timing and microarchitectural information. The signature is searched in two signature tables: one keeping track of signatures corresponding to a-priori bugs or bugs discovered during run-time, and another tracking signatures of correctly verified previous bundles. Missing both tables indicates an unverified computation and requires transferring control and state to the checker core to determine if computation was correct or not, updating the tables accordingly. The checker core is simpler than the DIVA one because it does not need to execute 100% of the time and it does not need to keep up with the main big core. However, the biggest issues are that it is is not clear what is the signature construction method and which subset of control signals are used to detect any possible bug. BCS is not able to detect faults in the checker core and there is still a minimal performance degradation.

3.2 Error Coding Techniques

The theory of error coding is a rich area in mathematics. Coding schemes are one of the most popular microarchitectural error protection mechanisms.

Error Coding Techniques for Memories

Error codes are generally applied to storage elements. From an implementation perspective, error coding is suitable when the data being protected is almost static (this is, generated once but not modified during its lifetime) and wide enough to amortize its overheads. Otherwise, multiple costly code generators and checkers would be required at every consumption and modification point.

Parity codes are possibly the simplest error detection technique. A parity code

Table 3.2: Comparison of error coding techniques

	Recovery	Separable	Domain	Supported Operators	HW Cost (complexity)	Sources of Failure	Concurrent Error Detection
Parity [210]	No	Yes	Data	- (data)	Minimal	Soft + hard errors	Yes
ECCs [70]	Yes	Yes	Data	- (data)	Very low to Medium	Soft + hard errors	Yes
CRCs [90]	No	Yes	Data	- (data)	Very low	Soft + hard errors	Yes
AN Codes [12]	No	No	Logic and Data	INT/FP +, −	Low	Soft + hard errors, bugs†	No
Berger Codes [17]	No	Yes	Logic and Data	INT +, −, logic ops, shifts, rotators	Low	Soft + hard errors, bugs†	Yes
Residue Codes [152]	No	Yes	Logic and Data	INT/FP +, −, ∗, /, SQRT, FMA, rotators, shifts, logic ops	Low	Soft + hard errors, bugs†	Yes
Parity Prediction [136]	No	Yes	Logic and Data	INT +, −, ∗, /	Low	Soft + hard errors, bugs†	No

† **: Protects against bugs in ALUs**

is a single bit aggregated to a wider data word. An ’even’ parity bit is set if the binary data word has an odd number of ones. Similarly, an ’odd’ parity bit is set if the the data word has an even number of ones. Parity codes are able to detect all single and all odd number of faults. In memories, parity codes are normally used for register files and reorder buffers as well as in low-level write-through caches, to allow for recovery methods [118, 199].

Common Error Correction Codes (ECC) use Hamming [70] and Hsiao [73] codes that provide single-bit correction, double-bit detection (SEC-DEC). Higher reliability levels are achieved with Double-bit Error Correction Triple-bit Error Detection codes (DEC-TED), symbol codes and b-adjacent codes. SEC-DED and DEC-TED [24, 72] allow the detection and correction for any possible location of faults, whereas symbol and b-adjacent codes [25, 39, 40] are restricted to adjacent locations. It can be observed that the higher degree and flexibility of correction, the higher overhead they pay [195]. SEC-DED and DEC-TED are normally used in second and third level caches which allows low-latency encoding/decoding while at the same time providing a tolerable overhead. On the other hand, extreme symbol based codes are used to provide Chipkill [48] support for DDR2 and DDR3 devices [83] and GDDR5 [32].

Cyclic redundancy checking codes (CRC) are interesting codes because of their high degrees of error detection and their simplicity. They are suited for the detection of burst errors in communication channels. The CRC code is the remainder of the division of a data word by a generator polynomial of length n where all its coefficients

are either a 0 or a 1. $n-1$ zero bits are attached at the end of the word and then it is divided by the CRC polynomial: the resulting $n-1$ bits are attached to the original word, constituting the CRC word. A CRC word is valid if it is exactly divisible by the polynomial. All errors in an odd number of bits will be detected. All burst errors of length n can be detected by any polynomial of degree n or greater.

Error Coding Techniques for Control Logic

Whereas regular memory arrays can be efficiently protected through coding techniques, few control logic blocks such as arithmetic and logic functional units are amenable for error detection through arithmetic codes. Arithmetic codes are preserved by correct arithmetic operations, that is, a correctly executed operation taking valid code words as input produces a result that is also a valid code word.

AN codes, represent an integer N multiplied by a constant A [12]. Before an arithmetic operation is performed on two numbers N_1 and N_2, each of them is multiplied by a constant A. Let R be $R = AN_1 * AN_2$, if R is not exactly divisible by A then at least one error has occurred. This invariant is true for many operators including integer and floating-point addition/substraction [170]. AN-codes are non-separable: the data part and the code part are processed and combined together, and the data value cannot directly be read from the code word. Input values are already multiplied by A, and as a consequence since the functional units operate with already transformed values, the required circuitry is increased and gets more complex.

Berger codes [17] are separable codes. The check bits are a binary representation of the number of zeros contained in the data. A codeword is valid if the value of the check bits equals the numbers of zeros in the data word. Since no mathematical property is exploited it is not easy to generalize Berger codes to any arithmetic or logical operation. Practical implementations exist for integer addition/substraction, logic operations and for shifters and rotators [107, 108]. Also, there are research proposals for multipliers and dividers [106], as well for FP operations [104]. However, the area and delay overheads can be non-acceptable in some designs [12].

Residue codes are separable codes. They have been deeply studied in the literature due to their cost-effectiveness, their capability in handling most operations as well as their levels of fault tolerance. Given two input values N_1 and N_2, and R being the chosen residue value, the arithmetic property $((N_1 \bmod R) \bullet (N_2 \bmod R)) \bmod R = (N_1 \bullet N_2) \bmod R$, holds true for most of the common operations $\bullet$ implemented by microprocessors. R is called the pre-selected residue base. Academia has proposed for most of the common operations effective residue functional units (functional blocks computing the expected results's residue from the operands' residues). Residue func-

tional units have been studied for integer arithmetic operations, including addition, substraction, multiplication, division and square root [96, 141, 152, 153, 169, 189, 210]. Similar ideas have been also applied to logical operations, including AND, OR, XOR operations [19, 58, 125, 177, 213] as well as shifts [74]. Residue functional units for single precision and double precision floating point operations (such as addition, substraction, multiplication, division, multiplication with addition and multiplication with substraction) are also supported [46, 68, 76, 77, 105, 124]. Residue checking has also been generalized for vector (SIMD) operations [21, 77]. Generally, residue codes are smaller than the Berger codes, and the residue functional units require much less area than Berger functional units [96, 105]. Residues are not intrusive into existing designs: execution units are left as they are, while the computation of the residue of the result is done in parallel without impacting the delay of the original circuit. Moreover, given its separability feature, for the cases where a residue functional unit is not cost-effective (for example for small logic blocks), the separability allows the designers to skip the checking of the operation, while still providing error detection for the source operands and computability of the result's residue, as opposed to non-separable schemes. Recent products like the Fujitsu's SPARC64 V processor also adopted 2-bit residue checkers for the multiply/divide unit (as well as parity prediction for ALUs and shifters) [8]. IBM z6 incorporates residue codes in its pipeline [216]. IBM Power6 - 7 [156] incorporate residue checking for decimal and binary floating-point units, and vector ones [102].

Parity can be used to protect arithmetic units by means of parity prediction circuits. Parity is not predicted in a speculative way, but actually in a safe, deterministic manner. The generation of the result's parity bit is based on the source operands parity and some properties of the carry chains of the computation to be checked. Parity prediction circuits have also been proposed for addition, substraction, division and multiplication [134, 135, 136]. Whereas parity prediction is very amenable for small adders and small multipliers, residue codes are cheaper for large multipliers and adders (as most commercial processors have) [134]. Moreover, despite parity prediction is a separable code, its circuitry invades the existing design to forward existing carry signals towards the redundant parity computation block.

3.3 Circuit-Level Techniques

Reliability can also be enhanced from a circuit-level perspective.

Upsani *et al.* [202] propose deploying a group of acoustic wave detectors [69] on silicon together with a hybrid hardware/firmware system that detects and locates the occurrence of a transient fault based on solving a system of equations that capture

the wave arrival times to the detectors. Fault detection latencies for caches are much shorter than traditional schemes where the detection is performed upon consumption time. However, the localization of the fault can take several cycles because solving the equations requires computation.

Asynchronous built-in current sensors (BICS) are circuits connected to the power lines of memory cells to monitor current variations caused by hard faults or soft errors. A BICS commonly monitors the memory cells belonging to the same column and shares the power bus of the column [60, 203]. BICS are often combined with parity codes associated to data words, so that whenever a parity mismatch occurs for a word, the affected bit can be deduced and corrected. Combining BICS and parity requires low area overhead, but they increases the correction latency. BICS can also protect combinational logic [132], but the overheads make them quite unpractical.

Circuit-level techniques can also be applied to provide mitigation against transient faults. There are two approaches: either increasing the capacitance of the node, thereby reducing the spectrum of particle charges that can upset the circuit, or using cells commonly referred to as radiation-hardened. Despite increasing the capacitance reduces the SER [86], it also negatively affects performance and power. As an alternative, capacitance can be added to the most vulnerable gates in a logic circuit [61, 122]. This selective approach is not able to provide complete error coverage while requiring less than twice the area. Radiation hardening, on the other hand, is applied to storage SRAM cells, latches or flip-flops. A radiation-hardened cell uses extra transistors that restore the state of the original circuit in the case of a particle strike (by maintaining a redundant copy of data) [30, 120, 161]. However, the area, power, and delay costs of radiation hardening approaches (often exceeding 100%) make these solutions impractical and are just used for specially selected circuits.

Some hardware schemes have been proposed to detect faults arising from variations and degradation. Razor [54] is a solution for detecting and correcting timing faults. Razor modifies existing stage flip-flops with a shadow latch so that they perform a double sampling: once with the normal clock, and another after a fixed delay. The skewed clock is set so that the shadow latch can capture most worst-case delays arising from degradation or variations. Upon a timing violation, the main latch and shadow latch will have different values, and the shadow latch is considered to hold the correct value. Razor must guarantee that there must not be a short path that can cause the output of the logic to change before the shadow latch latches the previous output. Razor can only guarantee correctness when the range of possible delays for a circuit output ($delay_{max} - delay_{min}$) falls within a window of size $T - hold$ where T is the clock period and $hold$ is the output latch $hold$ time. Hence, Razor's main drawback is its requirement of redoing all designs to guarantee a minimum short-path

Table 3.3: Comparison of circuit-level techniques

	Detection	Recovery	Sources Of Failure	Performance, Power Costs	HW Cost (complexity)	Domain	Adequacy
Accoustic detectors [202]	Yes (unprecise location)	No	Soft errors	No slowdown, Minimal power	Minimal	Data + Logic	Wide
BICS [203]	Yes	Yes (adding parity)	Soft + hard errors	No slowdown, Minimal power	Low	Data	Wide
Increased Capacitance [61]	No (mitigation)	No	Soft errors	High	High	Data + Logic	Non-critical paths
Radiation Hardening [161]	Yes	Yes	Soft errors	High	Very High	Data	Vulnerable circuits and non-critical paths
RAZOR [47]	Yes	Yes	Soft errors, delay faults	Speed-up, power benefits	Medium	Logic	Circuits with limited range of delays

delay, which is a huge overhead in design time and cost. Razor was later extended [47] to detect transient faults within the flip-flop and in the combinational logic.

3.4 Software-Level and Hybrid Techniques

Software techniques to detect faults are very popular in the literature because of their simplicity and low-cost. These techniques provide certain reliability levels for processors implementing no fault tolerance techniques.

Since the early 80s, a myriad of ad-hoc techniques have been proposed to detect control flow errors concurrently with the processor operation. SIS (Signatured Instruction Streams [171]), PSA (Path Signature Analysis [131]) and CSM (Continuous Signature Monitoring [219]) are some of these hybrid software-hardware approaches that provide detection of control flow errors in the fetch and decode logic. In signature checking schemes, checking is done at the hardware level [110] but compiler support is needed to appropriately partition the source code into sets of code sequences. Code blocks ('nodes') are selected to have one entry point and one or more exit points. Control flow checking is checked within nodes but not for edges among nodes. The compiler generates as many signatures as the number of exiting points. Every signature starts at the node entry point and includes all younger instructions (in program order) up to the exiting instruction, capturing its flow under a fault-free scenario. Each of these signatures are then embedded into the object code. The processor fetches instructions normally, and regenerates these signatures. A special

Table 3.4: Comparison of software and hybrid techniques

	Recovery	**Sources of Failure**	**Purely SW**	**Performance/ Power Costs**	**Detection Latency**	**CPU Featuring**	**Coverage**
SIS [171], **PSA** [131], **CSM** [219]	No	Soft + hard errors, bugs	No	Low	Unbounded	None	CF errors (Just intra-BB's)
CFCSS [137]	No	Soft + hard errors, bugs	Yes	Low	Unbounded	None	CF errors (Just inter-BB's)
EDDI [138]	No	Soft errors	Yes	Very High	Unbounded	None	Whole core
SWIFT [158]	No	Soft errors	Yes	Very High	Unbounded	None	Whole core (LDs/STs still vulnerable)
CRAFT [160]	No	Soft errors	No	Very High	Unbounded	None	Whole core
SRMT [211]	No	Soft + hard errors	Yes	Very High	Unbounded	CMP	Whole core
SWIFT-R, TRUMP [34]	Yes	Soft + hard† errors, bugs†	Yes	Extremely High	Unbounded	None	Whole core

† **: Protects against hard errors or bugs in data operands and ALUs**

hardware compares the dynamic signatures against the embedded ones. This increases the pressure in the instruction cache and thereby degrades the performance. Moreover, the error detection latency is unbounded and instructions can update the architectural state before they are completely checked.

CFCSS [137] (Control Flow Checking by Software Signatures) is a pure software solution for control flow checking. It extends the coverage of previous approaches to verify that control is transferred to a valid successor basic block. However, CFCSS does not ensure that the correct direction of the conditional branch is taken (a branch should fall-through but it actually takes the path, or viceversa). CFCSS extends the program to perform the instruction sequencing checking, based on the *a-priori* allowed transition among nodes. This solutions is able detect design bugs related to the instruction sequencing among/within nodes, flags generation/consumption and branch execution. However, all these techniques cannot validate the control logic in charge of performing efficient control flow recovery for out-of-order processors.

Redundant execution schemes can also be implemented in software. Software-based redundant execution allows executing the redundant instructions within the same thread hardware context. Since a fault in the fetch logic can affect both, the outcome of control-flow instructions must also be redundantly compared. Performance overheads are bigger than for purely hardware versions because there are more points of checking and extra instructions to do so.

The seminal work by Oh *et al.*, EDDI (Error Detection by Duplicated Instruc-

tions [138]), duplicates the instructions and adds extra checking instructions, so that both copies of the program uses different registers and memory locations and they do not interfere among them. Stores and branches are considered as comparison points. Checking is done through regular instructions. EDDI incurs a significant memory overhead because the memory space and all instructions are replicated.

SWIFT (SoftWare Implemented Fault Tolerance) [158, 160] is an evolution of EDDI. SWIFT splits the register file for the two execution streams but does not duplicate the memory because it assumes it is protected through coding techniques. This avoids the duplication of stores instructions, but forces stores, loads and branches to be synchronization points. SWIFT builds on top of CFCSS and extends its control flow protection to also ensure that the correct branch directions are actually taken: SWIFT has some points-of-failure for non-replicated instructions that cannot be avoided by pure software solutions: a fault between the checking of the operands of a load/store and their use can happen and cause silent data corruption.

CRAFT (CompileR Assisted Fault Tolerance) is a hybrid solution [159, 160] that augments SWIFT with hardware structures to cover against SWIFT vulnerability gaps. CRAFT introduces two hybrid techniques to protect store and load operations. For the treatment of stores, a special buffer is introduced for keeping the store addresses and data to write: store instructions simply update this structure, whereas the replica store instruction accesses the buffer entry for checking. The buffer commits then the entry to memory. Loads are handled equivalently: replica loads will access this structure for checking the address and obtaining the data. Globally, CRAFT performance is better than SWIFT because the use of hardware structures removes the need for some of the comparison instructions.

Wang *et al.* introduced Software-based Redundant Multi-Threading (SRMT [211]) for transient fault detection. SRMT uses the compiler to automatically generate redundant threads so that they can run on general CMP systems. Those threads communicate and synchronize through a reserved memory space.

Software-level reliability techniques can also provide fault recovery. Chang *et al.* propose several software recovery solutions [34]: SWIFT-R, intertwines three copies of the program and adds majority voting before stores and loads. TRUMP (Triple Redundancy Multiplication Protection) executes two copies of the program, but one of them implements software AN-codes. The AN version inherently offers error detection and globally, correction.

3.5 Industrial Validation Techniques

Due to the inherent imprecision of the photolithographic process, imperfections are accidentally introduced during manufacturing. Post-silicon *structural testing* is hence aimed to uncover these faults. The gate netlist is used as the golden reference model, and is used by ATPG (Automatic Test Pattern Generation) software to infer optimized test sequences and golden outputs that are ultimately probed in the real silicon. However, structural testing faces limiting 'controllability' and 'observability' issues. To counteract these limitations, engineers do normally incorporate *design for testability* (DFT) features into processors. DFT techniques enable ways to write and sample flip-flops and latches of circuits, so that the combinational logic can be tested by ATPG-generated input vectors.

Some examples of DFT techniques include *scan chains*, *hold-scan flip-flops* and BIST circuits [2, 94].

By using scan chains, engineers can inject an arbitrary state value through a special I/O pin across the flip-flops constituting the scan chain, enabling a finer controllability of the circuit. Engineers can also freeze the execution, scan out the values of the flip-flops forming the chain through the special I/O pin and analyze it, hence gaining in observability. However, *scan chains* increase the area and the interconnection complexity. Furthermore, the scan-in process introduces interferences and non-determinism into the regular processor activity, and the scan-out process does not allow resuming execution afterwards, because the scan elements are assumed to be unstable. *Hold-scan flip-flops* are an evolution of the *scan chains*, and avoid having to stop the processor execution while obtaining and inserting the scan outs and scan ins, respectively. Despite hold-scans provide higher validation flexibility, their overhead is very high. As a consequence, a small subset of the core logic is covered by this technique. Furthermore, an efficient usage of scan-based techniques is extremely dependent on the validators experience. The data capture of a scan chain is performed by externally asserting the enable pin, and is not triggered by internal conditions.

At-speed Built-in Self-Test (BIST) adds special circuitry into existing hardware units to test them using their own hardware (at full speed). The advantages are their low cost (no dependence on costly Automated Testing Equipment - ATE) and their capability to perform tests during processor lifetime. However they achieve a reduced coverage due to their inability to test big and interacting components. Furthermore, they cannot detect transient or intermittent faults.

Whereas *structural testing* targets the detection of manufacturing errors, *functional post-silicon validation* aims at detecting and debugging design bugs. Func-

tional post-silicon validation is driven by directed and parameterized random tests. Direct tests are often written by component designers and their expected outcome is normally known a-priori. Parameterized random tests allow uncovering more errors than directed tests, because they introduce non-determinism in the timings of internal activity but require slow architectural simulators to obtain the golden output. These tests allow uncovering complex errors because they introduce non-determinism in the timings of internal activity.

FRITS [146] (Functional Random Instruction Testing at Speed) is a software-based technique that enables automated test generation. DFT equipment is used to inject an automatically generated binary (the kernel) into the caches of the design under test. The kernel repeatedly executes pseudo-random or directed instruction sequences. For every execution of the kernel, the results (register file and memory) are compressed and stored in the cache. The DFT equipment then extracts those results and an architectural simulator validates the generated results. However, FRITS cannot validate the uncore because kernels cannot generate any cache miss because address-data-control pins are under the tester control. In order to keep the pace of the validation flow, big server farms are used to simulate the random tests and obtain their expected outcome, so that they can be compared against the prototype's results. The biggest concern is the dependence on large server farms for generating FRITS kernels and golden outputs for validation [163].

The debug process starts by collecting microarchitectural traces through traditional DFT techniques or by means of specific tracing (logging) technologies [1, 103], as a way to increase the internal observability. An analysis of the traces is conducted to reveal the succession of events that lead to the bug manifestation (either a system crash, deadlock, data corruption, or as a violation of some internal assertion).

In the case of tracing technologies, they use dedicated on-chip buffers to temporally store the internal activity as well as expensive machinery to obtain the data out of these buffers. However, they are rarely implemented because of their high area overheads. Tracing technologies have been used in the industry to increase observability. Intel Generic Debug eXternal Connection (GDXC) [220] was introduced in the Sandy Bridge processor [220] to debug the uncore. GDXC allows selecting and forwarding messages across the ring interconnect to an external analyzer for diagnosis purposes. This solution has many drawbacks. GDXC is restricted to uncore, and no internal observability is possible for the activity in the core. Moreover, since GDXC just forwards internal activity to an external analyzer through slow processor I/O pins, intense activity periods can clog the I/O pins or can cause the dropping of trace packets. Also, it is extremely costly when using logic analyzers, and on-chip buffering is constrained to trace the internal activity for only few hundreds of cycles, while bugs

may visibly manifest tens of thousand cycles later (e.g. hangs, data corruption).

One of the first steps to simplify the capture of traces is finding a way to freeze the system close to the bug root-cause, before the bug activation. Cycle breakpoint support [18] is used in combination with DFT techniques. Breakpoints can be defined by programming custom checks or assertions [103, 206] on available signals.

An important issue here is the capability to efficiently reproduce failures on the RTL model of the system under test. The biggest challenge is achieving a complete synchronization between the behavior of the silicon and the the RTL simulator. The PSMI (Periodic State Management Interrupt) methodology [178] is a well-known solution for this. PSMI periodically asserts a special processor pin, while the processor executes a test or application, forcing it to enter into a manually crafted software handler. The handler first dumps the content of the architectural state into memory, making it visible on the processor system bus (Dump point). Then, it pre-defines a state in some arrays and state machines (Sync point). The dumping allows validators to obtain checkpoints for initializing the RTL model, close to the error manifestation point. A successful state transfer provides full internal observability in the RTL model while eliminates the need of executing the whole test in that slow model. However, the main problem consists in the interference introduced by the Synch points. The Synch points in the handler can cause the elimination of the manifestation of bug that originally existed. More importantly, PSMI involves an extremely iterative hand-tuning process which requires deep knowledge of the underlying microarchitecture.

CHAPTER 4

EVALUATION FRAMEWORK

This Chapter describes the evaluation framework that we have used to implement and evaluate our proposed solutions. The baseline fault tolerance capabilities of our baseline processor model are also presented.

4.1 Benchmarks, Tools and Simulators

The global structure of the evaluation framework is depicted in Figure 4.1. It integrates a processor timing simulator that runs a set of benchmarks. Connected to the performance simulator, our framework integrates a fault injection model that allows computing the error coverage of the proposed solutions, based on the dynamic behavior of the simulated processor. Similarly, our infrastructure also incorporates a power and area model that based on the microarchitecture of the processor is able to compute the power and area overheads of our proposals. Section 4.1.1 details our benchmarks, Section 4.1.2 describes the performance (timing) model and the baseline processor that is simulated, Section 4.1.3 delves into our fault coverage evaluation model and Section 4.1.4 discusses the area, power and delay models.

4.1.1 Benchmarks

The focus of this thesis is reducing the vulnerability of advanced out-of-order superscalar processors, while at the same time minimizing the area, power and performance impact. Hence, one of the most suitable benchmarks are those for common high-performance and commodity systems. We use the SPEC CPU2000 suite, that is an industry-standardized CPU-intensive benchmark suite [190].

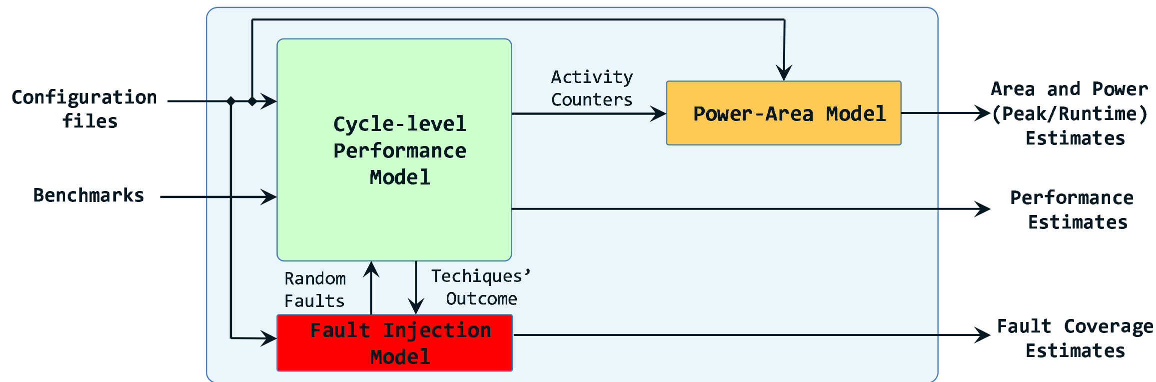


Fig. 4.1: Global structure of our evaluation framework

The benchmark suite consists of 12 integer programs and 14 floating point programs. For the sake of generality we have used both the integer and the FP programs in the whole thesis.

Table 4.1 shows the description of the benchmarks used across all our studies. We have used the *ref* input data set. The benchmarks have been compiled using Intel® ICC 8.0 C and Fortran Compilers, using the -O3 and -O4 flags, respectively.

To simulate significant parts of the programs, we have used the PinPoints [147] tool. As noted by the authors, we have configured it with 250M instruction slices, a parameter of max clustering of 10, and have picked the region with the highest weight. The first 150M instructions are used to warm-up the caches and the rest of micro-architectural blocks. In the rest of the thesis, the default number of instructions simulated is 100M (the rest of the 250M slice), except otherwise stated.

4.1.2 Timing Simulator

All the techniques presented in this thesis have been evaluated using an execution-driven microarchitectural simulator that runs Intel® x64 applications. The base simulator we have used is a detailed industrial path-finding performance simulator developed by the Intel® Barcelona Research Center team.

The simulator is highly configurable and is able to model advanced out-of-order processors that include register renaming and physical register files. Table 4.2 shows the values for the most important configuration parameters for all the evaluations, unless a different configuration is stated. At a high-level, the microarchitecture resembles the one found in the Intel® Core™ Sandy Bridge processor [79, 85, 130, 173], but does not totally correspond with it at every detail. For example, parameters such as the size and bandwidth of structures, etc. have been scaled down to account for

Table 4.1: Benchmarks used to evaluate our solutions

INT programs

Benchmark	Description
`bzip2`	Data compression utility
`crafty`	Chess program
`eon`	Ray tracing
`gap`	Computational group theory
`gcc`	C compiler
`gzip`	Data compression utility
`mcf`	Minimum cost network flow solver
`parser`	Natural language processing
`perlbmk`	Perl
`twolf`	Place and route simulator
`vortex`	Object Oriented Database
`vpr`	FPGA circuit placement and routing
`ammp`	Computational chemistry
`applu`	Parabolic/elliptic partial differential equations

FP programs

Benchmark	Description
`apsi`	Solves problems regarding temperature, wind, velocity and distribution of pollutants
`art`	Neural network simulation; adaptive resonance theory
`equake`	Finite element simulation; earthquake modeling
`facerec`	Computer vision: recognizes faces
`fma3d`	Finite element crash simulation
`lucas`	Number theory: primality testing
`mesa`	3D Graphics library
`mgrid`	Multi-grid solver in 3D potential field
`sixtrack`	Particle accelerator model
`swim`	Shallow water modeling
`train-galgel`	Fluid dynamics: analysis of oscillatory instability
`wupwise`	Quantum chromodynamics

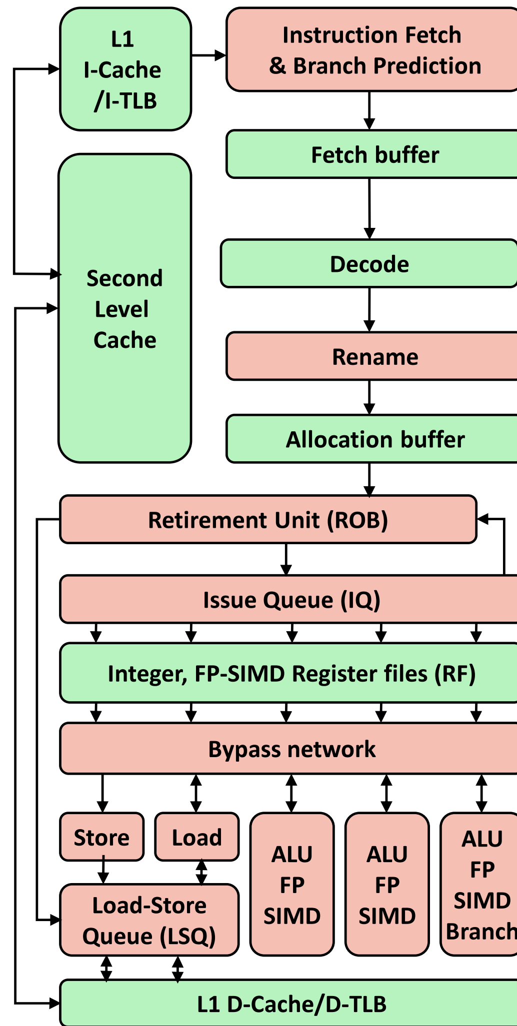


Fig. 4.2: Baseline processor microarchitecture. Light green blocks are protected by existing techniques

mobile segments.

Our evaluations focus on single-thread performance on a single core configuration, so we have scaled down the last-level cache accordingly.

The processor pipeline is shown in Figure 4.2: Intel® x64 instructions are fetched from the first level instruction cache, accessing the branch predictors if required. Then, *macro-instructions* are decoded into several *micro-instructions* (*micro-ops* [85]), following program order.

Micro-instructions are later sent to the rename logic, which is required to support out-of-order execution. Renaming instructions enforces the register dataflow specified by the programmer/compiler even though instructions may be executed not following original program order. After that, *micro-ops* are allocated in the Reorder Buffer, in

the Load-Store Queue if they are memory operations and in the Issue Queue. In the issue queue they wait until their operands are ready, so that they can be then issued to the execution ports for out-of-order execution.

When instructions are issued, they wake up the dependent instructions. When instructions finish their execution, they write-back their results into their allocated physical registers. At execution, unresolved branches find out if they were mispredicted or not. In case of misprediction the pipeline and rename table is recovered. Once executed, instructions send a completed signal to their ROB entries. Finally, the instructions at the head of the ROB commit, leave the pipeline and release microarchitectural resources. Store instructions access memory at this stage.

4.1.3 Fault Coverage Evaluation Methodology

The fault injection methodology in this thesis aims at modeling the faults caused by any source of error (transient error, intermittent error, design bug or other hard faults) in an advanced out-of-order processor pipeline, and study the response of the proposed techniques in detecting or diagnosing them.

From a circuit-level perspective, a fault can affect a stored bit in a sequential element or affect the transistors and wires of combinational logic blocks. However, modeling these faults requires a gate-level model of the processor pipeline. Even though gate-level modeling allows accurate measurements, microarchitectural-level fault injection regimes are more desirable from a design perspective [155]:

- **Simulation speed** Fault simulation at the gate-level is extremely time consuming. These models are very detailed, and their simulation speed is orders of magnitude lower than for performance simulators. Given that many fault injections are required for a high degree of confidence, simulating them at the circuit-level becomes almost impossible.
- **Reliability decisions during design path-finding** Fault injection at the circuit-level is not suitable for use during design phases. Early reliability estimates must be made in order to guide and adapt the design, in a similar manner as it is done with power or temperature budgets. This fact calls for reasonably accurate cost-effective methods to obtain error coverage metrics, and therefore, microarchitectural-level models (such as timing simulators) represent a sweet spot. Furthermore, abstract models are the ones available during these stages, and not circuit-level models.
- **Fault masking** Fault injection at a gate-level has the downside of fault masking [165]. Quantifying masking effects is critical when computing accurate (non-

Table 4.2: Simulator configuration

Parameter	Value
Frequency	2.8 GHz
Technology	32 nm
Voltage	1.1 V
Main Memory	DDR3-1600[‡], 48ns/54 for open/random RAM page + 27 cycles for load-to-use latency
Last-Level Cache (LLC)	2 MB, 16-way, write-back, 27 cycles load-to-use, 2 slices, 1 R/W port of 32B each. Runs at core f/V, 32B ring
Unified Second-Level Cache (L2$)	256 KB, 8-way, write-back, 12 cycles load-to-use, 1 R/W port of 64B
Data Cache (D$)	32KB, 8-way, write-back, 2 cycles hit[†] 2 R/W ports of 32B, 64B lines
Miss Status Holding Register (MSHR)	16 outstanding misses
Instruction Cache (I$)	32KB, 8-way, 3 cycles hit, 1 R/W port of 16B
Data/Instruction Translation Lookaside Buffer (DTLB/ITLB)	128 entries, 8-way, 25 cycles per miss
Branch Predictors	GShare [112] PHT-BTB 8K entries bimodal 4-way, 16-bit history, 16 entries return-address stack 14 cycles misprediction penalty
Decode width	up to 4 *micro-instructions*
Rename width	up to 4 *micro-instructions*
Allocator Queue (Alloc)	12 entries (*micro-instructions*)
Allocate width	up to 4 *micro-instructions*
Rename Tables (RATs)	1 frontend RAT, 8 checkpoint RATs
Issue Queue (IQ)	32 entries scheduler, connects to 6 exec ports
Issue width	up to 6 *micro-instructions*
INT Operations [*exec ports*]	ALU [0/1/5], LEA [0/1], Shift [0/5], Mult-Div [1], Jump Unit [5]
FP Operations [*exec ports*]	Adder [1], Mult-Div [0]
SIMD INT/FP Operations [*exec ports*]	SIMD INT: ALU [1/5], Mult-Div [0], Shift [0/5], Other [1/5] SIMD FP: Add [1], Mult-Div [0], Other [5]
Load-Store Queue (LSQ)	30 loads, 20 stores (up to 2 loads and 1 store per cycle)
Memory Operations [*exec ports*]	Load Address [2/3], Store Address [3], Store Data [4]
Register Files (RF)	128 INT, 128 FP-SIMD, 2 bypass levels
Reorder Buffer (ROB)	128 entries
Commit width	up to 4 *micro-instructions* (max. 1 non-bogus store)

[†] : +2 cycles for load-to-use latency due to address calculation.

[‡] : 10-10-10-28 tCL-tRCD-tRP-tRAS timings

> pessimistic) processor failure rates, but masked faults must be ignored when evaluating the error coverage potential of a fault-tolerance technique. A better approach is to directly model at a microarchitectural simulator non-masked anomalies or **failure scenarios** caused by faults at the circuit-level. These simulators are aware of several sources of masking. Instructions belonging to wrong paths or mispeculated, instructions with dead results and instructions suffering some types of logical masking can be identified and be avoided during the fault injection. The net result is that the incidence of unmasked faults is higher when using these models, resulting in a rigorous evaluation of the fault-tolerance techniques.

We use a fault injection approach where faults locations at the microarchitectural-level that end up manifesting in the same visible *failure scenario* are grouped together [155]. For example, faults in a register scoreboard entry, or in a shift-register, or in a select request or bid signal, or in the latency of producers, etc. can result in prematurely issued instructions. To do so, the pipeline stages and processor components described in Appendix A are thoroughly inspected to identify the high-level visible faults that can be modeled in a timing simulator, enabling a fast and reasonably accurate evaluation. We have used fault studies, such as Reddy's [155], to guide the finding of our particular *failure scenarios.* For fault locations not analyzed in previous works, we have conducted fault injection studies in order to understand the resulting manifesting failure scenario, and to reason about the conditions when they mask or manifest.

In each of the next chapters, we detail the different *failure scenarios* that can arise when faults affect the hardware involved in implementing the register dataflow, memory dataflow and control flow recovery logic. For each *failure scenario* we list the hardware components that, when faulty, can end up causing each type of failure.

For every considered failure scenario, 1000 effective faults are injected per benchmark. The fault injection is performed one-at-a-time during the first 10M instructions, in a random manner. Then, each experiment is allowed to run for 100M instructions, to let the fault manifest. An injection experiment is rejected (not effective) when the fault is masked. Masking happens when these conditions are satisfied: (i) the architectural state in the functional simulator is not corrupted (i.e the state matches the expected golden state), (ii) the functional simulator does not report an error (no assert in the benchmark is raised and no wrong exit status is returned by the simulated benchmark), and (iii) the watchdog timer (described in Section 4.2) does not trigger.

The timing simulator and the interface to the functional simulator have been deeply modified to support explicit fault injection. First, it has been extended to ex-

plicitly model micro-architectural structures that were originally implicitly modeled. This includes hardware blocks like the bypass network, the bypass-register file data, branch coloring fields for wrong-path tracking, logical register destinations, latency fields in the issue queue, ready fields, etc. In addition, the performance simulator has been modified to include buggy methods. The objectives are twofold: first, it allows supporting fault injection for hardware locations that cannot be explicitly modeled at a micro-architectural level, and second, by using buggy methods we can guarantee that the proposed solutions can cover against functional design bugs. Some examples include: buggy methods for the wake-up logic, select logic, load-store queue logic, instruction squashing logic, input multiplexors, and ROB walk logic.

For locations explicitly modeled in the performance simulator, faults are injected as single bit flips. For locations not explicitly modeled, faults are modeled as activation of the buggy simulator methods. The duration of the injected faults have no fundamental impact on the coverage of end-to-end schemes, as noted by Meixner [186]. For non-transient faults, instead of letting faults persist during the whole experiment execution, we have chosen a more pessimistic approach where they behave like "short intermittent" faults. This approach provides lower-bounds on error detection coverage for permanent faults, as the opportunity to detect them is limited by only one fault activation, and not be consecutive ones. It is important to note that by relying on spatial redundancy, permanent faults can be detected (the checked hardware is different to the hardware implementing error detection). Furthermore, design heterogeneity covers against design bugs (the checker logic is different to the checked logic).

Methods like AVF analysis [126, 128] have not been used because despite being suitable for computing estimates for SRAM and CAM structures, they cannot estimate the vulnerability for combinational logic.

4.1.4 Area, Power and Delay Evaluation Methodology

One of the objectives of this thesis is to satisfy the needs for efficient reliability solutions with minimal costs in performance, power and area, while at the same time providing the high reliability levels of traditional defect tolerance techniques. Therefore, area, power and delay studies also require specific evaluation tools and methodology.

We use an in-house [1] path-finding power, area and delay tool that models the processor micro-architectural blocks and units. This model allows driving power, area and delay analysis and takes into consideration the particular implementation

[1] Developed by the Intel® Barcelona Research Center team.

of specific micro-architectural blocks. For cache-like and array structures, our model is based on CACTI 5.3 [200]. For the rest of structures (such as combinational logic, wiring and clocking), our model ports and extends Wattch 1.0 [27]. As opposed to Wattch, our model works with new CACTI versions, interfaces into an advanced timing simulator and incorporates specific Intel®-internal values. An alternative model like McPAT [99] has not been used because it became publicly available and stable after we had begun evaluating some of our techniques. The models have been parameterized for a 32nm technology node.

Note that our model does not rely on costly and slow computer-aided design circuit tools (such as HSPICE), nor on electronic design automation tools. The reasons are twofold. First, the circuit-level implementation of our baseline processor was not available. And second, tools like CACTI and Wattch provide processor architects with power, area and delay modeling at abstraction levels above circuits and schematics. This enables the possibility to explore and cull the design space early on, using faster, higher-level tools [27, 200].

The power component also counts the number of times some predefined microarchitectural events occur. For example, we count the number of times a register is read or written. This is done for every major block in the micro-architecture during program execution. The peak power of individual units and these machine utilization statistics are used to calculate the runtime power dissipation. However, to evaluate the power overheads of our solutions, **we focus on peak dynamic power** [2]. Peak power numbers are obtained based on maximum activity factors and maximum peak energy-per-event. Peak power ends up defining the maximum power consumption of a processor and provides upper bounds estimates. Furthermore, this power metric critically impacts the reliability of the processor [191]. The power overheads we show are clearly pessimistic, as a consequence.

The main blocks that the model incorporates fall into these categories:

- **Array structures:** Caches, cache tag arrays, TLBs, branch prediction structures, rename tables, free lists, register files, the ROB, the issue queue payload RAM and register scoreboard, as well as the load-store queue payload RAM.

- **Fully Associative Content-Addressable Memories:** Issue queue wake-up logic, load-store queue memory checks.

[2] In CMOS processors, dynamic power consumption (P_d) is the main source of power consumption, and is defined as: $P_d = C * {V_{dd}}^2 * a * f$. C is the load capacitance, V_{dd} is the supply voltage, and f is the clock frequency. The activity factor, a is a value between 0 and 1 indicating how often clock ticks lead to switching activity on average.

- **Combinational Logic:** Decoders, renaming intra-bundle dependency checking, selection logic, functional units and ROB walk (RAT recovery) logic.
- **Data wires:** Result and bypass buses.
- **Global clocking:** Clock buffers, clock wires, etc.

The design, structure and sizing of micro-architectural blocks (described in Table 4.2) are used to derive their representation and parametrization in our power-area-delay model. A single high-level logical microarchitectural structure sometimes is represented as several components in the model. As an example, the issue queue is represented as a CAM memory and a RAM memory (modeled by CACTI), and as combinational logic and as wiring (modeled as in Wattch).

For array structures and CAM memories CACTI allows specifying a block configuration based on parameters such as: cache type (i.e. data arrays, data+tag arrays, and DRAM arrays), structure size, associativity, line size, number of read, write and read/write ports, technology, voltage, frequency, temperature, number of banks, output/input bus width, explicit tag size, tag and data access mode (i.e. fast, sequential, normal) and transistor type (high-performance, low stand-by power, low operating power, DRAM).

CACTI allows specifying optimization criteria and constraints in order to find a design that better suits the user needs. This allows the user to skip over many of the low-level details of the components being modeled and lessen the burden on the architect to figure out every detail. Configurations are evaluated by assigning a weight to each optimization criteria (delay, leakage power, dynamic power, cycle time and area), and the solution space is pruned based on maximum deviation with respect to the best solutions found during the process. Alternatively, the user can specify a design exploration criteria based on energy-delay (ED) or energy-delay square (ED^2).

ED^2 optimization criteria has been chosen for most blocks, as we target performance-oriented processors. Those blocks affected by our techniques are checked to meet the processor cycle time (*the target clock rate is used as a design constraint*). Those that are time-critical have been optimized by CACTI using other constraints. For example, the bypass network is time critical because it is routed over the functional units and the register files [140]. As a consequence, the register files have been optimized by prioritizing the area and dynamic power.

For combinational logic, data buses and clocking structures, our power-area-delay model is heavily based in Wattch. Next we provide details of several of our microarchitectural components.

- **Instruction Decoders:** In this case, we have used internal values from previous Intel ® products scaled by process technology and frequency.

- **Intra-Bundle Dependency Checking Logic:** Two parallel intra-bundle dependency checking blocks handle RAW and WAW dependencies. The area and power of each block is computed based on the number of comparators and their capacitance. Delay is assumed to be lower than the RAT access time, as noted by Palacharla et al. [140].

- **Functional Units:** In this case, we have used internal values from previous Intel ® products scaled by process technology and frequency.

- **Write-Back Bus and Bypass Network:** The number of wires equals to the data width times the number of stacks that produce a value within all the execution ports multiplied by the number of stacks of the same type. The result bus power is computed based on specific internal wire capacitances from the technology and clock frequency. The area of the functional units and the register files are used to compute the result bus length [140], which is multiplied by the capacitance per unit of length. Tristate buffers are used to model input multiplexors.

- **Select Logic:** We follow the approach of Wattch (and McPAT): we model it as a tree of cascaded arbiters, where each arbiter in the tree handles up to four selection requests. Select requests traverse the tree down to the root arbiter, and a bid answer traverses up to a leaf arbiter which eventually selects an instruction. An arbiter is modeled as OR gates and as priority encoders. Globally, as many trees as the number of execution ports are modeled. The centralized select logic that manages resource conflicts is included in our framework.

- **Wake-up Logic:** We follow the approach of Wattch (and McPAT): the CAM search operation serves as the wakeup logic for the issue queue. We model both the tag drive (including the power and area to write new tags) and the tag match components. This includes the buffers to drive the destination tags, taglines, comparators, wordlines, bitlines, matchlines and OR gates to produce the readiness bits [139].

- **LSQ Checking Logic:** The CAM search operation also models the detection of *store-to-load forwarding* and *memory ordering violations* scenarios. The full length of addresses are used in CAM matches. The load and store queue CAM memories are modeled separately but in as a similar way as in the previous item. Our power and area model also accounts for the comparators that handle

age information and the priority encoders to choose the youngest but older forwarding stores, as opposed to Wattch.

- **ROB Walk Logic:** The modeling is handled similar to the second item. In this case, only WAW dependencies are handled, but given that the RAT can be recovered by undoing or redoing register mappings, two independent blocks are needed. They are modeled as in the second item. In addition, we also account the power and area needed to store and access the register mapping fields (that are kept at separate ROB banks).

- **Global Clock:** We enhance Wattch's H-tree model where the global clock signal is routed to all portions of the chip using equivalent length metal wires and clock buffers. The model also accounts for the bits required to latch each stage, and uses the processor area number computed by CACTI or obtained from internal Intel® values, as opposed to Wattch.

4.2 RAS Features in the Baseline Processor

This section lists the error protection mechanisms that are included in the baseline processor. Modern advanced out-of-order processors include few simple RAS features to protect critical structures from an area and vulnerability perspective.

Therefore, our baseline processor also includes simple error code protection mechanism in several structures. Figure 4.2 shows in light green the arrays that we assume protected by an error code scheme, and in light red the blocks that cannot be protected by existing mechanisms (components heavily implemented by means of combinational logic). Cache structures such as the *instruction cache*, *data cache*, *second-level cache* and the *TLBs* are protected by error detection-correction codes. Whereas *TLBs* are protected by means of parity, the caches are protected by ECC codes that support error correction. The second-level and LLC caches are protected by stronger SEC-DED schemes.

Other storage structures, like buffers, are protected by simple error detection codes. The *fetch buffer* is protected by parity codes that are extracted from the instruction cache. Other arrays like the *allocation buffer*, or the entries in the *issue queue* payload RAM are protected by explicitly generated parity bit (they are wide, and non-mutable). Faults can be simply detected by checking the information code, and non-permanent faults can be recovered by means of the pipeline-flush and restart mechanism provided by the baseline core. The register files are protected by a parity bit, and the parity generators and checkers reside at the inputs of the write and read ports, respectively.

As most processors, ours also includes a watchdog timer that monitors the hardware for signs of deadlock. Specifically, the watchdog timer monitors the ROB: if no instructions commit for an extremely long time that exceeds a predefined threshold, then the watchdog timer reports that an error has occurred, the pipeline is flushed and execution is re-started from the instruction at the head of the ROB.

Instruction control flow and allocate logic is protected by this watchdog timer and a special checker residing in the ROB [49, 155]: the Program Counter (PC) of each instruction is checked against the following instructions PC to ensure correct program order. Sequential committing instructions add their length (recorded at decode time) to the retirement PC and branches update the retirement PC with their calculated PC. Comparing a committing instructions PC with the retirement PC will detect discontinuities. Detected failure scenarios include: wrong PC generation, unintended instructions (dis)appearing in the frontend, overwriting instructions in the frontend queues, instructions being moved forward in an unordered manner, allocation in wrong ROB/LSQ/issue queue entries (potentially overwriting).[3]

Decoders logic and PLAs (Programmable Logic Arrays) are protected using the method described in [37], due to their large area.

[3]Allocating an instruction in a wrong ROB entry is detected by means of the PC checker. If an instruction is wrongly allocated in the issue queue / LSQ (overwriting an existing unexecuted one), the ROB complete bit of the overwritten instruction entry will not be activated, leading to a deadlock.

CHAPTER 5

REGISTER DATAFLOW VALIDATION

5.1 Introduction

Whereas classical error detection mechanisms based on re-execution were amenable for high-end segments where high area, power and/or performance penalties could be tolerated, the radical increase in raw error rates calls for fault tolerance mechanisms that can be deployed in commodity segments. New requirements include negligible area, power and slowdown overheads, while at the same time providing the high reliability levels of traditional defect tolerance techniques.

On another axis, whereas critical SRAM structures (such as caches and register files) are already protected with parity or error correction codes in most commercial processors, limited research efforts have been devoted to design cost-effective error detection strategies for the wrapping control logic of high-performance microprocessors. Currently it plays a critical role for the whole microprocessor correct operation, and it represents a significant portion of the die area and testing and validation costs.

In this chapter we propose a low-cost online *end-to-end* protection mechanism that protects the control logic involved in the register dataflow. This includes the *rename* tables, *wake-up* logic, *select* logic, *input multiplexors*, *operand read and writeback*, the *register free list, register release, register allocation*, and the *replay* logic. Our proposal is based on microarchitectural invariants (applicable to any processor design) and allows detecting multiple sources of failures, including design bugs.

End-to-end protection is based in generating a protection code at the source where vulnerable data is generated, sending the vulnerable data with the protection code

along the path, and checking for errors *only* at the end of the path, where it is consumed. Faults caused by any logic gates, storage elements, or buses along the path are detected at the consumption site. Instead of individually checking specific low-level microarchitectural blocks, our solutions verifies high-level functionalities whose implementation is scattered across many components.

The centerpiece of the proposed solution is a *signature-based* protection mechanism. The implementation cost and the coverage provided by the protection framework depends, primarily, on the signature width and, secondarily, on how signatures are generated. We propose and thoroughly assess different multiple ways of generating and handling signatures. For each policy, we discuss the error coverage and their cost in area and power.

In this chapter, we also study how to extend fault coverage to cover against errors in register values. To achieve this, we first exploit the potential of residue codes to build an end-to-end self-checking microarchitecture that computes with encoded operands. Then, we describe how this end-to-end residue checking system can be smoothly embedded into our register dataflow end-to-end protection scheme, in order to amortize costs. The net result is that functional units, load-store queue data and addresses, register file storage and data buses are also protected at a low cost.

The rest of the chapter is structured as follows: Section 5.2 reviews how faults in the dataflow may manifest. Section 5.3 reviews our framework for a dataflow self-test mechanism. Section 5.4 overviews an end-to-end residue coding scheme and explains how to integrate it with our proposal. In Section 5.5 we propose and assess different policies for generating and handling the signatures. Section 5.6 discusses how the different signature generation policies impact the overall coverage and processor overheads. Section 5.7 reviews some relevant related work. We summarize our main conclusions in Section 5.8.

5.2 Register Dataflow Failures

Faults in the dataflow could result into different architectural errors. We classify them by error location, and depict some possible faults that caused them.

1. *Selection of wrong inputs*: The input multiplexors and the selection logic that chooses the input operands from the bypass/register file and feeds the functional units may select a wrong input, causing an incorrect data to be consumed.

2. *Wrong register file access*: A read access to the register file may provide a wrong data value. The causes might be: (a) a “register read” access that reads from a

wrong entry, (b) a "register write" access that writes into a wrong register (in this case, the readers will suffer the consequences), or any other cause.

3. *Premature issue*: A prematurely issued instruction will consume a wrong data value. Some causes are: (a) incorrect operation of the wake-up logic, (b) incorrect operation of the select logic, (c) incorrect assignment of the latency of a producer instruction (the consumers suffer the effects), etc.

4. *Wrong tag*: An instruction may depend on a wrong instruction (i.e. through a wrong register *tag*) and consume its data. The causes might be: (a) incorrect contents in the rename table, (b) wrong access to the rename table, (c) faults in the rename dependence checking inside the rename bundle, or (d) corruption of a *tag* tracking a register dependence in the issue queue.

5. *Data stall in the bypass network*: If the latches placed in the different levels of the bypass do not latch a new value (e.g. due to a missing or delayed clock signal) it may happen that it gets stalled with an old data value.

6. *Register free-list misuse*: If the register free list does not operate correctly (including wrong register release and allocation), the register *tags* might get corrupted. We also consider the situation when the old or current mapping in the ROB may get corrupted. The net result is that a physical register may simultaneously be the destination location for two different instructions.

7. *Load replay errors*: If the replay logic does not work properly, it may neither identify nor reissue all the instructions that depend on a load that misses in the data cache. [1] As a consequence, there could be silent commitment of bogus values, potentially corrupting the architectural state.

8. *Deadlock*: A deadlock will happen if the oldest instruction waits (incorrectly) for a *tag* that is not in-flight and, hence, cannot trigger a wake-up. This is a sub-case of a "wrong tag" with a different microarchitectural result.

Faults that result in a deadlock can be easily detected by means of a watchdog timer, already implemented in many current microprocessors [7, 78] and in our baseline processor (Appendix A). However, the other faults result in instructions operating with a wrong data value, and require more sophisticated detection mechanisms. These faults are the target of our protection mechanism.

---

[1] Also, these instructions could be replayed due to a TLB miss, bank conflicts in the data cache, or write port conflicts in the register file

5.3 End-to-End Dataflow Validation

This section describes our proposal for an efficient mechanism to perform online validation of the register dataflow logic. We first explain the signature-based protection scheme and the different high-level steps it is composed of. We then comment on how register dataflow faults can be recovered when detected by our technique. We finally analyze the required hardware changes needed to support our proposed solution.

5.3.1 Signature-Based Protection: General Idea

We propose a novel technique that is based on marking every data value flowing through the pipeline with a *signature*. A *signature* is a token associated to a chunk of information. Whereas codes such as residue, parity or ECC are a function of the data they are associated with, signatures in its general definition do *not* depend on any property of the protected information.

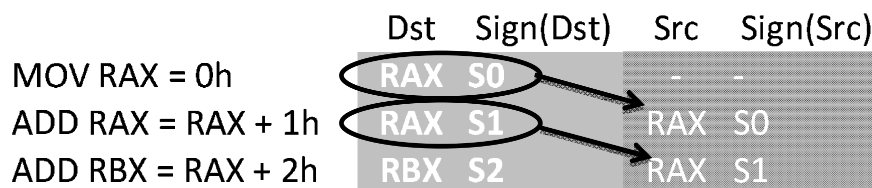


Fig. 5.1: Register signature assignment among dependent instructions: an example

Our online validation technique is exemplified in Figure 5.1, which shows three instructions with their corresponding destination and source signatures. Each operand, including sources and destination, receives a signature that allows tracking the dataflow. Each *a priori* source signature is compared with *a posteriori* signature obtained during execution. The signature obtained during execution can be considered as the result of the control logic that is protected, potentially faulty. If both signatures mismatch, an error is detected. Otherwise, the destination signature is written back and forwarded along with the data to any potential consumer. This way, the producer-consumer loop is continuously monitored through a hand-shake mechanism.

We now precisely describe the main signature-based protection scheme dividing it into three steps: signature assignment, signature flow and signature check. The complete flow is depicted in Figure 5.2.

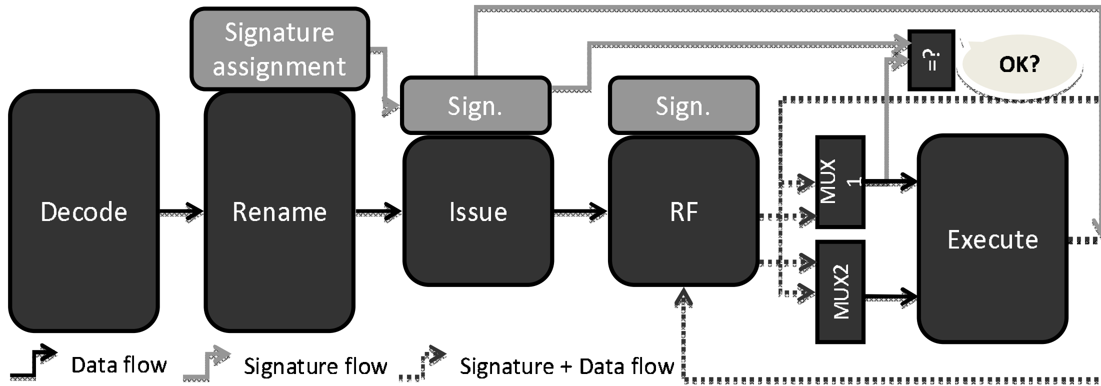


Fig. 5.2: End-to-end signature checking: extensions in the core dataflow

Signature Assignment

Signature assignment is performed in two steps, similar to register allocation. All instructions that generate a register value obtain a destination signature. Immediate values are also assigned a signature. Since the number of immediate operands is low, we will concentrate in the rest of the chapter on register operands.

Destination signature assignment can be performed as soon as the logical registers destinations have been identified. Without loss of generality we assume this is performed during the rename stages, although this could be done during decode time. Once instructions have been renamed, the destination signatures are stored into the rename table together with the allocated register physical *tags*. Since signatures can be arbitrarily generated for destination registers, the hardware in charge of generating them is independent of the proposed design framework. Different signature assignment policies will be discussed in Section 5.5, and it will be shown that they have a direct impact in complexity and coverage.

Source operands receive their corresponding signatures at rename time from the rename table. [2] In a fault-free scenario, the rationale is that such signatures must match the destination signature of the producer instruction of that operand. Overall, every instruction flows after the rename stage with 3 signatures (as shown in Figure 5.3 and Figure 5.2): the signature of the data it will produce ($Sign(Dst)$), and the signatures of the producers of its two operands ($Sign(Src_1)$ and $Sign(Src_2)$).

[2]Except for one signature assignment policy, as it will be discussed in Section 5.5.

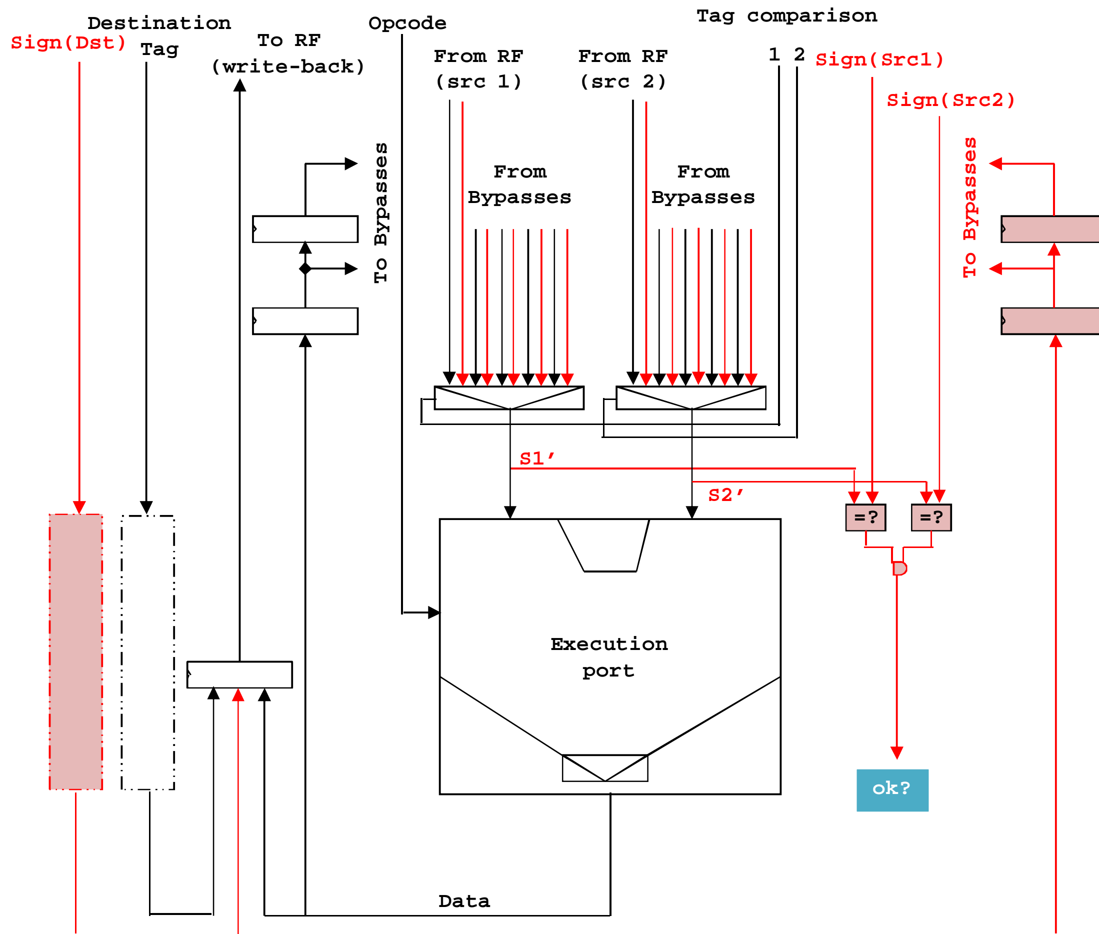


Fig. 5.3: End-to-end signature checking: extensions in the backend logic (signatures hardware is shown colored

Signature Flow

Figure 5.2 depicts how data and signatures flow through the pipeline. After being renamed, the issue queue receives instructions with two source and one destination signatures. At the issue queue, instructions wait until they are ready for execution. Once an instruction issues, it reaches the multiplexors at the input of the functional units ('*Execute*' box). Such multiplexors select among the different data paths the value to consume (i.e. a bypass path or register file entry in our design). Therefore, since data and signatures travel together, the operand selection also selects the signature that flows with it.

Once the instruction finalizes execution, it sends the result with its corresponding signature ($Sign(Dst)$) to all register datapaths in order to reach all potential consumers. Again, for our baseline processor, this includes the bypass network and the

register file. For other schemes where speculatively produced values are stored in the ROB, the signatures would also be temporally stored in that structure. By sending the signature through the bypass and storing it in the register file and all dataflow structures, subsequent instructions that consume it (either correctly or wrongly) can perform the flow check.

Special treatment must be paid on loads: processors speculate whether loads will be able to obtain the data from the cache (will hit), in order to allow dependent instructions to issue back-to-back, without paying the latency to wait for actual hit status. A misprediction implies that dependent instructions consume bogus data and must therefore be identified, nullified and re-issued later to consume the correct data.

We extend the signature flow step to also protect against faults in the *load replay system*. We transform the existing signature infrastructure into a poison propagation network (a *serial verification scheme* [89]) where corrupted signatures correspond to instructions depending (directly or indirectly) on a load that misses in the data cache.

The corruption process starts with the detection of a load missing in cache: its destination signature is immediately corrupted and propagated. A small circuit called *spoil circuit* performs this. The directly dependent instructions will observe the corrupt signature upon execution, and this will cause them to recursively corrupt and forward their own destination signature to inform any potential indirect consumer.

Signature Check

As shown in Figure 5.3, the dataflow checks are performed after the input multiplexors select the data that the functional unit will use. At that point, we have the information required to validate that everything went right: (i) the signatures associated to the source data values that the functional unit will consume, and (ii) the expected signatures for the values the functional unit *should* consume, which were obtained at the rename stage and read out from the issue queue upon issue.

Two signature comparators are placed next to the functional units (one for each source operand); if any of the signatures mismatch an error is detected unless it corresponds to an instruction that needs to be replayed by the *load replay system*.

To filter out signature mismatches that correspond to instructions that must be replayed due to load latency mispeculations, an additional action is taken. In parallel to the signature checking, we use the signal '*replay?*' provided by the *load replay system*, which tells whether an instruction should be replayed or make forward progress. Notice that in case of a load latency misprediction, we expect a mismatch in the signatures. Based on the output of the signature comparators and based on the '*replay?*' signal provided by the existing *load replay system*, Table 5.1 indicates

Table 5.1: Register signature mismatches corresponding to real register dataflow errors

Signatures mismatch?	**Replay Is Needed?**	**Flag Error?**
no	no	no
no	yes	***yes***
yes	no	***yes***
yes	yes	no

the cases when a signature mismatch corresponds to a real error scenario. Note that both signals do not share any control logic and hence, are redundant. It can be seen that in case both signals agree, nothing is done. Otherwise, a failure is detected and a recovery action is attempted.

5.3.2 Failure Recovery

We rely on flushing the pipeline to restore correct state in the event of an error detection. This mechanism is already provided by the processor to handle scenarios like the recovery of wrong memory ordering detection in the load-store queue, or to handle branch misprediction recovery.

Re-execution will start from the instruction observing a signature mismatch. By flushing the pipeline we can recover from multiple sources of failures that affect the dataflow and values, as long as the faults alter speculative state. However, faults affecting the architectural state that are later exercised, consumed and detected by an instruction cannot *always* be recovered by flushing the pipeline. The reason is that the causing instruction may have already left the pipeline. These failure scenarios correspond to faults that result in wrong data being written back to the register file, written to a wrong register file location, or simply corrupting the data stored in the register file. The same applies for faults in the rename table: wrong updates (entry or tag) to the rename table, or simply bit upsets in the rename table cannot *always* be recovered by flushing the pipeline. For these cases, we must rely on existing recovery mechanisms like checkpointing (recall Chapter 2) to roll-back the processor to a pristine state. Otherwise, we can just simply flag a machine check exception and guarantee that no silent data corruption has occurred.

After the pipeline flush and during re-execution the faulty hardware will not be exercised for those failures that can be effectively recovered. However, for those faults that have a permanent nature, we would like to disable the affected hardware. How the faulty block is disabled or replaced is out of the scope of this work.

5.3.3 Microarchitectural Changes

We describe now the changes in the processor stages and hardware structures required for implementing the proposed mechanism. Figure 5.3 and Figure 5.2 show a close view of the hardware changes required in the backend and in the core. A detailed list of the microarchitectural changes follows (assuming B bits per signature for a total of $M = 2^B$ signatures).

- **Rename stages.** The modifications in these stages depend on the signature assignment policy. We will assess them in Section 5.5.
- **Allocation stages.** Additional space in the instruction queues to hold the source and destination signatures (3 signatures of 3 bits).
- **Issue queue.** The CAM memory or bit matrix [166] to track register dependences is left unchanged. Hence, the delay of the critical wake-up/select loop is not affected. Conversely, we enlarge the payload RAM. Each entry in the payload RAM will hold extra fields for keeping the signatures of the sources and the destination (3 signatures of B bits). Input allocation write ports and output issue read ports are resized accordingly.
- **Register files.** Additional space and wires to store the signature per register (B bits per register).
- **ROB.** It will depend on the the control-flow recovery implemented and the signature assignment policy. We will assess it in Section 5.5.
- **Bypass network.** Additional wires to carry the signature of each register value (B bits per value), and wider input multiplexors at the inputs of the execution ports.
- **Execution units.** Signature checkers that compare the signature of the register value received at the execution units with the expected signature (2 comparators of B bits).
- **Write-back network.** Additional wires to carry the signature of each value (B bits per value).
- **Replay logic.** The added hardware to implement the error detection for the replay logic is just the *spoil circuits*. The *spoil circuits* can be implemented with just an XOR inverter, since we only want to corrupt the signatures. We require one *spoil circuit* for every functional unit that propagates a destination register and signature, including the load execution ports.

Large part of the hardware overhead mainly comes from (i) the additional register file storage, (ii) the additional wires in the bypass network, and (iii) the additional fields in the ROB. Therefore, the signatures should be as narrow as possible.

5.4 End-to-End Register Value and Dataflow Validation

This section starts with an overview of residue coding and an end-to-end implementation that protects the register data values and computation. Next, we detail how to integrate it with our end-to-end register dataflow protection mechanism, in such a way that their overheads are shared but the detected failure scenarios are expanded. Microarchitectural changes needed to support our combined solution are also analyzed. We finally exemplify how our combined end-to-end signatures and residues technique works together to detect failures in the register dataflow logic, values and computation.

5.4.1 Implementing End-to-End Residue Checking

Arithmetic codes have been deeply studied in the past for protecting data but also for protecting arithmetic and logic functional units (computation). They are based on attaching a redundant code to every data word. While data is protected by verifying the associated redundant code, arithmetic operations are protected by operating in parallel the data and the codes. This is, arithmetic codes are preserved by correct arithmetic operations: a correctly executed operation taking valid code words as input produces a result that is also a valid code word. Several arithmetic codes exist (see Section 3.2), such as AN codes, Berger codes, residue codes and parity codes.

We choose residue codes [11, 58, 96] to build a system where register values and computation is covered against errors. Among the different available separable arithmetic codes, the size of a residue code is much smaller than the size of a Berger code, and also the residue functional units require much less area than Berger functional units [96, 105]. Compared to parity prediction, residue codes are less invasive and cheaper for wide multipliers and adders [134].

Residue codes are based on the property that the residue of the result of an arithmetic operation can be computed from the residues of the operands as well as through a modular division of the result. Given two input values N_1 and N_2, and R being the chosen residue value, the arithmetic property $((N_1 \bmod R) \bullet (N_2 \bmod R)) \bmod R = (N_1 \bullet N_2) \bmod R$, holds true for most of the common operations '$\bullet$'.

Figure 5.4 shows a typical implementation of how residue checking works. The computation $\otimes$ is performed independently for both the regular data (operating A and

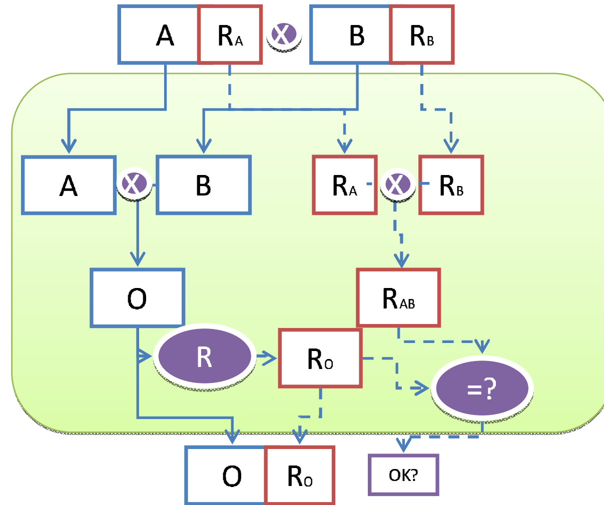


Fig. 5.4: Concurrent error detection with residue codes

B and producing O) and the redundant codes (operating R_A and R_B and producing R_{AB}). Then, in order to verify that both the data values A and B as well as the functional unit operation are correct, the redundant code of O is computed through function $R(O)$ and compared against R_{AB}. A mismatch indicates an error.

If R is in the form of $R = 2^k - 1$ for some k (for example R being 3, 7, 15, etc.), the residue code is called low-cost, because it allows a simple calculation of the residue value.[3] It is important to note that low-cost residues leave one value of the code unused (specifically, the value 2^k). The reason is that residues of the form 2^k cannot be used, because any fault affecting the word at position i, where $i \geq k$, will remain undetected. From a fault coverage perspective, if multiple faults add or substract a value by a multiple of $2^k - 1$, the faults will be undetectable (faults that alias back into the same residue value). A modulo-3 residue can detect not only all single-bit errors, but also most of 2-bit errors. When using a low-cost residue, burst faults of up to $k-1$ bits are guaranteed to be 100% detectable [11, 13, 213]. We choose $R = 3$; previous works [105, 141, 189] show that the implementation costs are rather small. It will be discussed in Section 5.6.

The research community and the industry have proposed, for most of the common operations, effective residue functional units (this is, functional blocks computing the expected results's residue from the operands' residues).

Residue functional units have been studied for integer arithmetic operations, in-

[3]The residue of an n-bit number is computed by dividing the binary number into k-bit chunks, and then summing these numbers through modulo-k addition. This allows the implementation of the residue encoders to be extremely simple, because no division or multiplication is needed [57, 210].

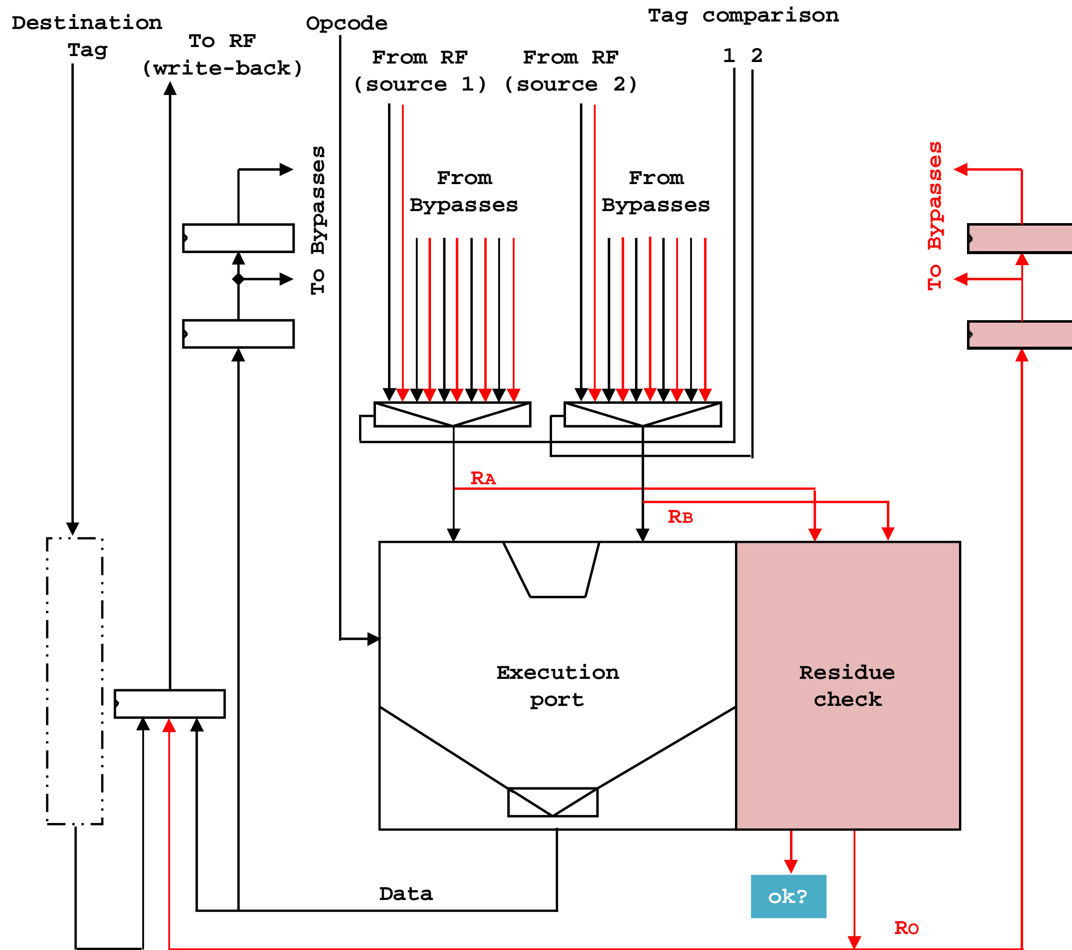


Fig. 5.5: End-to-end residue checking: extensions in the backend logic (residue hardware is shown colored)

cluding addition, substraction, multiplication, division and square root [96, 141, 152, 153, 169, 189, 210]. Similar ideas have been also applied to logical operations, including AND, OR, XOR operations [19, 58, 125, 177, 213] as well as shifts [74]. Residue functional units for single precision and double precision floating point operations (such as addition, substraction, multiplication, division, multiplication with addition and multiplication with substraction) are also supported [46, 68, 76, 77, 105, 124]. Residue checking has also been generalized for vector (SIMD) operations [21, 77].

The separability of residue codes simplifies the implementation of the checking component. Residues are not intrusive into existing designs: execution units are left as they are, while the computation of the residue of the result is done concurrently without impacting the delay of the original circuit. Moreover, for the cases where a residue functional unit is not cost-effective and is not implemented (for example

for small logic blocks), the separability allows the designers to skip the checking of the operation, while still providing error detection for the source operands and computability of the result's residue through function $R(O)$.

There are two different possibilities for embedding a residue code in a self-checking system: residue codes can just be applied locally inside the functional units, or the complete system computes with encoded operands [96, 119].

During the beginning of the arithmetic code era, residue codes were applied locally inside the functional units. This basic design option is commonly referred to as a "self-checking system" [209]. In this design, the residues of the source operands are computed before they are fed into the residue functional unit, possibly introducing extra delay in the computation and checking part.

Forty years later, Iacobovici extended the concept to out-of-order processors where the complete processor computes with encoded operands [96, 119] and baptized this kind of residue protection as "end-to-end residue checking" [75, 76]. Figure 5.5 shows an implementation of such end-to-end residue checking scheme. Residue codes are calculated where data is originated: (i) loads from the data cache, and (ii) output from the functional units. Residue codes flow through the bypass network, and are stored in the register file. This way data is protected in an end-to-end fashion: from the point it is originated, to the point it is consumed. Notice that for this implementation, we substitute parity with residue coding, since both protect the data. Correctness of functional units is achieved by the residue checkers placed next to them. Furthermore, this design option not only avoids adding residue generators to compute codes on-the-fly for the source operands, but also minimizes the delay introduced.

5.4.2 Integrating Signatures with Residues

Comparing Figure 5.3 and Figure 5.5, and the detailed hardware modifications listed in Section 5.3.3 and Section 5.4.3, one observes that end-to-end signature checking and end-to-end residue checking implementations have pretty much the same hardware requirements. Therefore, we adopt the end-to-end residue checking design and propose to merge the calculated signatures attached to values with the residue values flowing through the backend of the processor.

We share the hardware infrastructure and amortize costs for implementing both error detection techniques simultaneously: we encode a new residue value that is a function of the original residue and the signature of the destination register. Similarly, each encoded residue value is decoded back to the original residue using the signature of the corresponding source (obtained at rename time).

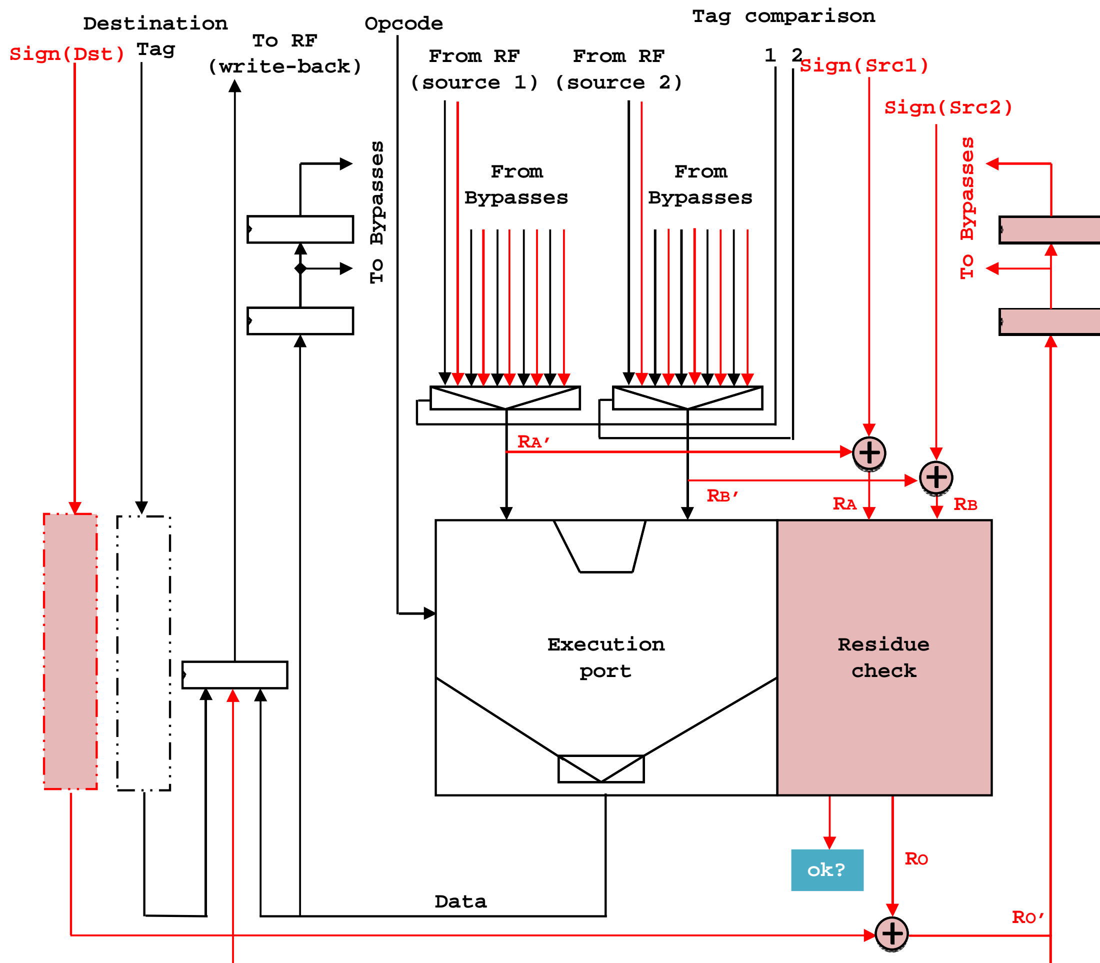


Fig. 5.6: Combined end-to-end signature and residue checking scheme: extensions in the backend logic (added hardware is shown colored)

The transformation function must be easy to implement. Besides, it has to be possible to construct the inverse function, so residues can be recovered. An extremely simple and fast function of this type is the bitwise XOR function (whose inverse function is also the XOR function). For example, given a residue of 01 and a destination signature of 11, a XOR-based encoding will forward a transformed residue value of 10 to the register dataflow paths. Reversely, given a transformed residue value of 10 and a source signature of 11, a XOR-based decoder will provide an original residue value of 01.

For those functional units implementing a concurrent residue functional unit, there is no need to keep the signature comparators. Specifically, we leave the residue checkers and remove the signature checkers. The rationale is that if there is an error in any of the different end-to-end paths that we protect, we will not be able to decode a correct residue, and the residue checker will suffice to flag that an error has

been detected. Conversely, for those functional units lacking residue functional units, we need residue generators for each incoming source operand value and a residue comparator for each incoming source residue value (after it has been decoded back using the expected signature). A residue generator for the produced value will also be needed.

Special attention must be paid to execution ports lacking ALUs. For example, the store data execution unit may just access the register file and write back the value and transformed residue value into the Load-Store Queue. In that possible scenario, we must extend these execution ports with residue decoders, in order to obtain the real residue (not the transformed one). It will be the residue generators and checkers of the consuming execution ports that will detect a possible failure at retire time, increasing the failure coverage therefore.

The whole encode/decode process is depicted in Figure 5.6. When an instruction is executed it writes back `Data` into the register file. Then, its residue `Ro` is XOR-ed with the destination signature `Sign(Dst)` of the instruction writing back (assigned at rename time). This encoded residue `Ro'` will be written back into the register file and will travel through the bypass network together with its associated data.

A consumer instruction requires the correct signature to retrieve the original residue `Ro`. Consumers use the signatures received from the rename stage (`Sign(src1)` for the left operand and `Sign(src2)` for the right one) to decode the input residues (R'_A and R'_B). If an error happens in the dataflow, the decode process will generate an incorrect residue. If an error in a data value happens, the residue decoders will obtain a residue that does not correspond to the expected for the the wrong input value, and the residue checker will detect the error.

When combining the signatures with the residues, the implementation of the *spoil circuits* (needed to protect the *load replay system*, as described in Section 5.3.1) also changes. Instead of corrupting the destination signatures, we corrupt the residues. We spoil residues by using the invalid residue value `11`: residue functional units are modified so that when one of the sources is this invalid value, the output will also be the invalid residue. This way all instructions depending on the missing load will observe a wrong output residue.

5.4.3 Microarchitectural Changes

Assuming we need R bits per residue and B bits per signature, and being $K = max(R, B)$, the mechanism requires the following hardware modifications:

- **Rename stages.** Same as for our signature checking scheme (Section 5.3.3).

- **Allocation stages.** Same as for our signature checking scheme (Section 5.3.3).

- **Issue queue.** Same as for our signature checking scheme (Section 5.3.3).

- **Register files.** Additional space to store per each register its transformed residue (K bits per register).

- **ROB.** Same as for our signature checking scheme (Section 5.3.3).

- **Bypass network.** Additional wires to carry the transformed residue of each value (K bits per value). In addition, wider input multiplexors to obtain the proper transformed residue per operand are required.

- **Execution units.** For every functional unit, we need a signature decoder $\bigoplus$ for each source operand so that its transformed signature can be converted into a residue (two XOR functions operating on K bits each). In addition, for every functional unit, a residue unit that operates with these incoming residues, a residue generator for the produced value by the functional unit, and a residue checker that validates (compares) both redundantly generated residues.

  For each execution unit that cannot operate with the incoming residues (no residue functional unit implemented), we need residue generators for the incoming values, residue comparators (to compare against the pre-computed ones) and a residue generator for the produced value (if any). Note that this design does *not* need comparators for the source signatures.

  Finally, a signature encoder $\bigoplus$ is also needed to encode the produced residue with the destination signature of the instruction being executed.

- **Write-back network.** Additional wires to carry the transformed residue of each value (K bits per value).

- **Data cache.** A residue generator for every load port.

- **Replay Logic.** Residue functional units are modified in such a way that whenever they observe an invalid residue as an input, they produce an invalid residue as an output. For every data cache read port, we add a *spoil circuit* that produces an invalid residue in case of a miss.

- **Load-Store Queue.** Additional space to store the residue for the data and address per entry ($2 * R$ bits per entry).

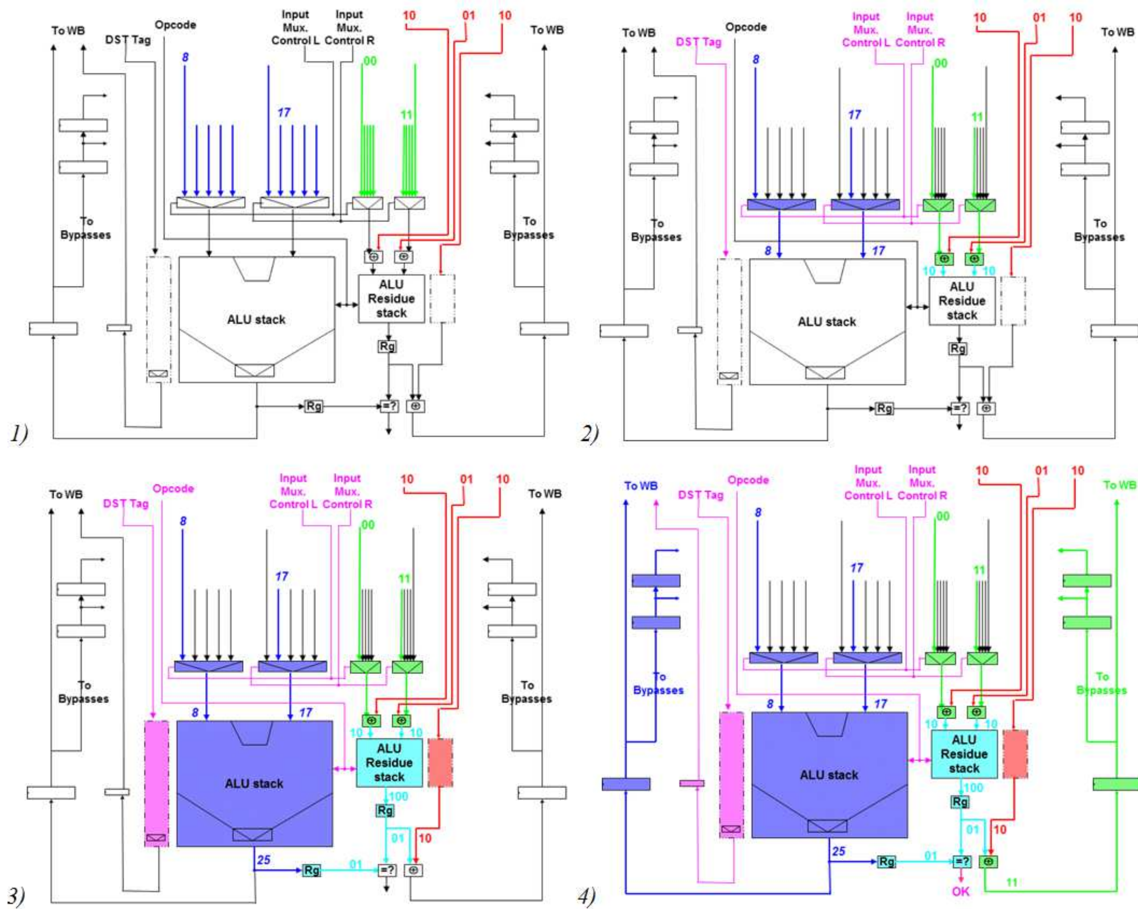


Fig. 5.7: End-to-end signatures and residues operation: fault-free scenario example

5.4.4 Examples

For the sake of clarity, we provide three examples on how our combined end-to-end signatures and residue works together to detect failures in the register dataflow.

Example of fault-free operation

We begin with a working example where we show how residues and signatures work together to validate the register dataflow in the common case of fault-free operation. In this first example, we take a look on how instruction $pr7 \leftarrow pr1 + pr4$ executes. Table 5.2 shows for the two source operands ($pr1$ and $pr4$) the value they contain, the corresponding residue, the signature used to encode the residue, and the result of this encoding. Last row shows the outcome of executing the instruction (for physical register destination $pr7$). Step-by-step operation is shown in Figure 5.7.

Figure 5.7(1) shows how the bypass-RF operands values (in dark blue) and their

Table 5.2: Values, residues, signatures and combined residues-signatures for fault-free example

	value	*residue*	*sgn*	*sgn*⊕**residue**
pr1	8	*10*	*10*	*00*
pr4	17	*10*	*01*	*11*
pr7	25	*01*	*10*	*11*

corresponding transformed residues (in green) arrive to the functional unit input multiplexors. The expected source signatures (`10` and `01`, in red) and the assigned destination signature (`10`, also in red) are retrieved from the issue queue upon instruction issue. Figure 5.7(2) depicts how the the input operand values (8 and 17, in dark blue) and the input transformed residues (`00` and `11`, in green) are selected based on the existing *Input Mux Control* signals (in pink). Furthermore, the selected transformed residues are decoded by the two ⊕ blocks using the expected source signatures, producing the expected input residue values (`10` and `10`, in light blue). In Figure 5.7(3), the input residue values `10` and `10` are consumed by the residue functional unit (*ALU Residue Stack*) to produce the expected residue for the value being produced by the functional unit (*ALU stack*). The residue functional unit produces residue `100` that is reduced by residue generator *Rg* to `01` (in light blue). Concurrently, the functional unit sums the input operand values to produce the resulting value (25, in dark blue). This value feeds a wider residue generator *Rg* that produces the associated residue value `01` (in light blue).

Finally, block *=?* in Figure 5.7(4) successfully compares both residues (*OK* signal in pink asserts). The produced value (25) is written back to the register file (*To WB* path, in dark blue), and flows through the bypass network (*To Bypasses* paths, in dark blue too). In parallel, the produced residue (`01`, in light blue) is encoded with the destination signature (`10`, in red) by means of a ⊕ block. The transformed residue (`11`, in green) is also written into the register file (*To WB* path in green) and flows through the bypass network (*To Bypasses* paths) to any potential consumer.

Example of *Selection of Wrong Inputs*

Next example depicts how *selection wrong input* scenarios can be detected when combining signatures and residues. Specifically, Figure 5.8 shown an incorrect generation of the multiplexors control signals. In this case, instruction $i_3 : pr0 \leftarrow pr4 + pr7$ should grab its operands values from $pr4$ (produced by instruction i_1) and $pr7$ (produced by instruction i_2), but the multiplexors signals make the instruction to wrongly obtain the operand in the right from $pr1$ (produced by instruction i_0).

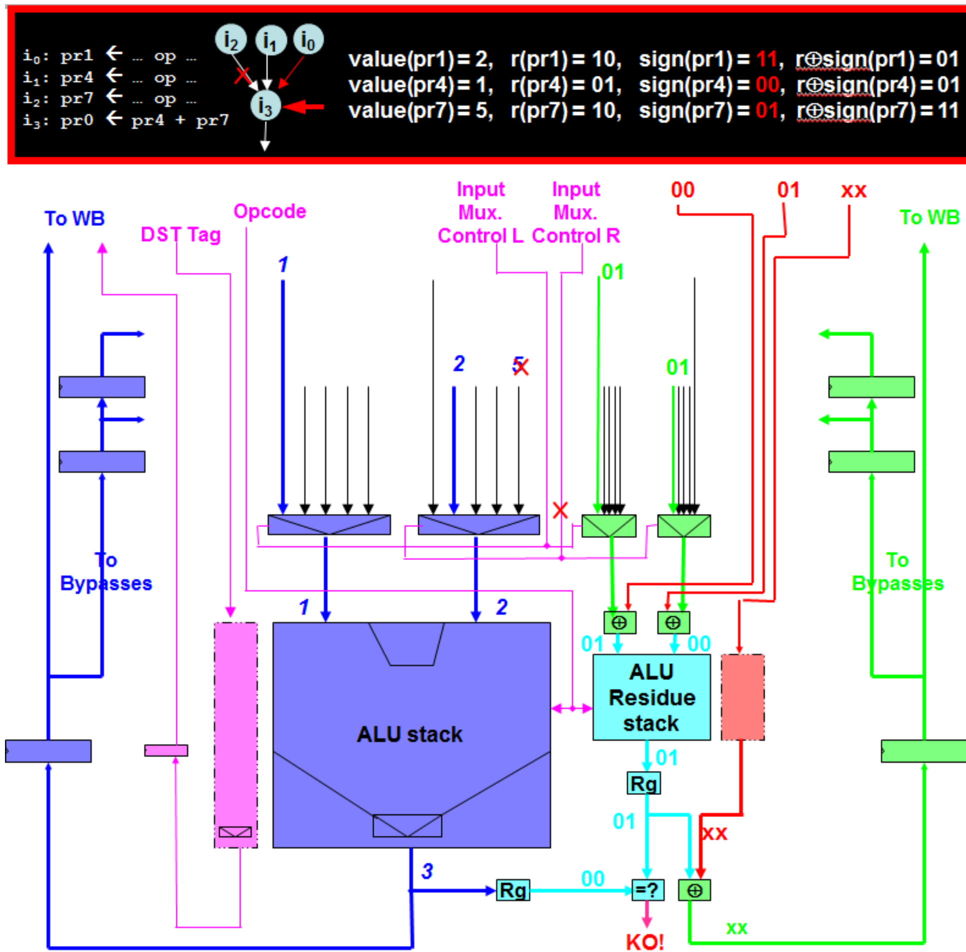


Fig. 5.8: End-to-end signatures and residues operation: *Selection of wrong inputs* example

However, during rename time instruction i_3 was given the expected (correct) source signatures (00 for pr_4 and 01 for pr_7). When instruction i_3 executes, the transformed residue from pr_1 will be decoded using the signature used to encode the residue of pr_7, rather than the signature used to encode pr_1 (which is 11). As a consequence, the $\bigoplus$ block will yield a wrong register value. Upon computation, the expected residue of the generated value (residue 00 for value 3) will not match the computed residue (01). Hence, an error is detected by the residue checker and reported.

Example of *Wrong Register File Access*

Figure 5.9 shows how our combined scheme would detect an error for case (3) *Wrong Register File Access*. For this particular example, we assume that we want to execute instruction $i0 : pr1 \leftarrow pr2 - pr3$. The register file shows on the left hand side, the

stored value prior to execution and the transformed residue value. To its right, for every physical register we list its residue and the signature used to encode it.

Let's assume that due to an error, instead of reading physical register $pr3$, we read $pr1$. As a result, instead of reading out the value 2 with the encoded residue 11, we read out value 7 with the encoded residue 10. As a result, when decoding 10 with the signature that we obtained at rename time 01, we obtain 11 instead of 10. The residue checker does the rest, signaling an error because the residue of the substraction of the operands is different from the residue obtained operating with the incoming residues (by the residue functional units).

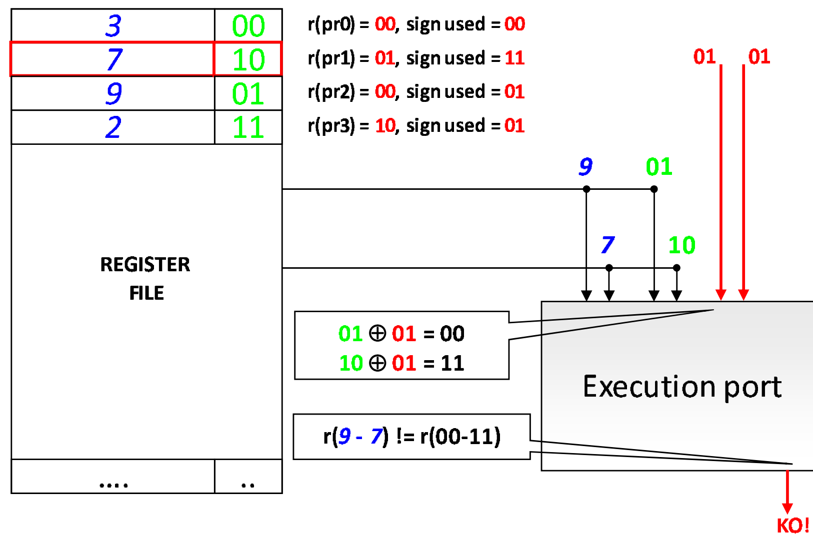


Fig. 5.9: End-to-end signatures and residues operation: *Wrong Register File Access* example

5.5 Signature Generation Policies

Given a signature, the probability to match another signature will depend on the total number of signatures and the way they are generated/assigned. An error may not be detected if the signature observed when there is an error is the same one as the expected one (i.e. *aliasing*). When using B bits to encode the signature, assuming they are uniformly distributed and used, the average case probability of having aliasing is $\frac{1}{2^B}$. Therefore, the expected average-case coverage in this case would be $1 - \frac{1}{2^B}$. For example, if $B = 2$ then fault coverage would reach 75% and for $B = 3$ it would be 87.5%.

This is true as long as signatures are evenly used. However, the way we use the signatures will depend on the generation mechanism; for instance, one policy may tend to use a lot a given signature for a sequence of instructions that reuse a lot a

particular logical register in a short interval of time. In this case, some signature could be used more than the others (i.e. there is low signature variability), which would hurt coverage.

In the next subsections we propose five different paradigms of signature generating policies: Section 5.5.1 describes round-robin policies based on auto-incremented counters, a minimum in-flight signature policy that favors the least present signature is described in Section 5.5.2, a register policy based on the physical register *tag* representation is presented in Section 5.5.3, Section 5.5.4 shows a static policy that is based on the logical destination register, and an enhanced static policy that boosts coverage for the cases where the static policy underperforms is introduced in Section 5.5.5.

5.5.1 Round-Robin Policies

Round-robin signature generation policies are based on the value provided by auto-incremented counters. We have explored different flavors of such *round-robin* policies:

1. **Basic *round-robin* policy** (`RR`). We just use one modulo counter, that is checked and auto-incremented for every destination register. Although this policy is simple to implement, there is a high probability of repeating a signature for the same physical register because the physical registers are used round-robin (the free list works as a queue). This "wrap-up phenomenon" aggravates when the number of physical registers is a multiple of the number of signatures. This could decrease the coverage for situations such as (3) *Premature issue*: early issued instructions will consume the data from the register file stored by the previous producer. Since the physical register will have the same signature, the error will be undetected. In order to mitigate the "wrap-up phenomenon" we next propose *pseudo round-robin* policies that we describe in the next bullets.

2. **Minimum-based *round-robin* policy** (`MRR`). We have just one round-robin counter, but every cycle it is reset to the least-present signature in the register file. During renaming it will provide different signatures for each instruction in the rename bundle. This policy attempts to maximize the distribution of signatures across the register file.

   Notice that we require 2^B counters to keep track of how many live signatures we have of each class. The counter is incremented when signatures are allocated, and decremented at rename time when the old register mapping is read.

3. **Multiple *round-robin* policy** (`nRR`). Instead of having just one modulo auto-incremented counter, we use multiple of them. Then, each logical register is

statically paired to one of the counters, which can be done randomly, or manually based on profile information. In this way, we avoid the signature "wrap-up phenomenon" while maintaining the benefits of an homogeneous signature distribution. We named these pseudo round-robin schemes `DRR` when using 2 counters, and `QRR` when using 4 counters.

4. **Logical register *round-robin* policy (`LRR`)**. We have one round-robin counter for each logical register. It corresponds to the `nRR` policy where `n` is the number of logical registers. Signatures are given in a *local* manner: we maximize the randomness for each logical register, but diversity for a given cycle is not guaranteed across different logical registers, since they use different counters. This may decrease coverage for situations as (5) *Data stall in the bypass network*.

Hardware modifications

Round-robin policies generate signatures once the logical registers are known. Besides, signatures must be stored on the rename table, in such a way that the consumers can obtain their expected signature. On top of the modifications detailed in Section 5.3.3, we would require:

- **Rename stage.**
  - A different number of counters depending on the specific round-robin policy: 1 counter of B bits for `RR`, 2 counters of B bits for `DRR`, 4 counters of B bits for `QRR`, 2^B counters of $log_2(R)$ bits (where R is the number of physical registers) for `MRR`, or as many counters as logical registers of B bits for the `LRR` policy.
  - An additional B-bit field per register in the rename table to store the last assigned signature.
  - Wider multiplexors in the *operand override logic* to select the proper signature for every instruction register *tag*.
- **ROB.** Since the control-flow recovery of a branch misprediction is implemented using a ROB-walk mechanism, we require additional space and wires to store the signatures of the *old* and *new* register mappings (2 signatures of B bits per entry).[4]

[4]If the processor does not support rename state recovery for any arbitrary position in the ROB, but rather at fixed locations that have associated shadow checkpoint tables, no extra hardware would be introduced in the ROB.

5.5.2 Minimum In-Flight Use Policy

Round-robin policies work in an *incremental* manner. While this approach works for most of the cases, it cannot guarantee the balance in the usage of signatures. For example, it may happen that many long-lived physical registers may have assigned a small subset of signatures. While these physical registers are not released, they may create un unbalanced distribution of signatures. Therefore, we try to increase the balance by assigning all the time the signature with the minimum presence in the register file (`MIN`). In order to keep track of the usage of each signature, we use the same mechanism implemented in the `MRR` policy.

Whereas `MRR` provides different signatures for each instruction in a rename bundle, `MIN` can assign the same one for all of them. The good side of this policy is that it targets a high variability per physical register (i.e. it benefits case (3) *Premature issue*). However, signatures are given in bundles (many different physical registers close in time will have the same signatures since signature can be repeated while they are the less utilized), which hurts most of the other cases - especially case (1) *Selection of wrong inputs* and (5) *Data stall in the bypass network*.

Hardware modifications

Similar to `MRR` policy, the `MIN` policy generates signatures once the logical registers are known. Besides, signatures must be stored on the rename table, in such a way that the consumers can obtain their expected signature. The hardware requirements are the same as the ones described for the `MRR` policy (described in Section 5.5.1).

5.5.3 Physical Register Policy

The physical register policy assigns signatures based on the destination register *tag*. Specifically, we opt to use the modulo of the physical register tag as the signature (`MOD`). This approach simplifies the implementation because we do not need to keep track of the assigned signatures in the rename table; the assigned signatures can be obtained from the already existing physical register *tag* (available after renaming has been completed). However, faulty situations like (3) *Premature issue* and (6) *Register free-list misuse* would remain unprotected because *a physical register will always have the same signature.*

Hardware modifications

The implementation of this policy is very simple, since we do not need to keep signatures in the rename table. On the other hand, the modulo calculation starts once the physical register tag is known, which is at the rename stage. On top of the modifications detailed in Section 5.3.3, we would require:

- **ROB.** Since the control-flow recovery of a branch misprediction is implemented using a ROB-walk mechanism, we require additional space and wires to store the signatures of the *old* and *new* register mappings (2 signatures of B bits per entry).

5.5.4 Static Policy

Static policy is based on *statically* coupling each logical register to a fixed signature value. This is, every logic register `RAX`, `RBX`, etc, will always be mapped to the same signature. Note that a physical register is not always tied to the same signature for this policy, as opposed to the `MOD` policy. Like `MOD`, signature generation is generated independently of the rename operation, and therefore we can reduce the area overhead in the rename table and the ROB. However, we focus on improving cases (3) *Premature issue* and (6) *Register free-list misuse*, where `MOD` is expected to underperform.

Given the limited number of signatures, many logical registers will share the same signatures: a good register distribution must be found so that signature usage is balanced. We run our set of SPEC benchmarks (described in Section 5.6) and count every use of the logical registers as an operand source. Figure 5.10 shows the distribution in terms of percentage over all accesses. One can see that the total number of logical registers accessed is very small and nicely distributed.

We build the following buckets for a 2-bit signatures scheme, although other buckets could have been chosen:

- **Signature** `00`**.** `RSP`, `RDI`, `MM3`, `TMP1`, for a total usage of 24.96%.
- **Signature** `01`**.** `RAX`, `RBP`, `FTMP0`, `MM7`, `MM5`, `OTHER`, for a total usage of 24.97%.
- **Signature** `10`**.** `TMP0`, `RSI`, `RBX`, `MM6`, `MM1`, for a total usage of 25.12%.
- **Signature** `11`**.** `RCX`, `RDX`, `MM0`, `MM2`, `MM4`, `FTMP1`, for a total usage of 24.95%.

When moving to a 3-bit signature scheme, the chosen signature distribution is as follows:

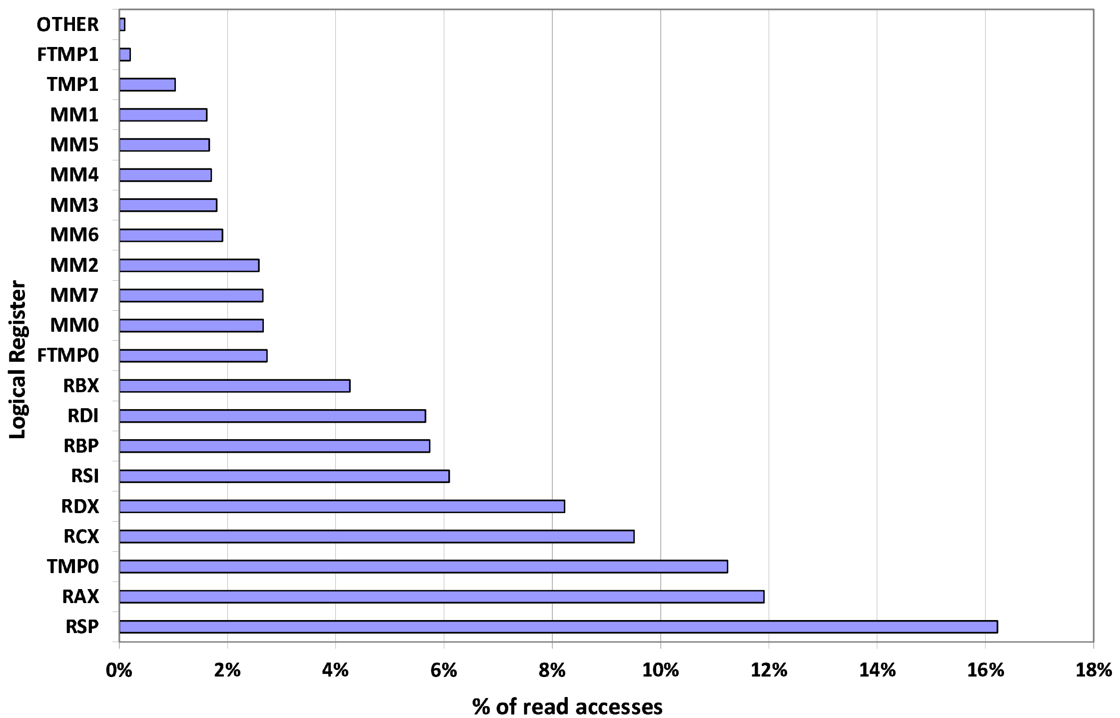


Fig. 5.10: Distribution of usage for the different logical registers across all benchmarks

- **Signature** 000. RSP, for a total usage of 16.22%.
- **Signature** 001. RAX, for a total usage of 11.91%.
- **Signature** 010. TMP0, FTMP1, for a total usage of 11.66%.
- **Signature** 011. RCX, MM3, OTHER, for a total usage of 11.66%.
- **Signature** 100. RDX, MM7, MM1, for a total usage of 12.50%.
- **Signature** 101. RSI, MM0, MM6, MM5, for a total usage of 12.32%.
- **Signature** 110. RBP, FTMP0, MM2, TMP1, for a total usage of 12.08%.
- **Signature** 111. RDI, RBX, MM4, for a total usage of 11.65%.

These distribution were empirically determined so that signature usage is balanced. However, it is interesting to note that as the number of signatures increases beyond 8 (3-bits), the distribution is ultimately determined by single logical registers having the biggest percentage of accesses. For example, logical register RSP will always cause that a signature has a minimum of a 16.22% of usage.

Hardware modifications

Our mechanism just requires a hard-wired table that indicates the signature for each particular logical register. Access to this table is done in parallel and independently of the regular operation. Since the signatures depend on the logical register, we do not need to keep the signatures in the rename table or the ROB, since we can obtain this information from the mapping table. On top of the modifications detailed in Section 5.3.3, we would require:

- **Rename stage.** A mapping table that holds the signature for every logical register. Based on our experiments, that would be a table with 32 entries, B bits each.

5.5.5 Enhanced Static Policy

The *Static* policy will tend to use a lot a given signature for codes that reuse a lot a particular logical register in a short interval of time. In this case, some signatures will be used more than the others (i.e. there is low signature variability), which may hurt coverage. Some of the failure scenarios described in Section 5.2 suffer from this phenomenon.

In case (1) *Selection of wrong inputs*, the probability of not detecting an error grows since the chance of a wrong entry's signature to be the same as the correct entry's one will be higher than in the average case.

For case (5) *Data stall in the bypass network*, a coverage lower than the average case is expected since the probability of two consecutive writes storing the same signature will grow.

In cases (2)-(4) *Wrong register file access, Premature issue, Wrong tag*, we do not expect a noticeable impact. The reason is that the number of "wrong choices" is big enough so that the probability of aliasing is the average one. In this case, the short-term variability is not so important as the long-term one, which is expected to be as good as the average case.

The following sections describe signature handling optimization that can be applied to the `Static` policy. These approaches solve the variability problem for cases (1) *Selection of wrong inputs* and (5) *Data stall in the bypass network*, as well as an additional enhancement to boost coverage for case (3) *Premature issue*. We refer to this policy as `Enhanced Static`.

Signature Masking

If a functional unit input multiplexor selects a wrong input it may happen that such error remains undetected due to signature aliasing. In order to alleviate the aliasing problem in *selection of wrong inputs*, the variability of signatures at its inputs must be maximized. By doing so we minimize the probability of picking a wrong input and not detecting the error.

In order to increase signature variability for the `static` policy, we propose to *dynamically transform* the signatures flowing through the bypass network (actually, the transformed residues) by XOR-ing them with a mask. Such mask will be statically defined for each combination of bypass level and execution port. Our proposal is depicted in Figure 5.11. Note that the figure does and the description is given for an implementation without residue integration, for clarity's sake.

The whole process is as follows:

- When a value is produced, we continuously transform its destination signature ($Sign(Dst)$) with different masks as it moves around the bypass network and its levels. The masks will vary depending on the execution port and the bypass level. In the example depicted in Figure 5.11 we assume only one execution port *P0*, and two bypass levels *BL0* and *BL1*. When the value generated in *P0* is currently available in the bypass level *BL0*, its destination signature is masked with mask *L0*. The next cycle, when the value and signature will be located at the next bypass level (*BL1*, the signature will be XOR-ed with mask *L1*.
- When an instruction issues, the expected signatures for its operands (the ones read out from the issue queue) are XOR-ed with the proper masks ($m1$ and $m2$). The information to select the proper mask is determined by *tag* comparison (that decides from where the operand is to be grabbed from). This information indexes a hardwired table *mask table* to obtain the proper mask.

Masks tables are built by trying to equally distribute the occurrence of masks reaching to each execution port. A possible mask table is depicted in Table 5.3. For instance, a signature grabbed from bypass level *BL1* in port *P1* will be XOR-ed with `01`. We apply the neutral mask `00` to the signatures coming from the register file.

Table 5.4 shows the different masks that are dynamically applied to signatures traveling in the bypass network so that the specifications shown in Table 5.3 are met.

Example. If a $Sign(Dst)$=`10` gets through the execution port *P0* and reaches the first level bypass *BL0*, it will be XOR-ed with `01`, resulting in a transformed

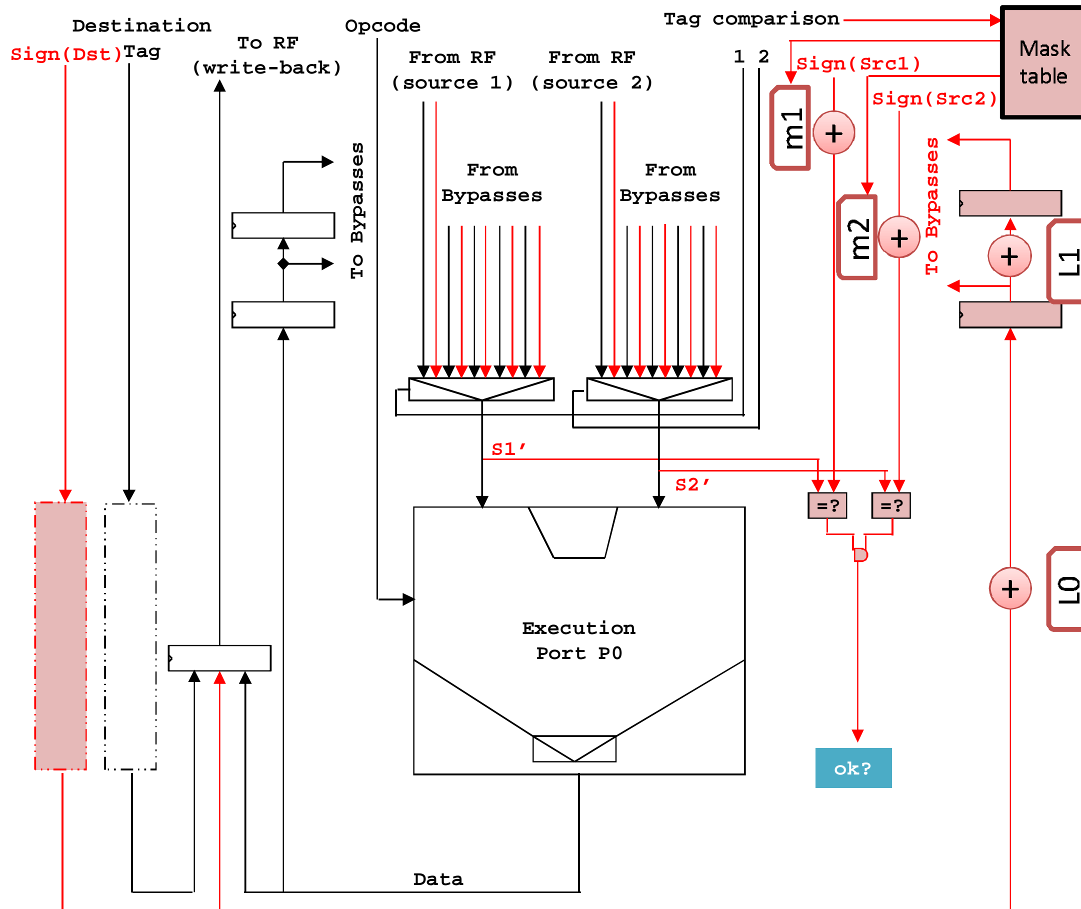


Fig. 5.11: Signature masking enhancement to boost coverage for *'Selection of wrong inputs'* case: extensions in the backend logic. *L0* is the mask for the bypass level *BL0*, whereas *L1* is the mask for the bypass level *BL1*

signature $Sign(Dst)$=11. When it later reaches bypass level *BL1*, it will be XOR-ed with 11, obtaining a transformed signature $Sign(Dst)$=00. An original source signature $Sign(Src)$ that is supposed to obtain the value from port *P0* and bypass level *BL1*, would use the mask 10 according to Table 5.3:

$$Sign(Dst)\ XOR\ 01\ XOR\ 11 = Sign(Src)\ XOR\ 10$$

Rotating Signature Masking

Assume the same signature being written by two different data values in consecutive cycles in a given bypass latch. If that latch fails to store the second write, such error (*Data stall in the bypass network*) will remain undetected. This will happen even if we apply masks per bypass/port (since the masks are hard-wired in each stage).

Table 5.3: Mask table for a processor with two execution ports (*P0* and *P1*), two bypass levels (*BL0* and *BL1*) and the write-back port (*WB*)

	BL0	***BL1***	***WB***
P0	01	10	00
P1	11	01	00

Table 5.4: Values of the masks set up at every bypass level and execution port

	P0	***P1***
BL0	01	11
BL1	11	10

To increase coverage against this failure scenario, we improve the masking technique by rotating every cycle the masks that are applied at each bypass path. This way, even though the signatures reaching the latches might be identical, they are transformed with different masks in different cycles prior to being latched.

This requires the mask table at the input multiplexors to be also accessed in a rotated fashion, which allows the issued instructions to use the proper mask.

Signature-Based Free Lists

An instruction suffering from a *premature issue* will consume an old version of a physical register (since it will not find the value in the bypass network). Although the `static` policy does not seem to suffer much this situation (unlike `RR` policy), we have proposed a mechanism to boost the probability of detecting such an error: we enforce that in two consecutive allocations physical registers are assigned a different signature.

To guarantee this property we arrange one free-list per signature, instead of having just one free list. The sum of all capacities equals the capacity of the original free list. We return to free-list i all registers which in their last allocation were paired to signature $Sign(i)$. At rename time, an instruction that receives $Sign(i)$ as destination signature will pick its physical register from any free-list *but* i. This guarantees that in its previous allocation, a physical register was signed with a signature $Sign(j), Sign(j) \neq Sign(i)$. In addition, in order not to penalize performance, if the only free-list containing registers is $Sign(i)$ we choose not stall the pipeline and proceed with one of its registers. When releasing a register, if the destination free

list is full any other is chosen for the same reason.

Note that in order to choose from a convenient free-list the destination signatures must have been assigned prior to the rename stage (i.e. in the decode stages). Therefore, choosing among free-lists is off the critical path.

Hardware modifications

In order to enhance the *static* policy, on top of the hardware requirements described in Section 5.5.4 it is needed:

- **Issue system.**
  - We need the *mask table* to perform *signature masking*. The *mask table* is accessed upon issue, because at that moment it is known where the operand will be grabbed from. Every entry in the *table* has B bits; and the total number of entries is the number of execution ports multiplied by the number of bypass levels. To access the table in a rotated manner, we need an auto-incremented modulo counter to offset the entry address.
  - After accessing the *mask table*, the source signatures obtained from the issue queue need to be masked. This is achieved by means of two XOR functions of B bits each, per execution port.
- **Bypass network.**
  - For every bypass latch, we need hardwired masks of B bits each. To implement the masking rotation, we need as many masks per bypass latch as the number of bypass levels.
  - For every bypass latch, we need a XOR function of B bits to mask the signatures.
- **Allocation logic.**
  - The register free list containing R *tags* is split into 2^B free lists containing $R/2^B$ *tags* each. For small values of B the overhead is negligible.

5.6 Evaluation

This section presents a detailed evaluation of the proposed register dataflow validation system. We evaluate it in terms of area, power and coverage for our baseline processor.

5.6.1 Coverage Results

From a global perspective, our previous studies [205] based on AVF analysis indicate that all the hardware involved in implementing register dataflow/computation functionalities represents 74.58% of the SDC FIT rate caused by soft errors (excluding protected structures, like caches, TLBs, etc.). Previous studies [212] report similar error rates using fault injection methodologies. Furthermore, 61.06% of the baseline processor area is exposed to other sources of failures, including wear-out, design and hard faults (again, excluding protected structures). By analyzing the microarchitectural blocks and by means of fault vulnerability studies (as described in Section 4.1.3), we have determined that the proposed technique is able to potentially cover 60.56% of the baseline processor area, and to potentially target 70.38% of the SDC SER FIT.

Given these area and potential soft error rate targets, actual error detection coverage is assessed by computing the capability of our framework to detect the faulty situations described in Section 5.2: (1) *Selection of wrong inputs*, (2) *Wrong register file access*, (3) *Premature issue*, (4) *Wrong tag*, (5) *Data stall in the bypass network*, (6) *Free-list misuse*, (7) *Load replay errors*, and (8) *Deadlock*. Residue coding fault coverage has been deeply studied in the literature and shown to be about 90% for 2-bit residue codes, so we just focus on its hardware costs. *Load replay errors* has 100% coverage since we enforce the usage of a wrong signature to trigger the error detection mechanism. For the *Deadlock* case, a watchdog timer is enough. Therefore, for the rest of the evaluation, we will concentrate on cases (1)–(6).

Coverage results have been obtained by means of error injection, as described in Chapter 4. For each SPEC benchmark, we perform 1000 effective fault injections for each class of error independently. Error injection is performed at the microarchitectural level (e.g. a tag is corrupted, an instruction issues too early, etc). We allow the fault to propagate and check wether the fault is detected or not. Each fault has been randomly injected during the ten first million executed instructions, after the warm-up period. Each experiment has been allowed to run for 100M instructions, as described in Chapter 4.

As we have discussed in Section 5.5, when using B bits to encode the signature, the average case probability of having aliasing is $\frac{1}{2^B}$. Therefore, the expected average-case coverage of our technique in this case would be $1 - \frac{1}{2^B}$. We now compare this theoretical numbers with results obtained experimentally.

We focus on 2-bit (see Figure 5.12) and 3-bit (see Figure 5.13) signatures, since a small number of signatures has a larger impact on the expected average-case coverage and the efficiency of the signature allocation policy. Our experiments with 4-bit signatures show that the variability is enough and we achieve the expected coverage

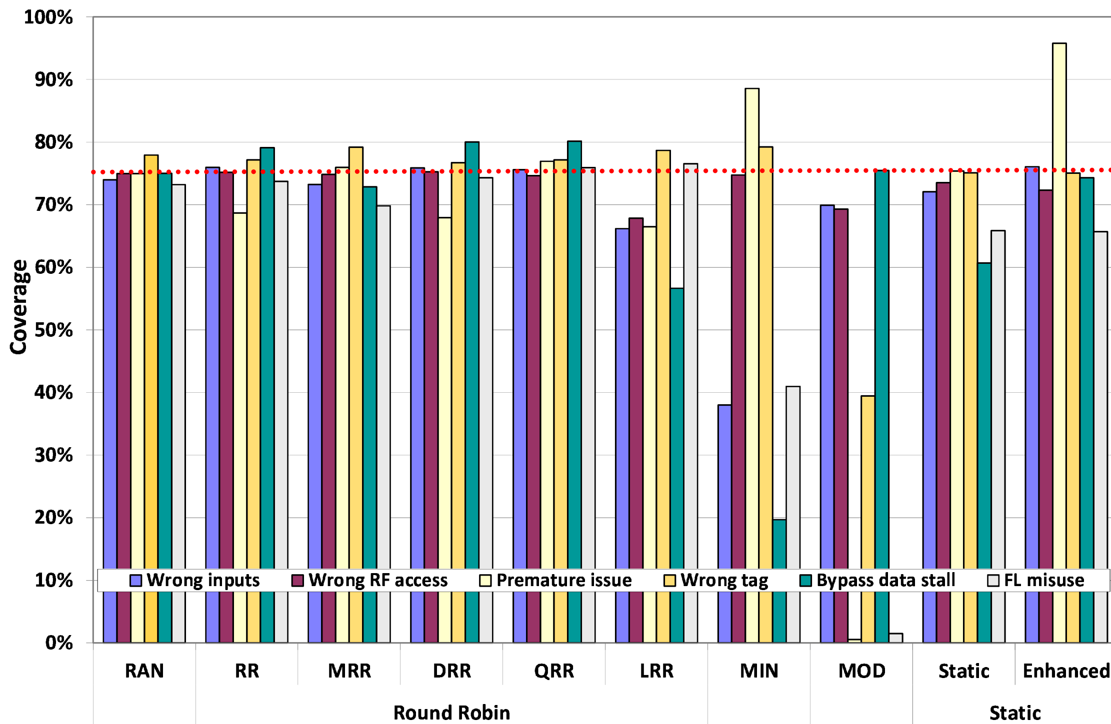


Fig. 5.12: Coverage results for all policies and error scenarios (1)-(6) for 2-bit signatures

in all situations. Note that for 2-bit and 3-bit signatures, expected average-case coverage is 75% and 87.5%, respectively (dotted lines in Figures).

A random assignment policy, `RAN`, is shown for comparison purposes. Although `RAN` is not a real implementation, it serves the purpose of showing the coverage of a signature generation mechanism that offers homogeneous signature usage. As one can see in both Figure 5.12 and Figure 5.13, the coverage achieved by `RAN` is almost the expected average case for all cases.

Round-Robin Policies

We can first notice that `RR`, `DRR` and `QRR` show similar trends, whereas `MRR` and `LRR` behave a bit worse (`MRR` specifically for 3-bit signatures).

We start analyzing the `RR` policy. We observe in Figure 5.12 that when we only have 4 (2-bit) signatures, the coverage is below the expected average for the (3) *Premature issue* case. However, it works fine when the number of signatures is 8 (Figure 5.13). This was expected and it is caused by the "wrap-up phenomenon" (see Section 5.5.1). Our results show that for 2-bit signatures `DRR` does not help in this case since we do not increase enough the signature variability. However, `QRR` increases coverage from 69% up to 76%. For 3-bit signatures, we have observed that this class

of round-robin policies behave very similarly (`RR` does not underperform for case (3) *Premature issue*), because in that case the variability is high enough to naturally avoid the wrap-up phenomenon. However, for 3-bit signatures `DRR` still misbehaves (2 counters are still not enough to counteract the "wrap-up phenomenon").

For 2-bit signatures, `MRR` achieves an average coverage close to the expected one. This is due to the fact that `MRR` tries to balance the amount of different signatures in the pipeline by starting the assignment with the least present signature. The first instruction in a rename bundle will be given the least present signature, in order to try to balance its distribution. The rest of the instructions in the rename bundle will be given consecutive signatures, potentially introducing additional unbalancing. However, with few signatures, less consecutive rename cycles are required to balance them again. As we increase the number of signatures, balancing cannot be rapidly achieved. This implies that for 3-bit signatures a subset of the 8 signatures will be generated in consecutive cycles. As a consequence, cases (1) *Selection of wrong inputs*, (5) *Data stall in the bypass network* and (6) *Register free-list misuse* are specially impacted, and show a coverage below the expected one (in dotted lines).

For both 2-bit and 3-bit signatures, `LRR` shows similar behavior. As discussed in Section 5.5.1, case (5) *Data stall in the bypass network* suffers the most. The reason behind is that diversity for a given cycle is not guaranteed across different logical registers because they use different counters. It could happen that a small loop may have all its instructions mapped to different round-robin counters having the same value. In this case, the signature assignment distribution would have no diversity during each rename cycle. Cases (1) *Selection of wrong inputs*, (2) *Wrong register file access* and (3) *Premature issue* also suffer because of that reason.

Finally, when comparing `RR`, `DRR` and `QRR` against `MRR` and `LRR`, we notice that for failure scenario (5) *Data stall in the bypass network*, the former ones show better coverage. The reason behind is that these policies maximize the distance between two consecutive uses of the same signature, and therefore, the probability of reading a stalled latch with the same expected signature is lower.

Minimum In-Flight Use Policy

The goal of this policy is maximizing the variability for each physical register, which is achieved, as shown by the good coverage numbers in cases (3) *Premature issue* and (4) *Wrong tag*. However, the same minimum occurring signature can be assigned to many consecutive instructions, which decreases the variability in the rest of the pipeline, and therefore, hurts the coverage for cases (1) *Selection of wrong inputs*, (5) *Data stall in the bypass network*, (6) *Register free-list misuse*.

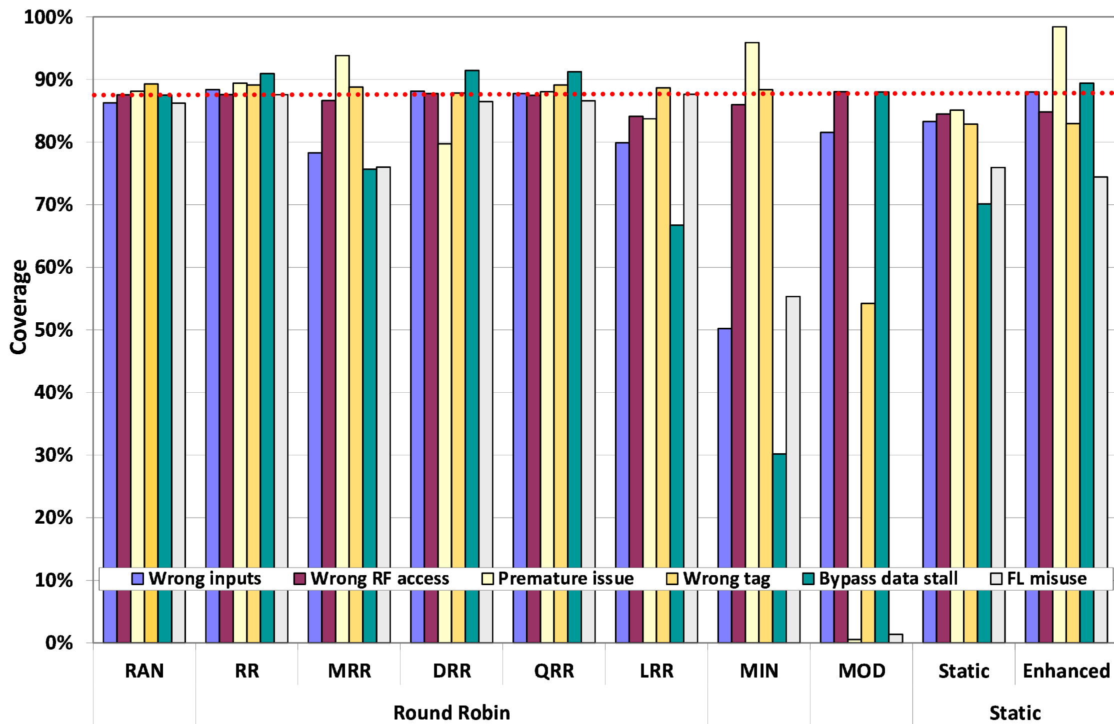


Fig. 5.13: Coverage results for all policies and error scenarios (1)-(6) for 3-bit signatures

Physical Register Policy

It is worth noting that, as discussed in Section 5.5.3, cases (3) *Premature issue* and (6) *Register free-list misuse* are unprotected because each physical register is always given the same signature.

The signature variability achieved for case (4) *Wrong tag* is lower than expected in either of the two configurations (2-bit and 3-bit signatures); the reason behind is that modulo 4 (i.e. using 4 signatures) allows detecting errors only in the two least significant bits. When using 8 signatures we work with modulo 8, which can detect errors in the three least significant bits of the word it is protecting. In order to solve this coverage problem we would need to use different modulo (e.g. 3 or 5), but this is much more costly to implement and will use less signatures. Overall, results indicate that `MOD` is not attractive from a coverage point of view.

Static Policies

Now, we assess the coverage for the `static` and `enhanced` policies. Although it has a lower hardware cost that the previously discussed policies, the coverage is below *round-robin* policies for many of the scenarios. Our results for the `static`

policy confirm the intuition that cases (1) *Selection of wrong inputs* and (5) *Data stall in the bypass network* obtain a lower than expected coverage. This is specially significant for case (5) *Data stall in the bypass network*, which is 15% worse than the expected coverage for both signature sizes. Low variability of signatures is suffered during program phases using few logical registers. In summary, the results show a coverage ranging between 60% and 75% for 2-bit signatures. A similar behavior can be observed for 3-bit signatures: coverage ranges from 70% to 85%.

When we apply the enhancements mechanism (`enhanced static` policy), coverage hits the expected value. Furthermore, the Signature-Based Free Lists enhancement for case (3) *Premature issue* failure boosts coverage above 95%. In summary, the coverage for the targeted special cases ranges between 72-96% and 84-98% for 2-bit and 3-bit signatures respectively. A good design point can be found for 3-bit signatures where the average coverage is around 90%.

5.6.2 Overheads

This section details the impact of our framework in terms of cycle time, power and area. We took the detailed design of our baseline processor, and modeled the extra hardware to implement our schemes on top of the area-power-delay model described in Section 4.1.4.

Delay

When implementing the signatures stand-alone, checking the signatures is done in parallel to execution and has a delay lower than the functional units. We also increase the width of some multiplexors and the bypasses. Although wider multiplexors and bypasses may make them a bit slower, our assessment shows that it is not enough to impact the cycle time.

When implementing the signatures on top of residue checking, we only add two blocks of XOR gates. Adding one level of XOR gates to decode the transformed residues into source signatures does not impact the critical path of the execution stages, because residue functional units take less than half a cycle to compute the expected residue values. Similarly, encoding the produced residue with the destination signature is done with the expected residue computed by the residue functional unit. The signature/residue checking does not need to be accounted into the execution delay, and can be performed cycles after the full computations, as long as no instruction is retired before it has been checked. This staggered error checking approach is possible if the lag is lower than the writeback-to-retire latency of the processor.

Finally, signature allocation is performed in parallel with current logic for all different policies, and therefore, has no impact in the cycle time.

Area and Power

We classify the different signature generation policies in four different groups depending on their costs (see Section 5.5).

For a stand-alone signature checking implementation (Section 5.3.3), all policies require modifying the allocation stages, the issue queue, the register files, the bypass network, the execution units (to do signature checking) and the replay logic (to initially corrupt the signatures upon a miss). The hardware to perform signature checking in the execution units is negligible as shown in the following results, and the overhead to extend the replay logic has been accounted on top of the Load-Store Queue block (`LSQ` row in the result tables).

For an implementation on top of end-to-end residue checking (Section 5.4.3), the data cache interface and the Load-Store Queue need to be modified as well. The data cache interface is modified to compute residues for the data retrieved by loads, whereas the Load-Store Queue is modified to hold residues for both addresses-data.

For all the policy groups, on top of the required hardware previously mentioned, we detail the differences in hardware to implement them:

- **Class *round-robin***. It includes all *round-robin* policies and *also* the `MIN` policy. The cost of their counters are roughly the same, and these policies require extending the rename tables and the ROB to track for each physical register its associated signature.
- **Class *MOD***. It includes the `MOD` policy. The most important characteristic in terms of overhead is that there is no additional cost in the rename stage/tables because signatures depend on the physical tag and there is no need to store them to infer them.
- **Class *static***. It includes the `static` policy. There is no cost in the ROB, and no cost in the rename tables. There is a small cost in the *Rename* block to implement a hardwired table that indicates the fixed signature assigned to the each logical registers.
- **Class *enhanced***. It includes the `enhanced` policy. It requires the same hardware modifications as the previous item, but we account for the extra cost of the mask tables, the rotating masks (accounted in the Issue Queue block) and the signature-based free lists (accounted in the Allocation block).

Table 5.5: Area and power overheads for the different signature generation policies when end-to-end residue is absent. In each cell, we show the results for the 4 classes of policies: `round-robin` / `MOD` / `static` / `enhanced` when they are different

Block	**Area**			**Power**		
	%	Δ%		%	Δ%	
	Original	Sign-2	Sign-3	Original	Sign-2	Sign-3
`Bypass`	5.28	3.13	4.69	4.97	3.13	4.69
`FUs`	17.98	0.0	0.00	13.73	0.00	0.00
`L2$`	18.15	0.0	0.00	2.72	0.00	0.00
`Rename`	2.33	25.71/0.00/4.11/4.11	38.57/0.00/6.17/6.17	5.27	25.71/0.00/4.11/4.11	38.57/0.00/6.17/6.17
`IQ`	3.89	1.81/1.81/1.81/2.26	2.72/2.72/2.72/3.40	6.73	0.77/0.77/0.77/1.22	1.15/1.15/1.15/1.83
`RF`	2.92	2.37	3.55	8.10	2.37	3.55
`D$`	15.02	0.00	0.00	12.73	0.00	0.00
`ROB`	2.54	4.71/4.71/0.00/0.00	7.06/7.06/0.00/0.00	9.85	4.71/4.71/0.00/0.00	7.06/7.06/0.00/0.00
`Alloc`	1.69	1.00	1.00	1.99	1.00	1.50
`LSQ`	7.11	0.50	1.50	3.02	0.50	1.50
`Frontend`	23.09	0.00	0.00	24.17	0.00	0.00
`CLK`	N/A	N/A	N/A	6.72	1.07/0.48/0.45/0.47	1.66/0.76/0.72/0.75
Total	100.00	**1.07/0.48/0.45/0.47**	**1.66/0.76/0.72/0.75**	100.00	**2.33/0.93/0.68/0.71**	**4.20/1.42/1.05/1.09**

Stand-Alone End-to-End Signature Implementation We first detail the area and power overheads for all classes of policies in Table 5.5, for a scenario where signatures are implemented stand-alone (no end-to-end residue infrastructure is available). We show in the *'Sign-2'* and *'Sign-3'* columns at the left side, the relative area increase due to the hardware additions required to implement them (with respect to the baseline core). Notice that when there are differences in terms of overhead for the different classes, we list them separated by slashes (*round-robin/MOD/static/enhanced*).

The results estimate that the overall area increase is small with respect to our baseline core. For 2-bit signatures, the largest area increase is 1.07% for 2-bit signatures, and 1.66% for 3-bit signatures, that corresponds to the *round-robin* class.

Static and *enhanced* policies just require a mere 0.45%-0.47% and 0.72%-0.75% area overhead for 2-bit and 3-bit signatures, respectively. Clearly, these policies provide a very high coverage for the different failure scenarios, while at the same time they have a negligible area overhead. *MOD* incurs 0.48% and 0.76% area increase for 2-bit and 3-bit signatures, respectively; its area costs are worse than for the *static* and *enhanced* policies and the same time the coverage was shown to be below the expected. We therefore discard the *MOD* policy in the rest of analysis.

We show the detailed power analysis on the right-hand side of Table 5.5. The overheads are stated with respect to the whole core for an end-to-end stand-alone signature implementation.

For the most expensive 2-bit signature protection scheme (*round-robin* class), we estimate a 2.33% power increase, and 4.20% for 3-bit signatures. *Static* and *enhanced* policies incur a minimal 0.68%-0.71% and 1.05% and 1.09% power cost, for 2-bit and 3-bit signatures. This low overhead (compared to *round-robin* policies) is possible because the ROB is not enlarged, and the rename stages are minimally modified.

Integration with End-to-End Residue Checking We now detail the overheads when signature checking is implemented on top of a processor with residue checking.

Column *'Residue 3'* in Table 5.6 shows the area overhead (with respect to the baseline core) that is needed to implement an end-to-end residue checking with a residue divisor of 3 (2-bit residues). We use previous works [58, 74, 76, 96, 102, 119, 125, 141, 169, 189] to estimate the area and power overhead for 2-bit residue for our baseline processor. It can be observed that an area of 2.43% is paid with respect to the baseline core, and the overhead mainly comes from the functional units, bypass network and register files. Note that the overhead in the register file is lower than when a 2-bit signature scheme is implemented stand-alone: the baseline register file is protected by parity, and residue coding can replace it.

Table 5.6: Area and power overheads for the different signature generation policies when end-to-end residue is implemented. In each cell, we show the results for the 4 classes of policies: `round-robin` / `MOD` / `static` / `enhanced` when they are different

Block	**Area**				**Power**			
	%	Δ% w.r.t Baseline	Δ% w.r.t Residue 3		%	Δ% w.r.t Baseline	Δ% w.r.t. Residue 3	
	Original	Residue 3	Sign-2	Sign-3	Original	Residue 3	Sign-2	Sign-3
`Bypass`	5.28	3.13	0.00	1.52	4.97	3.13	0.00	1.52
`FUs`	17.98	10.00	0.00	0.00	13.73	5.00	0.00	0.00
`L2$`	18.15	0.00	0.00	0.00	2.72	0.00	0.00	0.00
`Rename`	2.33	0.00	25.71/0.00/4.11/4.11	38.57/0.00/6.17/6.17	5.27	0.00	25.71/0.00/4.11/4.11	38.57/0.00/6.17/6.17
`IQ`	3.89	0.00	1.81/1.81/1.81/2.26	2.72/2.72/2.72/3.40	6.73	0.00	0.77/0.77/0.77/1.22	1.15/1.15/1.15/1.83
`RF`	2.92	1.18	0.00	1.17	8.10	1.18	0.00	1.17
`D$`	15.02	1.00	0.00	0.00	12.73	1.00	0.00	0.00
`ROB`	2.54	0.00	4.71/4.71/0.00/0.00	7.06/7.06/0.00/0.00	9.85	0.00	4.71/4.71/0.00/0.00	7.06/7.06/0.00/0.00
`Alloc`	1.69	0.00	1.00	1.00	1.99	0.00	1.00	1.50
`LSQ`	7.11	4.00	0.50	1.50	3.02	4.00	0.50	1.50
`Frontend`	23.09	0.00	0.00	0.00	24.17	0.00	0.00	0.00
`CLK`	N/A	N/A	N/A	N/A	6.72	2.43	3.27/2.68/2.65/2.67	3.86/2.96/2.93/2.95
Total	100.00	**2.43**	**0.82/0.24/0.21/0.23**	**1.39/0.52/0.48/0.51**	100.00	**1.35**	**2.10/0.73/0.48/0.51**	**3.28/1.21/0.84/0.89**

Table 5.7: Overheads summary of implementing end-to-end signature checking and end-to-end residue (residue is 3 for all configurations). Δ% with respect to a our baseline processor.

Size	Expected Coverage	*Round-Robin*		*MOD*		*Static*		*Enhanced*	
		Area	Power	Area	Power	Area	Power	Area	Power
2-bit	75.00%	3.27%	3.48%	2.68%	2.08%	2.65%	1.84%	2.67%	1.87%
3-bit	87.50%	3.86%	4.67%	2.96%	2.58%	2.93%	2.20%	2.95%	2.25%
4-bit	93.75%	4.44%	5.85%	3.25%	3.06%	3.20%	2.56%	3.24%	2.63%

Conversely, signatures are not meant to protect data, and must be accumulated with the parity bit. Regarding power costs, the overhead for the end-to-end residue checking implementation has been estimated around 1.35%.

Columns *'Sign-2'* and *'Sign-3'* in Table 5.6 show for each block, the area and power overhead with respect to a processor implementing end-to-end residue checking.

We can first observe that the area overheads are very small when end-to-end residue is the baseline infrastructure, even for *round-robin* policies. For 2-bit signatures and for the *round-robin* class, the largest area increase is 0.82%.

For 3-bit signatures area costs are 1.39% with respect to a baseline processor implementing end-to-end residue checking. Power increase is limiting as signature size increases: the overheads in the power-hungry structures like rename tables and ROB are big contributors. In these cases, the power costs are around 2.10% and 3.28% for 2-bit and 3-bit signatures, respectively.

By adopting the *Static* and *Enhanced* policies, these overheads can be minimized because those structures are not modified. Area costs are just 0.21%-0.23% and 0.48%-0.51% for 2-bit and 3-bit signatures, with respect to a baseline processor implementing end-to-end residue checking. From a power perspectives, *static* and *enhanced* policies incur 0.48%-0.51% extra power for 2-bit signatures, and 0.84%-0.89% for 3-bit signatures.

We finally show in Table 5.7 a *summary* of the costs of implementing a combined system performing end-to-end signature checking and end-to-end residue checking (overheads are computed with respect to the baseline processor described in Appendix A). Results were obtained for the different policy classes and for several signature sizes. The end-to-end residue checking system uses a residue base of 3 (2-bit residues) for all configurations. We include results for 4-bit signatures and we show the average-case coverage [5] just to illustrate the general overheads trends. Data shows that implementing a *static* or an *enhanced* signature checking scheme plus an end-to-end residue checking scheme incurs low costs. For 3-bit signatures, area increases at most by 2.95% and dynamic peak power by 2.25% with respect to

[5] $1 - \frac{1}{2^B}$, where B is signature size

Table 5.8: Comparative table of techniques that detect errors in the register dataflow

	Recovery	Detection Latency	Sources of Failure	μarchitecture Specific	SW Support	HW Cost (complexity)	Performance/ Power Costs
RNA [154]	No	Unbounded	Soft + hard errors, bugs	No	No	Low-Medium	Very low power, No performance
TAC [154]	Yes (pipe flush)	Bounded	Soft + hard errors, bugs	No	No	Low-Medium	Very low power, No performance
Scoreboard / Tag Reuse [33]	Yes (pipe flush)	Bounded	Soft errors	Yes	No	Very low	Very low power, No performance
DDFV [115]	No	Unbounded	Soft + hard errors, bugs	No	Yes + ISA extensions	High	Medium
Argus [114]	No	Unbounded	Soft + hard errors, bugs	No	Yes + ISA extensions	High	Medium
Our approach	Yes (pipe flush)	Bounded	Soft + hard errors, bugs	No	No	Very low	Very low power, No performance

our baseline core. We can also see for these two policies that increasing the signature size boosts coverage considerably at a small extra cost: area and peak dynamic power overheads grow almost linearly while at the same time the number of undetected faults is divided by half (coverage grows in a logarithmic trend). However, the overheads for the *round-robin* class are noticeable even for 2-bit signatures: the area requirements are roughly similar to the area requirements for a 4-bit *enhanced* configuration (but at a fraction of the achieved coverage). We therefore conclude that an *enhanced* policy is the best choice for the coverage-overhead design space.

5.7 Related Work

A few dynamic verification techniques have been proposed to detect errors in the control logic and hardware blocks implementing register dataflow tasks. Table 5.8 summarizes the features and pros and cons of each one of them.

Reddy *et al.* [154] propose two ad-hoc hardware assertion checkers. The first one, Register Name Authentication (RNA), aims at detecting errors in the destination *tags*. RNA assumes there is an additional rename table at the commit stage holding architectural mappings. When an instruction is renamed, the previous register *tag* is stored in the ROB. When the instruction retires, the register mapping in the redundant rename table will necessary contain the previous physical register in the ROB. RNA reads it and compares it with the one in the ROB. In order to detect faults in the free list and in the register allocation, RNA proposes managing two extra bits for every register *tag* in the free list: a ready and a free bit. When an instruction writes its result back, these bits are accessed and checked to be zero. RNA detects

faults affecting the *tags* in the rename table, faults in the architectural rename table, faults in the shadow rename tables, faults affecting the destination *tags* in the ROB, and faults in the free list and in the register allocator. However, RNA has several limitations and problems: (i) it is not able to detect errors in the source *tags*, (ii) the detection latency is unbounded, and an error can be architecturally committed before it is detected, and (iii) it requires adding a redundant architectural rename table with non-negligible area and latency overheads.

The second technique, TAC (Timestamp-Based Assertion Checking), detects error in the issue logic by checking that a chain of dependent instructions follow a valid chronological order. TAC assigns timestamps to instructions when they issue, and compares consumer timestamps with producer timestamps. TAC is hard to implement because every instruction must know its issue timestamp, the issue timestamp of its producers, and the latency of its producers. The size of a timestamp is big (13 bits) and does not scale with respect to the ROB and with respect main memory latency, incurring in non-negligible hardware costs. Furthermore, TAC does not catch the scenario where an instruction ends consuming wrong values from other datapaths.

Carretero *et al.* [33] propose two light-weight ad-hoc techniques to protect the issue logic. The detection of errors is achieved by: (i) redundantly checking at issue time operand availability by using idle register scoreboard read ports, and (ii) replicating the source tag in the CAM storage for those instructions that only require one renamable operand. Faults in the select logic, in the tag broadcast buses, in the CAM memories-matchlines, and in the ready bits can be detected with minimal modifications. However, faults affecting the register scoreboard go unnoticed. Most importantly, these techniques fail to define a comprehensive correct behavior for the register dataflow logic and they are tailored for a specific issue queue design.

Meixner's DDFV scheme (Dynamic DataFlow Verification) [115] detects faults and bugs in the fetch, decode, and register dataflow logic. DDFV is similar to control flow checkers that verify intra-block instruction sequencing by means of compiler support. DDFV dynamically verifies that the dataflow graph specified by an application is the same as the one computed and executed by the core. First, the compiler computes for every basic block a compact representation of its static (expected) dataflow graph, and embeds these signatures into the application binary. At runtime, the dataflow graph for every basic block is reconstructed and compared against the reference one.

A state history signature (SHS) is computed for each architectural register: it captures the instruction that generated the value and the history of the input operands, but not their values. Hence, a signature is recursively dependent on the chain of backward register-dependent instructions. Every register, data bus, value in the

Table 5.9: Blocks and logic protection for register dataflow validation techniques

	Fetch	Decode	Rename	Free List - ROB pdsts	Issue Queue	Ld/St Queue	ALU	RF + Bypasses	Data	Load Replay	CF Recovery
RNA [154]	No	No	Yes	Yes	No	No	No	No	No	No	No
TAC [154]	No	No	No	No	Yes	No	No	No	No	No	No
Scoreboard / Tag Reuse [33]	No	No	No	No	Yes	No	No	No	No	No	No
DDFV [115]	Yes†	Yes†	Yes†	Yes†	Yes†	No	No	Yes†	Yes	No	No
Argus [114]	Yes	Yes	N/A	N/A	Yes†	N/A	Yes§	Yes†	Yes	No	Yes
Our approach	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No

† : Protection within basic block, not across basic blocks
§ : ALU uses different error detection mechanisms than the one used for protecting values

ROB, etc. is extended to keep the SHS associated to that value. When the last instruction in a basic block commits, the SHSs are combined to form the execution-time DFS (dataflow graph signature). DGSs are 24 bits and SHSs are 10 bits each. Big area overheads are clearly required. The most critical issue is that checking is not supported for registers crossing basic blocks, as this information is unknown at compile time. In addition, there is a pressure during fetch, decode and commit because of the extra instructions and the added extra commit cycle. Errors are detected at the end of basic blocks, causing unbounded error detection latencies and errors being committed before being caught. Furthermore, SHSs must be saved by the OS to support exception and interrupt handling.

Argus [114] proposal by Meixner *et al.* extends DDFV to include computation and control flow checking capabilities. Argus is however meant for simple in-order cores. Unlike DDFV, Argus embeds into each basic block the DGSs of potentially two legal successors, rather than inserting its own DGS. During execution, Argus picks among the two DGSs the one belonging to target basic block. For computation checking, Argus uses residue checking or operand shifting. Even though Argus extends DDFV's coverage, it poses the same problems: ISA and OS modifications, compiler support, no failure containment and big area and performance overheads.

Table 5.9 summarizes for each of these register dataflow validation techniques the different features, control logic or blocks that are covered.

To begin with, DDFV and Argus are the only solutions that perform control flow checking ('Fetch' and 'Decode' are covered): they build upon existing control flow checker techniques that verify intra/inter-block instruction sequencing by means of compiler support (recall Section 3.4). However, DDFV only provides protection

within basic blocks, which ultimately ends up limiting the achievable coverage. Our technique does not check the control flow, but the baseline RAS features described in Section 4.2 can actually cover them in a simple manner.

The rename table, rename logic ('Rename' column), as well as the free list and register allocation-release functionalities ('Free List - ROB pdsts' column) are covered to a varying degree. RNA detects errors in the rename table and rename logic as long as they affect destination *tags*, not source operands. DDFV covers all these scenarios but at a basic block level. Conversely, our technique extends the protection to all *'Wrong tag'* and *'Register free list misuse'* cases by removing this basic block level restriction. Argus is meant for in-order cores, and thus these blocks are not covered.

None of the techniques cover the 'Load-Store Queue' logic. For DDFV or Argus, the compiler cannot help identifying producer-consumer memory instruction pairs. In Chapter 7 we introduce a unique solution to verify the Load-Store Queue logic in a targeted manner, so that coverage can be further extended.

ALUs are not covered by DDFV: a parity bit is just added to each produced register value. Argus does computation checking, but it relies on a set of techniques that are different than the mechanism used to protect values (parity). As a consequence, DDFV and Argus introduce extra delay before and after computation to check and produce the codes for the sources/results. Our technique protects computation and values using a unified mechanism, avoiding extra delays.

Regarding access to the RF and bypasses, neither TAC nor the Scoreboard Reuse techniques protect against scenarios like *'Wrong register file access'*, *'Selection of wrong inputs'* or *'Data stall in the bypass network'*. DDFV and Argus cover them as long as the consumed operands are produced within the same basic block. Our technique removes this severe constraint and covers against any possible failure scenario.

The 'Issue Queue' column captures faults manifesting as *'Premature Issue'* and *'Wrong Tag'* scenarios. TAC can just detect scenarios where instructions are issued prematurely but cannot detect errors in the operand *tags*. [33] catches faults in *tags* for single source instructions, and *'Premature Issue'* is covered as long as the scoreboard is not faulty. DDFV and Argus protect against *'Premature Issue'* and *'Wrong Tag'* scenarios, as long as the wrongly consumed value belong to the same basic block.

None of the existing techniques, but ours, are able to detect *'Load replay errors'*. Since DDFV or Argus signatures do not capture value information, a load hitting or missing in the cache will have the same signature.

Finally, in column 'CF Recovery' we list the techniques that validate that the state of the processor is correctly recovered upon a control flow recovery event (such

as a mispredicted branch, or an exception/interrupt). DDFV cannot validate this because all checking is done within basic blocks. Argus targets in-order cores, and control flow recovery just involves flushing the pipeline. In Chapter 6 we propose a solution for the control flow recovery logic, so that when combined with our baseline RAS features (described in Section 4.2), control flow can be protected.

5.8 Conclusions

In this chapter we have presented a novel continuous online validation solution to detect multiple sources of failures and bugs in the register dataflow logic, data values and computation. Our approach is and end-to-end solution that exploits microarchitectural register dataflow invariants.

We propose a novel technique that is based on signing every produced value flowing through the pipeline with a *signature*. Register consumers validate by means of signature matching that the value being consumed has been produced by the expected producer, independently if the producer belongs to the same basic block or not. Signatures flow together with values through the different register data paths and storage. Signature checking is performed upon execution and allows detecting errors with a bounded and small detection latency (allows recovery and fault isolation).

We have also described how our technique lends itself to a beneficial integration with end-to-end residue checking. First, both techniques have similar hardware requirements, and therefore the area cost can be amortized. And second, protection can be extended to values and computation.

Different implementations of the technique have be instantiated based on how signature generation is handled, and how big signatures are. This flexibility allows designers tuning the solution to better suit their area-power budgets as well as their FIT budgets. We propose nine different implementations for the signature allocation policies, and evaluate their coverage and overheads. Overall, our design is able to protect the *rename* tables, *wake-up* logic, *select* logic, *input multiplexors*, *operand read and writeback*, the *register free list, register release, register allocation*, and the *replay* logic. By combining it with an end-to-end residue coding scheme, we extend the protection to the *functional units*, *Load-Store Queue* data and addresses, *bypasses* and the *register file* storage.

Our studies show that the our approach is extremely light-weight in terms of power, area or slowdown (as opposed to global general techniques based on re-execution). Evaluations show that for a scheme with 2-bit signatures, total area overhead (depending on the chosen signature assignment policy) ranges from 0.21% to 0.82% with respect to a core implementing a 2-bit end-to-end residue checking

scheme. Similarly, power moves from 0.48% to 2.10% also with respect to a core implementing 2-bit end-to-end residue. For 3-bit signatures (and 2-bit end-to-end residue), area moves from around 0.48% to 1.39% and power varies between 0.84% and 3.28%.

The proposed technique is able to potentially cover 60.56% of the baseline processor area against faults, and to potentially target 70.38% of the SDC SER FIT (excluding protected structures). Given these area and potential error rate targets, actual error detection coverage depends on signature size and signature generation policies. On average, for most signature generation policies error detection coverage stays above the expected one (75% and 87.5% for 2-bit and 3-bit signatures, with respect to the mentioned potential area and error rate targets). However, it has been shown that specific failure scenarios are better handled by certain signature assignment policies. No impact on performance is introduced, and no ISA or OS changes are needed. Globally, an *enhanced static* signature assignment provides the best trade-off between fault coverage and area-power overheads: few hardware structures need to be modified or expanded while at the same time coverage is above the expected one.

CHAPTER 6

CONTROL FLOW RECOVERY VALIDATION

6.1 Introduction

Control flow recovery logic plays a critical role in current microprocessors, being involved in the execution of hardware performance improvement techniques like control-flow speculation, or functional issues like handling exceptions and interrupts.

Actually, a myriad of hybrid hardware-software techniques have been proposed since the early 80s to detect control flow errors in the fetch and decode stages, as described in Section 3.4. Recently, simple yet effective hardware solutions have also been presented to detect multiple sources of failures in the control flow, decode and allocation stages (described in our processor baseline RAS features). However, none of these solutions can validate the control flow *recovery* logic of modern processors.

In this chapter we propose, to our knowledge, the first solution to protect in a targeted way the whole control flow recovery control logic. Coupled with our processor baseline RAS features described in Section 4.2, the control flow logic can therefore be protected in an effective manner.

To achieve comprehensive failure detection in the control flow recovery logic, we exploit microarchitectural invariants that are validated at end-to-end paths. Similar to the control flow recovery logic, we split the validation process in two independent validation steps: (i) validating that the rename table (also known as RAT, register alias table) state recovery logic works correctly, and (ii) validating the squashing of instructions dependent on mispredicted control flow paths by providing a fault tolerant identification of these instructions.

This chapter presents two novel and light-weight continuous on-line testing techniques that cover the control flow recovery logic against multiple sources of failures (including soft, intermittent, hard errors and design bugs).

The rest of the chapter is organized as follows. Section 6.2 delves into the current implementation of control flow recovery in modern out-of-order processors. Section 6.3 describes the failures that this control logic may suffer. Section 6.4 and Section 6.5 present the end-to-end microarchitectural solutions for detecting errors in the control flow recovery logic. Afterwards, Section 6.6 evaluates our techniques in terms of fault coverage, area and power overhead. Finally, we summarize the main conclusions in Section 6.7.

6.2 Control Flow Recovery in Modern OoO Processors: Overview

Modern out-of-order processors rely on speculative execution to boost performance. By predicting the target of branch instructions before they have been resolved, control flow speculation allows the processor to exploit higher instruction level parallelism. Moreover, multiple mispredicted paths can simultaneously coexist with instructions belonging to the corresponding corrected paths. Current processors also demand precise exception/interrupt handling. A precise exception means that exceptions must be taken in program order, in such a way that only instructions prior to the offending instruction can be completed, whereas the following instructions are skipped. In all cases, the processor needs to handle the unexpected change in the control flow, restore the microarchitectural state and resume the execution correctly. Whereas these features improve the processor performance, this also clearly requires an important overhead in terms of area and complexity [221].

Modern processors devote a significant amount of hardware and complex control logic to provide an efficient implementation of control flow recovery. Next, we provide a high-level description on how a branch misprediction typically affects the microarchitectural state, and what microarchitectural components and logic is required to support control flow recovery. For the sake of simplicity, in the rest of the chapter we will focus on the branch misprediction mechanism. Notice that control flow recovery for exceptions and interrupts and other speculative performance solutions use similar mechanisms.

Restoring State. Modern processors hold the register speculative state in the rename table (RAT). The main approach to assist RAT recovery is based on mechanisms that take checkpoints of the register mappings and roll back to the proper checkpoint upon a control flow misprediction. A low-complexity (but inefficient) approach consists in progressively reconstructing the RAT state. This is typically achieved by

accumulating on a retirement (architectural) RAT the register map changes of all the previous in-flight instructions [71]. This old scheme restricts the frontend from renaming instructions belonging to the corrected path until the branch commits, and therefore degrades performance. Conversely, a checkpoint of the RAT can be immediately copied to the frontend RAT. Therefore, there is a clear trade-off between the number of checkpoints and the branch misprediction recovery efficiency.

Modern designs use hybrid solutions that combine both benefits. For example, the RAT recovery process may be proactive: different shadow RATs can continuously monitor and *walk* through the Reorder Buffer regions as branches resolve (in parallel to the instruction execution). During the ROB walk, the shadow RATs are updated with register mappings so that whenever a branch misprediction occurs, the corresponding shadow RAT already reflects the register mappings up to the branch instruction [22]. Reactive recovery schemes are also possible: when a branch is resolved as mispredicted, these solutions identify the closest valid RAT checkpoint and copy it into the frontend RAT. Then, the frontend RAT is reconstructed by traversing the ROB from the checkpoint position until the mispredicted branch entry and undoing all the register updates that should have not been reflected. This introduces few cycles to traverse the ROB region, but results in a simpler yet efficient design [5].

Squashing of Control-Flow Dependent Instructions. Whenever a branch resolves as mispredicted, a squashing mechanism takes care of identifying, and marking the instructions that belong to the wrong path created. The instructions that are control-flow dependent on a mispredicted branch are frequently referred to as *'bogus instructions'* for conciseness purposes. We will use this term hereafter.

The challenge is that multiple unresolved (and potentially mispredicted) branch instructions may be in-flight across different pipeline stages. Thus, any instruction may depend on multiple outstanding branch predictions. Selectively squashing instructions without introducing too much tracking complexity is a microarchitectural design problem, particularly when instructions may be executed out of order. As described in Appendix A.2, branch coloring/tagging mechanisms are commonly used to tag and to invalidate speculative instructions. These squashing mechanisms are in charge of marking fetched branch instructions with a unique tag (one-hot bit-vector) and all fetched instructions with a branch path tag (aggregation of all previous branch tags). Upon execution, if a branch turns out to be mispredicted it sends to all in-flight instructions a squash broadcast signal, using its branch tag. Instructions whose branch path tag include the mispredicted branch tag are therefore identified. Conversely, if a branch turns out to be correctly predicted, it sends to all in-flight instruction a clear broadcast signal. The branch tag is then recycled for future usage. Therefore, complex hardware is needed to: allocate-deallocate colors,

implement broadcast buses to flush/clear colors, and store color bit-vectors for all in-flight instructions.

Typically, once the squashing mechanism marks as *squashed* the instructions in the ROB (with a *'bogus'* bit), they are not removed from it until commit time (as the rest of instructions) because: (i) bogus instructions may have uncontrollable activity (such as pending cache misses) that may otherwise affect instructions allocated to that entry, and (ii) the ROB entries indicate the bogusly allocated physical registers that must be released back to the free list, and this would be time critical if done during branch recovery.

To further improve performance, modern processors also *nullify* the instructions in the wrong path that have not been yet executed and therefore reside in other backend structures (like in the issue queue). This means that non-executed bogus instructions are forced to release their backend entries and are not allowed to execute. This is typically performed to achieve a fast drain of these instructions, releasing resources from the backend in such a way that instructions from the correct path can be allocated and executed as fast as possible. Notice that even if nullification is not implemented, if a branch resolves as mispredicted but belongs to a wrong path, no fetch redirection must be performed. Otherwise, we could be violating the program instruction sequencing.

Finally, it is necessary to provide the frontend engine the correct target address and stall the rename logic until all the recovery steps have been performed.

6.3 Control Flow Recovery Failures

Unlike RAM-like structures that can be protected by coding techniques like parity/ECC/residue, the dynamic control logic and the elements implemented around RAM cells (e.g. decoders, word-lines, etc) can hardly be protected by coding techniques. Faults and bugs in the control flow recovery logic could result into different microarchitectural errors. We start discussing the faults related to the *state recovery*, shown in Figure 6.1:

(A) *Wrong RAT flash-copy*: an error may arise when copying the frontend RAT into a checkpoint RAT. For example, we could be copying the RAT into a wrong checkpoint, either free or already in use. Similarly, a wrong checkpoint RAT could be restored when performing the recovery.

(B) *ROB mapping information error*: the ROB may suffer from faults in the register mapping information, such as the destination logical register, previous or

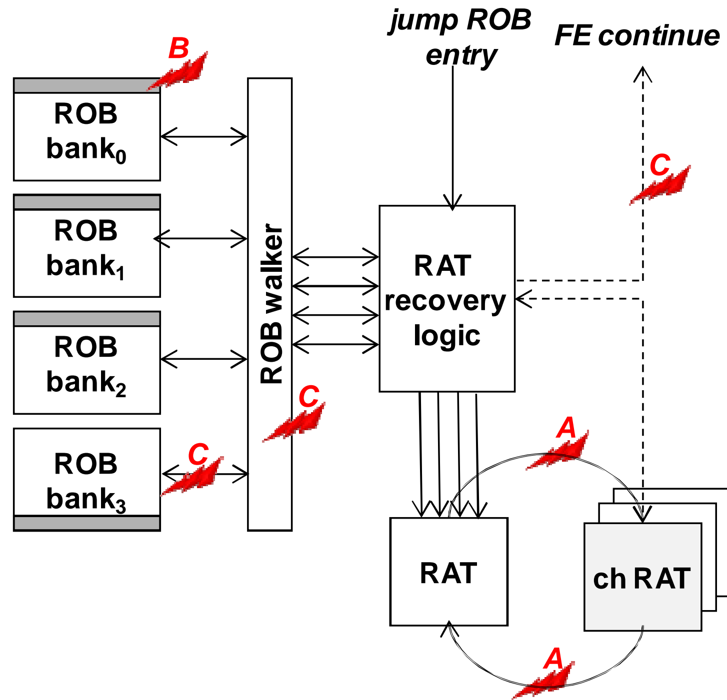


Fig. 6.1: Failure scenarios related to RAT state history reconstruction

current mapping. As a consequence, if the faulty ROB entry is used to perform the frontend RAT recovery, an instruction from the corrected path would consume or release a wrong register.

(C) *Wrong ROB walk*: recovering the RAT state requires traversing the ROB from the checkpoint entry up to the instruction whose control flow was mispeculated. The "ROB walking" control logic can also experience errors: some entries may not be accessed, the ROB bank pointer generation may pose design bugs, etc. Similarly, the frontend may start renaming the instructions from the correct path before the whole RAT state recovery has finished.

Next, we discuss the faults that may impact our ability to *squash control-flow dependent instructions*. They are shown in Figure 6.2:

(D) *Wrong identification of bogus instructions*: instructions have associated fields indicating their bogusness (such as the *bogus?* bit in Figure 6.2). In addition and as commented in the previous section, the processor must implement mechanisms to identify and mark (i.e. squash) wrong-path instructions as bogus. The branch tags, path tags, tag broadcast buses, management logic, etc. can suffer from faults. The net result is that the commit logic may perceive

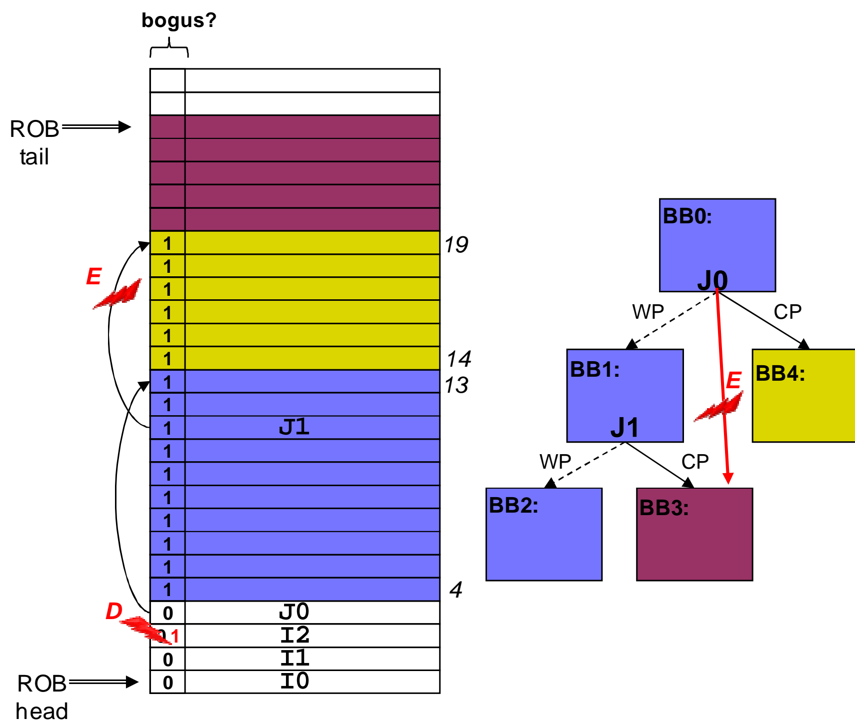


Fig. 6.2: Failure scenarios related to identification of control-flow dependent instructions

an instruction as bogus when the architectural state should be updated. The opposite scenario, where a correct instruction is perceived as bogus, may also happen.

(E) *Wrong fetch path redirection*: An error may cause that some non-executed instructions are not nullified once a branch resolves as mispredicted (in case of exceptions and interrupts, it could be a control-flow break). Whereas for non-branches this would simply cause a performance degradation, branches not being nullified can induce functional correctness problems. For example, in Figure 6.2 branch `J0` executes and resolves as mispredicted. Branch `J0` must nullify all the instructions residing at positions 4 to 13 in the ROB (that belong to basic blocks `BB0`, `BB1` and `BB2`). This includes nullifying branch `J1`, that has not been executed yet. However, it could happen that because of a fault or bug, branch `J1` is not nullified and later executes and resolves as mispredicted. `J1` would squash instructions belonging to the correct path: instructions at positions 14 to 19 in the ROB (that belong to basic block `BB4`). Moreover, branch `J1` would then redirect the fetch path to basic block `BB3`, which is not a successor of branch `J0`.

We propose two light-weight microarchitectural end-to-end error detection solutions. The first one will target failure scenarios A, B and C (Section 6.4), whereas

the second one will deal with D and E (Section 6.5).

6.4 End-To-End Validation of RAT State Recovery

In order to detect errors in the RAT state recovery control logic, we use an end-to-end RAT state *signature* generation-validation mechanism. The *end-to-end path* begins at the rename stage by computing for each instruction a small token (the *RAT state signature*) that summarizes the set of register mappings carried by all older instructions. The *RAT state signature* is based on the encoding of the logical and physical registers pairs. The *end-to-end path* ends once the recovery for a mispredicted branch is completed; after recovery, the signature of the frontend RAT state should match the signature that the mispredicted branch obtained at rename time.

Since we redundantly reconstruct the signature of the frontend RAT as we perform the misprediction recovery, we can validate the recovery logic by comparing the signature generated at rename time with the signature of the recovered RAT.

Hence, a simple end-to-end generation-validation mechanism is enough to detect errors in the RAT state recovery logic. Note that this concept also allows to dynamically check the RAT recovery upon another unexpected event (such as an exception, interrupt, etc.).

Section 6.4.1 describes how we generate the *RAT state signature* at rename time for each instruction. We will also show how the *RAT state signature* flows through the pipeline. Later, Section 6.4.2 will explain how we perform the validation upon a branch misprediction.

6.4.1 RAT State Signature Tracking

The *end-to-end path* begins at the rename stage. At this point, each instruction is given a codeword that summarizes the set of register mappings, including the own instruction mapping. These codewords flow with the instructions through the pipeline until they reach the issue queue (instruction scheduler), where they are allocated. Similarly, each of these codewords are read out from the issue queue once their instructions are issued.

For the sake of clarity, we start assuming that each of these codewords precisely encode the RAT state upon its instruction renaming. Conceptually, a RAT state R_i for an instruction I_i is a set $R_i \in R \subseteq l \times p$, where l and p are the set of logical and physical registers, respectively. Each mapping is a pair of logical and physical register $(l_i, p_{l_i}) \mid l_i \in l, p_{l_i} \in p$. Note that it does not mean that logical register l_i is

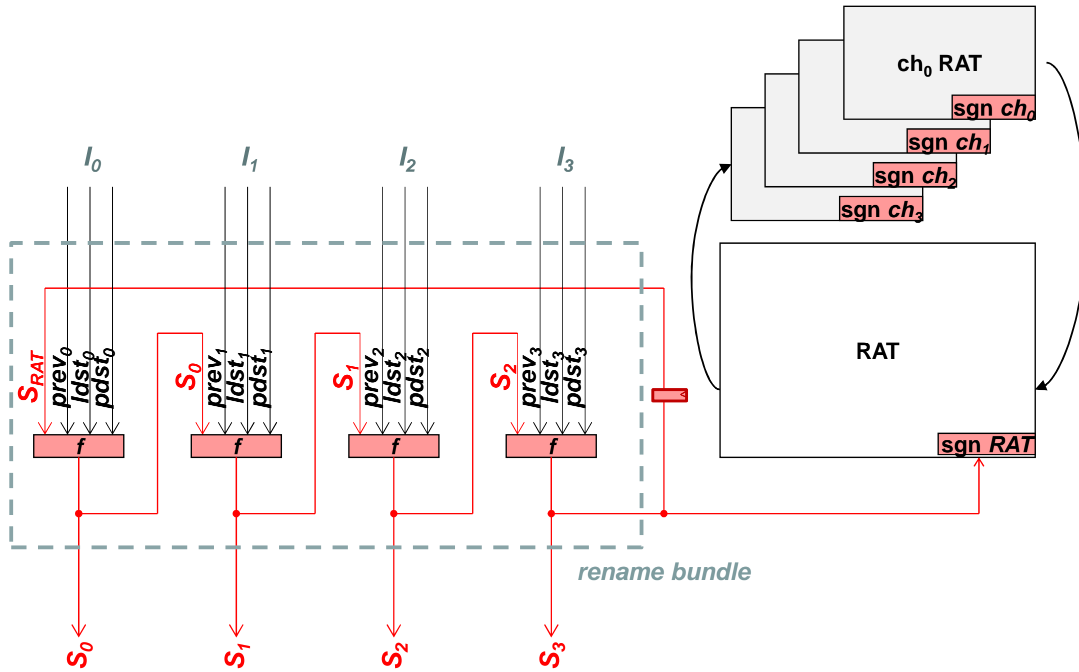


Fig. 6.3: RAT state signature generation: extensions in the rename logic

always bound to physical register p_{l_i}.

One may think of generating a RAT state R_i by inspecting the mapping for every renamable logical register after renaming instruction I_i. However, this generation scheme is impractical. Since the rename stage is performed in parallel for a rename bundle and the RAT is updated atomically at the end of the cycle, any instruction not being the last of its *rename bundle* would never observe a RAT state reflecting all the register renames up to these instructions. Moreover, such implementation would require (i) large area overhead for storing the RAT state, and (ii) a complex hardware, since many RAT read ports would be necessary to access the RAT entries in order to build the set of pairs for R_i.

In order to solve the issues mentioned in the previous paragraph, we introduce the concept of *RAT state signature*. The objective of a *RAT state signature* S_i is to encode a RAT state with a fraction of its codeword size. However, since some information is lost, a *RAT state signature* may correspond to multiple RAT states. We will discuss its impact in the coverage study in Section 6.6.

Figure 6.3 shows the process involved in generating the *RAT state signature* $S_i \in S$ for every instruction I_i. As described in Appendix A.2, each renamable instruction in the rename bundle obtains a *new* physical register tag (*pdst*) and the *old* physical register tag (*prev*) mapped to its logical register destination (*ldst*). We exploit the following property: the RAT state for a given instruction I_i is the RAT state for

the previous instruction I_{i-1}, just replacing the contents of the logical register being renamed (this is, *prev*) with the new destination mapping *pdst* (replacing the pair $(I_i.ldst, I_i.prev)$ with $(I_i.ldst, I_i.pdst)$). This approach allows computing a RAT state in a *forward* and accumulative way, avoiding traversing all the RAT mappings to build it.

The remaining step consists in applying this generation approach to *RAT state signatures*, in order to avoid big codeword overheads. The next subsection shows how to achieve this and what properties implementations must satisfy.

RAT Signature Generation: Formal Properties

As commented, the *RAT state* for a renamed instruction I_i is the RAT state for the immediately previous renamed instruction (instruction I_{i-1}) but just replacing the pair $(I_i.ldst, I_i.prev)$ with $(I_i.ldst, I_i.pdst)$.

We can formally define a function *RAT* that computes the RAT state for any instruction $I_i \in I$. Equation 6.1 shows that an instruction RAT state depends on the instruction mapping information but also depends on the RAT states of all previous instructions (including the boot-time RAT state $\perp_{<R>}$). The *add* and *remove* function are defined in Equation 6.2 and Equation 6.3.

$$RAT(I_i) = \begin{cases} \perp_{<R>} & : \; i = 0 \\ add(remove(RAT(I_{i-1}), I_i.ldest, I_i.prev), I_i.ldest, I_i.pdst) & : \; i > 0 \end{cases} \tag{6.1}$$

$$add(R_i, l_i, p_i) = R_i \cup \{(l_i, p_i)\} \tag{6.2}$$

$$remove(R_i, l_i, p_i) = R_i \setminus \{(l_i, p_i)\} \tag{6.3}$$

We can compute the RAT signature of a RAT state by means of a *SGN* function, as defined in Equation 6.4. To achieve so, we must define a function $h : l, p \rightarrow S$ and also a function $\oplus : S, S \rightarrow S$ that must be associative and commutative. We refer to h as a hashing function and to $\oplus$ as a combining function.

$$SGN(R_i) = \begin{cases} h(l_0, p_0) & : \; R_i = \{(l_0, p_0)\} \\ SGN(R_{i-1}) \oplus h(l_L, p_L) & : \; R_i = R_{i-1} \cup \{(l_L, p_L)\} \end{cases} \tag{6.4}$$

From Equation 6.2 and Equation 6.4 we can deduce Equation 6.5. However, up to this point we can not deduce the equivalence for the *remove* function. As a consequence, we define it as described in Equation 6.6, by introducing a function $\ominus : p, l \rightarrow S$.

$$(l_k, p_k) \notin R_i \overset{6.2,6.4}{\Rightarrow} SGN(add(R_i, l_k, p_k)) = SGN(R_i) \oplus h(l_k, p_k) \tag{6.5}$$

$$(l_k, p_k) \in R_i \Rightarrow SGN(remove(R_i, l_k, p_k)) = SGN(R_i) \ominus h(l_k, p_k) \tag{6.6}$$

A *forward* and accumulative signature generation mechanism could be implemented if we could prove that the result of applying the SGN function (Equation 6.4) to every RAT state generated by a *forward* RAT state generation mechanism is the same result as if we applied Equation 6.5 and Equation 6.6 in a forward manner.

If $R_i = RAT(I_i) = \{(l_1, p_1), \ldots, (l_k, p_k), \ldots, (l_L, p_L)\}$, and $l_k = I_{i+1}.ldst, p_k = I_{i+1}.prev, p_m = I_{i+1}.pdst$, then we deduce I_{i+1} signature from R_{i+1} as follows:

$$SGN(R_{i+1}) = SGN(add(remove(R_i, l_k, p_k), l_k, p_m) =$$

$$SGN(add(remove(\{(l_1, p_1), \ldots, (l_k, p_k), \ldots, (l_L, p_L)\}, l_k, p_k), l_k, p_m)) \xrightarrow{6.3}$$

$$SGN(add(\{(l_1, p_1), \ldots, (l_L, p_L)\}, l_k, p_m)) \xrightarrow{6.2} SGN(\{(l_1, p_1), \ldots, (l_k, p_m), \ldots, (l_L, p_L)\}) \xrightarrow{6.4^*}$$

$$h(l_1, p_1) \oplus \ldots \oplus h(l_k, p_m) \oplus \ldots \oplus h(l_L, p_L)$$

On the other hand, if I_{i+1} signature is computed in a *forward* manner then we deduce its equivalency as:

$$SGN(add(remove(R_i, l_k, p_k), l_k, p_m)) \xrightarrow{6.5} SGN(remove(R_i, l_k, p_k)) \oplus h(l_k, p_m) \xrightarrow{6.6}$$

$$SGN(R_i) \ominus h(l_k, p_k) \oplus h(l_k, p_m) =$$

$$SGN(\{(l_1, p_1), \ldots, (l_k, p_k), \ldots, (l_L, p_L)\}) \ominus h(l_k, p_k) \oplus h(l_k, p_m) \xrightarrow{6.4^*}$$

$$h(l_1, p_1) \oplus \ldots \oplus h(l_k, p_k) \oplus \ldots \oplus h(l_L, p_L) \ominus h(l_k, p_k) \oplus h(l_k, p_m)$$

As a consequence, the equivalency

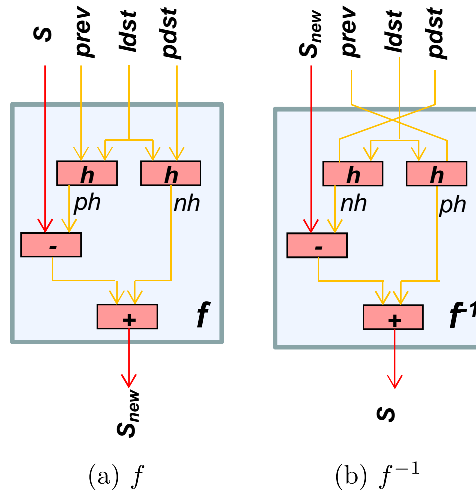
$$h(l_1, p_1) \oplus \ldots \boxed{\oplus h(l_k, p_k)} \oplus \ldots \oplus h(l_L, p_L) \boxed{\ominus h(l_k, p_k)} \overbrace{\oplus h(l_k, p_m)}$$

$$=$$

$$h(l_1, p_1) \oplus \ldots \overbrace{\oplus h(l_k, p_m)} \oplus \ldots \oplus h(l_L, p_L)$$

holds true if if we are able to find a commutative and associative function $\oplus : S, S \rightarrow S$ that has as an inverse function $\ominus$. Note that no restrictions apply for h.

From an implementation perspective, we define both $\oplus$ and $\ominus$ as bit-wise XOR operation. Other (more complex) design choices could be possible by defining $S \equiv \mathbb{Z}_n$, $\oplus$ as modulo addition ('+') and $\ominus$ as modulo substraction ('$-$'). Similarly, we define 'h' as a folding function implemented by means of XOR gates. Other generic hash functions are also possible, as described in [167].



(a) f (b) f^{-1}

Fig. 6.4: f and f^{-1} blocks implementation

Implementation of RAT Signature Generation

As soon as each instruction I_i has obtained its new *pdst* and the old physical destination mapping *prev*, the *RAT state signature* generation can start. This can be accomplished with very simple hardware shown in Figure 6.4(a); the centerpiece hardware is f, which is in charge of generating for each instruction its *RAT state signature*.

The logical destination and previous physical mapping are combined together using function h to form an individual register mapping hash signature (ph). The same happens with the new physical mapping (nh). Afterwards, function $-$ will remove the old register mapping hash from the previous instruction signature (S) and function $+$ will add the new register mapping hash. As a result, f produces S_{new}. It can be observed that 'f' implements $SGN(RAT(I_{i-1})) \ominus h(I_i.ldst, I_i.prev) \oplus h(I_i.ldst, I_i.pdst)$.

Each 'f' block forwards its output to the next 'f' block input. Note that the first instruction in the rename bundle will obtain the previous instruction signature from an instruction not in the rename bundle (because it was renamed cycles ago). This implies that at the end of the rename process, the signature for the last instruction in the rename bundle must be stored. This is accomplished by extending the frontend RAT to store the last renamed instruction's signature. Hence, the first instruction in the rename bundle will obtain the previous instruction's signature directly from the frontend RAT. Moreover, in order to implement the *forward* signature generation scheme, the frontend RAT signature must be reset to a static signature value at boot time. This signature is known a priori and consequently can easily be hardcoded. This

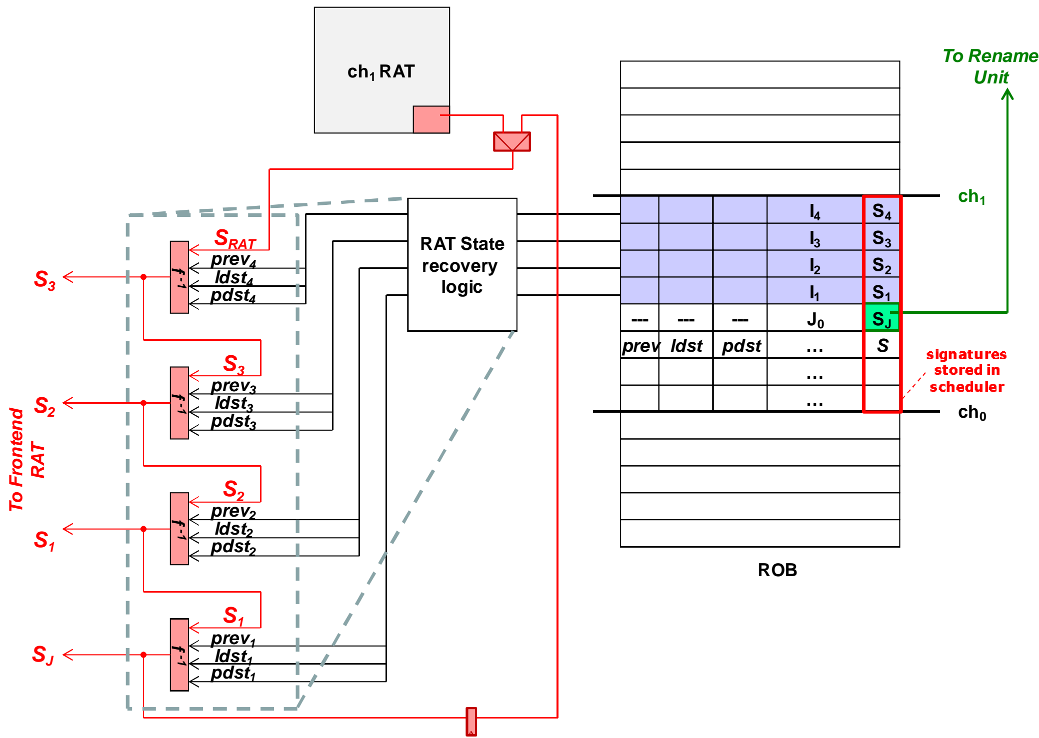


Fig. 6.5: RAT state signature reconstruction: extensions in the commit logic

value corresponds to $SGN(\perp_{<R>})$. We also change the checkpoint RATs to store the *RAT state signature*. This is necessary because checkpoint *RAT state signatures* are used during the validation step, as it will be explained next.

Once the rename process has been completed, every instruction has a *RAT state signature* that is written into the issue queue upon allocation.

6.4.2 RAT State Signature Validation

The *end-to-end path* ends once a control flow recovery action has been completed. At that point we can check whether during the RAT recovery process the proper sequence of steps were performed. This is simply accomplished by comparing the *RAT state signature* of the mispredicted branch (read from the issue queue upon instruction issue), with the *RAT state signature* obtained after recovering the RAT state.

The generation of the recovered frontend RAT signature is performed in parallel to the ROB walk recovery process (described in Appendix A.2), by mimicking the steps it performs (which can be potentially faulty). Figure 6.5 shows how this step

is performed. As the ROB keeps the mapping information and the checkpoint RATs have been extended to hold their own signatures, we can regenerate the signature of the frontend RAT by piggybacking on the ROB walk logic while it recovers the RAT state.

We start with the signature of the checkpoint RAT that the recovery process chooses for recovering. Then, we transform it so that all the changes introduced by the instructions covered by the checkpoint up to the mispredicted branch are removed from the checkpoint signature. For the example in Figure 6.5, this corresponds to the register mappings of instructions I_4 to I_1. In other words, for every instruction we walk, we remove the new register mapping hash (nh) from the *RAT state signature* and add the one from the old mapping (ph). We implement such function through a *backward* signature generation hardware f^{-1}.

As a starting point, the new register mapping hash of instruction I_4 would be removed from checkpoint ch_1 signature, and instruction I_4 old register mapping hash would be added to the resulting RAT signature. This would produce signature S_3 in Figure 6.5. Figure 6.4(b) shows the implementation details for f^{-1}. It can be observed that f^{-1} implements $SGN(RAT(I_{i+1})) \ominus h(I_i.ldst, I_i.pdst) \oplus h(I_i.ldst, I_i.prev)$.

In parallel, the *RAT state signature* of the mispeculated branch is sent to the rename logic from the issue queue for checking purposes. Once the recovery has been performed and before the rename of the instructions from the corrected path has started, we determine whether the signature generated at rename time (the one obtained from the issue queue) matches with the signature generated at recovery time. In case of a mismatch, a failure is detected.

6.4.3 Microarchitectural Changes

The mechanism requires the following hardware modifications (assuming N bits per signature).

- **Frontend RAT and checkpoint RATs.** Every RAT contains its own signature of N bits. Only 1 read/write port is needed for this extra field.
- **Rename logic.** As many f blocks as number of instructions in the rename bundle are needed. Every f block internally works with 3 signatures of N bits.
- **Issue Queue.** The CAM memory and wake-up/select logic is left unchanged. We enlarge the RAM memory, so that each entry in the payload RAM holds an extra field for keeping the RAT signature for that instruction (N bits per entry). The allocate and issue logic is widened in order to write and read out the instruction signatures.

- **RAT State Recovery Logic.** The ROB walk logic is extended with as many f^{-1} blocks as the maximum number of ROB entries that can read out during a RAT recovery cycle. Every f^{-1} block works with 3 inputs of N bits.

6.5 End-To-End Validation of Instruction Squashing

In order to detect errors in the mechanisms that implement the identification and squashing of control-flow dependent instructions, we use an end-to-end mechanism that tracks the range of instructions within the ROB that should be considered as bogus. The *end-to-end path* starts when a branch executes and resolves as mispredicted, because this is the earliest moment when a sequence of wrongly fetched instructions can be identified and tracked. At this point, the technique will update a small structure called *bogus check table* (`BCT`). Each entry of the `BCT` will store the range of instructions under the shadow of a given mispredicted branch. The *end-to-end path* ends when an instruction retires, because this is the very last moment when it is possible to check if the processor instruction squashing mechanism has been faulty or not. Furthermore, since our baseline processor (see Appendix A.2) can recover from control-flow mispeculations before the mispredicted branch reaches the head of the ROB (hence, supporting multiple in-flight corrected fetch paths), it is necessary to check the validity of branches upon execution as well. Otherwise, in case a mispredicted jump inside a mispeculated path was not squashed, it could corrupt our tracking mechanism. In case the processor recovers from control-flow mispredictions at retire time, this second check would not be necessary. However, due to the performance loss this latter option is rare.

Therefore, retiring instructions will access the `BCT` to check whether they belong to a wrong path interval (case (D) in Section 6.3) and mispredicted branches will access it at execution time in order to verify that they redirect the fetch path only when they are not under the shadow of an older mispredicted branch (case (E)).

Next, we will show how we generate the entries in the `BCT` and later, how we validate that the execution is resumed correctly.

6.5.1 Bogus Region Tracking

The end-to-end path starts when a branch is executed and resolved as mispredicted. Our idea is validating that we only commit the right instructions by tracking the ones that are control-dependent on mispredicted branches. We show in Figure 6.6 the information that we keep in each `BCT` entry. Since instructions are allocated in the ROB in sequential order, we can summarize the range of bogus instructions in a

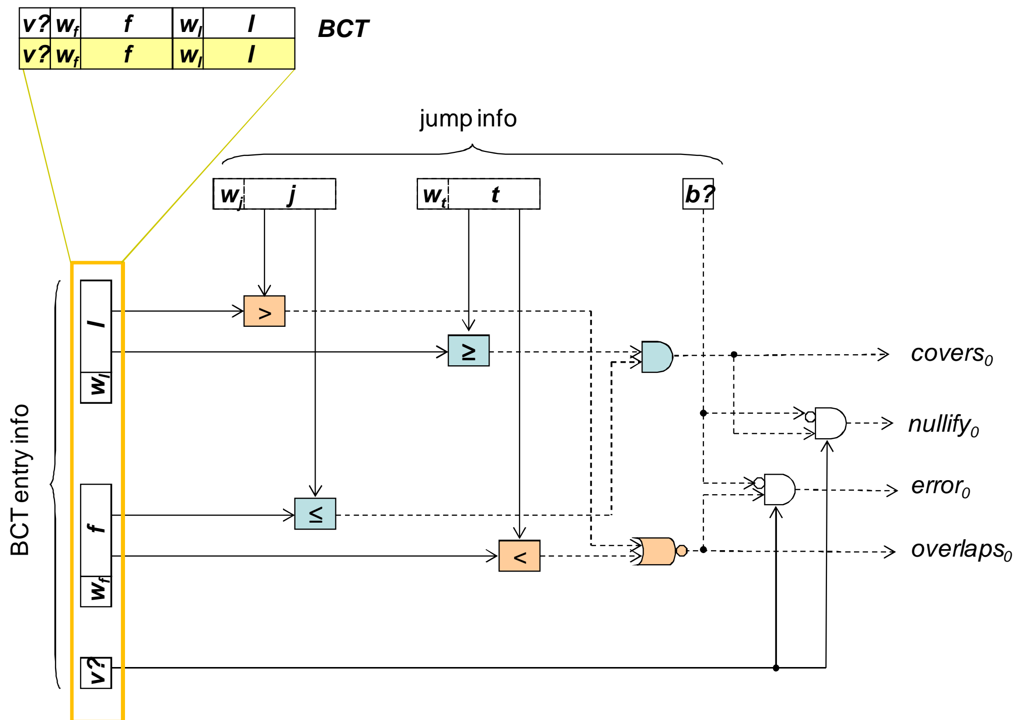


Fig. 6.6: BCT mechanism: extensions for bogus region tracking and validation

compact manner: we only store in the BCT the oldest and youngest instruction that are under the shadow of a mispredicted branch.

Once a branch executes and resolves as mispredicted it will fill an unused BCT entry by storing its ROB entry (f field in the BCT entry), the ROB tail value (l field), jointly with their corresponding wrap bits (w_f and w_l fields) to allow total age order determination among in-flight instructions [36, 142]. Note that this information is easy to obtain since each instruction in the issue queue keeps its ROB entry position and that the ROB tail value can be obtained upon a branch misprediction. Once the BCT entry is filled, it is marked as valid (v? field).

The BCT can become eventually full. Therefore, some branches may not allocate a free entry when resolved as mispredicted. For our particular implementation we add a special BCT entry, called *Lost Intvl*, to track the ROB region which cannot be verified. Clearly, this implies some coverage loss. Whenever a branch mispredicts and the BCT is full, it fills the *Lost Intvl* entry. In the case that the entry is already in use, we update its ROB start field f from the *Lost Intvl* entry with the branch position if the branch is older than the branch contained in the *Lost Intvl* entry. Similarly, the ROB tail field l is updated in case it now points to a younger instruction.

Reducing the number of BCT entries.

Modern processors implement out-of-order resolution of branches. This gives us an opportunity to reduce the required number of entries in the BCT, since out-of-order resolution of branches may cause two BCT entries to have in common some wrong-path instructions. This may happen whenever, within a fetch stream, a branch resolves as mispredicted after a younger branch has already resolved as mispredicted and filled a BCT entry. This scenario will render useless the BCT entry of the mispredicted branch that was resolved first, as the current ROB tail must point to an instruction no older than the ROB end slot for the previous resolved branch. The BCT entry that needs to be filled for the later branch will clearly cover the other branch entry.

We take this opportunity to reduce the total number of BCT entries by determining whether a BCT entry is covered by the region of a given misprediction, and hence, whether we can reuse the entry or not. It is important to note that the invalidation of BCT entries does not require modifying the *Lost Intvl* entry: a mispredicted branch can allocate a new BCT entry despite it is located in the *Lost Intvl* region. The validation step will take care of that situation.

We show in Figure 6.6 the checks that we implemented for a given entry. The branch is located at position (w_j, j), where w_j is the wrap bit and j is its ROB entry. Similarly, the instructions under the shadow of that branch span until the ROB tail (w_t, t), where w_t is the current wrap bit. The BCT entry groups bogus instructions from ROB entry (w_f, f) to entry (w_l, l). In case $(w_t, t) \geq (w_l, l)$, and $(w_j, j) \leq (w_f, f)$ and the branch was not previously marked as bogus, the *nullify* signal is asserted. If the *nullify* signal is asserted, the entry contents can be nullified and replaced with the covering branch information. Otherwise, an idle BCT entry must be allocated and filled.

Despite these are complex checks, the total overhead is very small because few BCT entries may be needed to achieve high error coverage.

6.5.2 Bogus Region Validation

The end-to-end validation step is carried at two points: once instructions commit and once mispredicted branches resolve during execution. Typically, upon a branch misprediction, all younger instructions in the issue queue and ROB are identified, squashed (marked as bogus). Furthermore, wrong-path instructions in other resources are also nullified (to avoid executing them), and potentially cleared.

Figure 6.7 depicts the commit time validation process. As usual, the *commit logic* obtains the possible retiring instructions out of the ROB banks by means of the

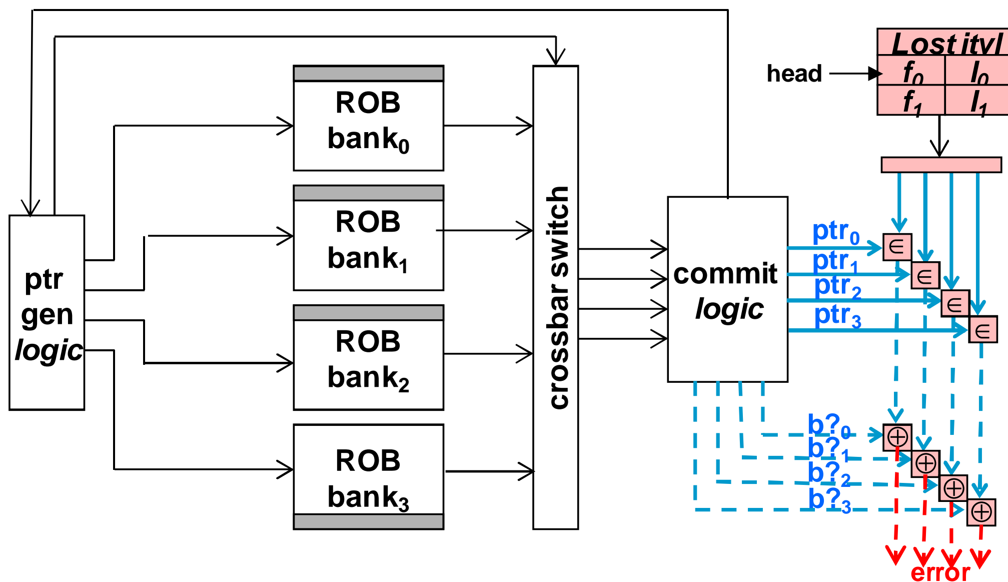


Fig. 6.7: BCT mechanism: extensions in the commit logic

Table 6.1: Commit time assertion checks for instruction squashing verification

Belongs to **BCT**?	Belongs to **Lost Intvl**?	**Error Check**
No	No	Must not be bogus (i.e. not in wrong path)
No	Yes	Nothing (coverage loss)
Yes	No	Must be bogus (i.e. in wrong path)
Yes	Yes	Must be bogus (i.e. in wrong path)

pointer generation logic (*ptr gen logic* in the figure). Then we obtain the oldest BCT entry (head pointer) and the *Lost Intvl* entry and check whether the instructions were marked as bogus or not. We take different actions depending on whether the commit pointer belongs to the BCT interval or *Lost Intvl*. We summarize them in Table 6.1. Note that as explained before, an instruction in the domain of *Lost Intvl* cannot be checked against errors despite it updates the architectural state. Hence, coverage loss is suffered.

Our mechanism detects a wrong instruction nullification scenario when a mispredicted branch finds an overlapping bogus region in the BCT (see *overlaps* and *error* signals in Figure 6.6). For instance, if a mispredicted branch spans from entry (w_j, j) to entry (w_t, t), and a BCT entry keeps the interval (w_f, f) to (w_l, l), then the branch must have been marked as bogus if $\overline{(w_t, t) < (w_f, f) \lor (w_j, j) > (w_l, l)}$. In case the *overlaps* signal is asserted but the instruction is not marked a bogus, an error has been detected.

BCT entries are freed as the ROB head advances. Every BCT entry whose ROB tail field l points to an instruction that is older than the instruction contained in the current ROB head, marks its valid $v?$ bit as false.

6.5.3 Microarchitectural Changes

A detailed list of the microarchitectural changes follows (assuming E is the number of bits required to indicate a ROB position jointly with its wrap bit).

- **BCT.** We implement the BCT table within the ROB module. The BCT consists of B entries, and each one of them requires 2 fields of E bits. Also, every entry has a valid bit indicating whether it is idle or in use.
- **Commit Logic.** Additional control logic is needed to access the BCT table during instruction commit time. This includes a BCT head pointer of $\log_2 B$ bits to indicate the youngest in-flight bogus region. Also, we need to provide the BCT with as many read ports as the number of committing instructions. Finally, we need extra logic to perform the assertion checks listed in Table 6.1. This includes two comparators per committing instruction to determine whether its ROB entry belongs to a wrong-path or not (2 comparators of E bits).
- **Branch Execution Stages.** The branch ROB entry and the current ROB tail must be obtained during execution in order to check and/or update the BCT table. Since the instruction ROB entry is read upon instruction issue, we just need an extra read port for the ROB tail pointer. Furthermore, we need four comparators of E bits per BCT entry, as shown in Figure 6.6. Two of them are used to compute the *cover* signal and the other two are used to compute the *overlap* signal.

6.6 Evaluation

This section presents a detailed evaluation of the proposed end-to-end techniques. We evaluate them in terms of area, power and coverage for our baseline processor.

6.6.1 Coverage Results

From a global perspective, our previous studies [205] based on AVF analysis indicate that all the hardware involved in implementing control flow recovery functionalities represents 11.40% of the SDC FIT rate caused by soft errors (excluding protected

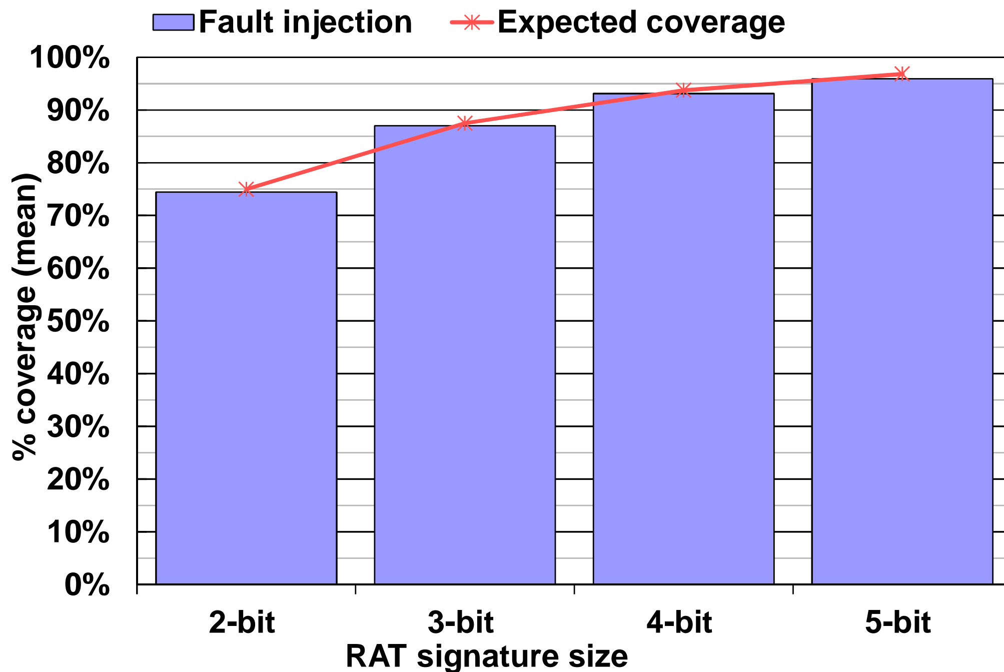


Fig. 6.8: Coverage for end-to-end RAT state signatures

structures, like caches, TLBs, etc.). Furthermore, 4.56% of the baseline processor area is exposed to other sources of failures, including wear-out, design and hard faults (again, excluding protected structures). By analyzing the microarchitectural blocks and by means of fault vulnerability studies (as described in Section 4.1.3), we have determined that the proposed technique is able to potentially cover 3.86% of the baseline processor area, and to potentially target 9.65% of the SDC SER FIT.

Given these area and potential soft error rate targets, actual error detection coverage has been obtained by means of error injection, as described in Chapter 4. For each SPEC benchmark, we perform 1000 effective fault injections for each class of error (see Section 6.3) independently. For every error class we inject errors in two manners: some faults are modeled as a flip of storage information and other faults are modeled as control logic misbehavior (by introducing bugs in the simulator). For both cases, we allow the fault to propagate and check at commit time and at RAT recovery time whether the fault is detected or not. For example, mapping information or bogusness information stored in the ROB is modeled as bit flips, whereas the wrong ROB walk failure scenario is modeled by implementing buggy code in the simulator. Each fault has been randomly injected during the ten first million executed instructions, after the warm-up period. Each experiment has been allowed to run for 100M instructions, as described in Chapter 4.

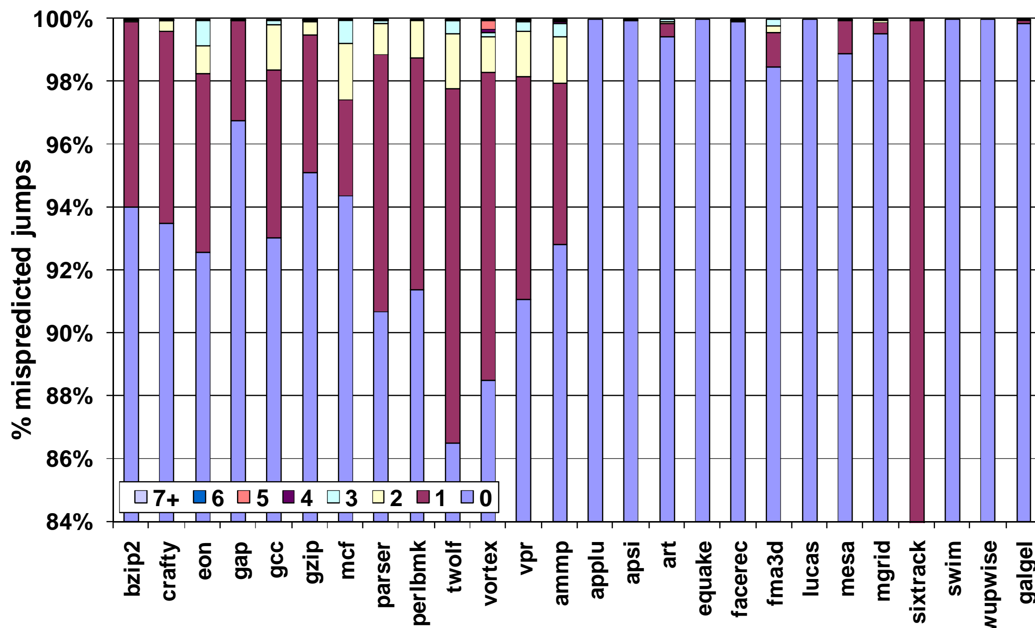


Fig. 6.9: Breakdown of number of younger resolved bogus regions for each mispredicted branch

RAT State Recovery

First, we have evaluated the ability of our technique to detect the injected errors related to the state recovery process for different sizes of the *RAT state signatures*. Reliability techniques based on signatures do not have perfect error coverage because of signature aliasing. Given a signature, the probability to match another will depend on the total number of signatures and the way they are generated. An error may not be detected if the signature observed when there is an error is the same as the expected one. When using n bits to encode the signature, the average case probability of having aliasing is $\frac{1}{2^n}$, assuming they are uniformly distributed. Hence, the expected *error detection coverage* in this case would be $1 - \frac{1}{2^n}$.

Figure 6.8 shows the expected coverage when RAT signatures are uniformly distributed, and the actual error detection coverage achieved through error injection experimentation. The achieved error detection coverage is very close to the expected one for all the considered signature sizes. On average, the difference with respect to the expected coverage is -0.58%, -0.47%, -0.60% and -0.94% for 2, 3, 4 and 5-bit RAT signatures, respectively. Specifically, 3 bit signatures allows detecting 87.03% of all the failures on average across benchmarks, whereas when moving to a 4-bit RAT signature we are able to increase the coverage to 93.15% on average. In all these cases, the errors are detected timely and without polluting the architectural state.

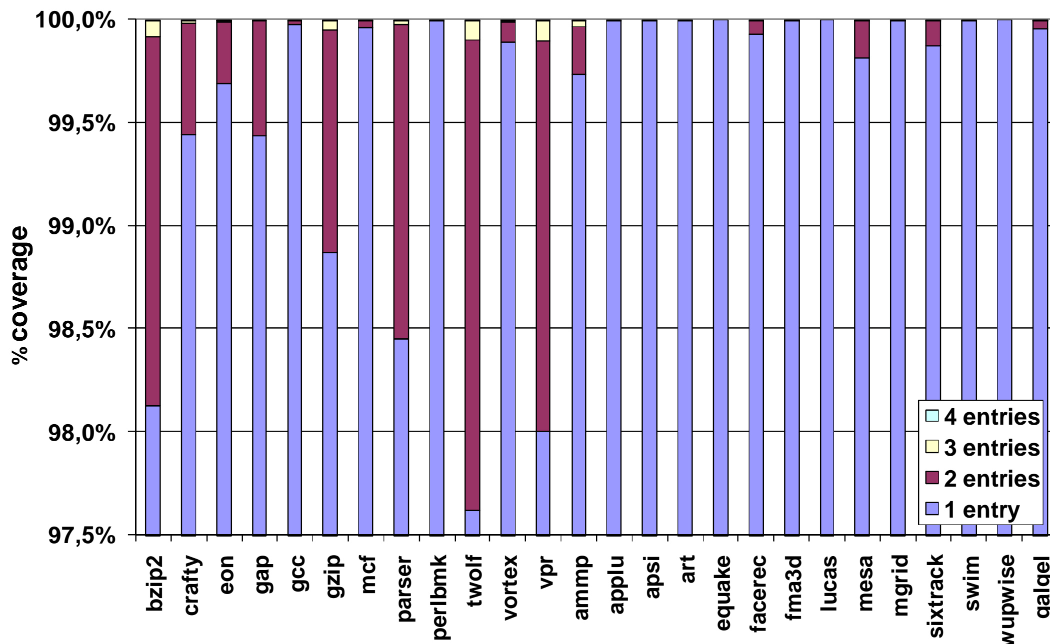


Fig. 6.10: Coverage for identification of control-flow dependent instructions (1 to 4 BCT entries)

Squashing of Control-Flow Dependent Instructions

We have conducted a similar analysis for the the technique shown in Section 6.5. In order to achieve 100% error coverage we would need as many BCT entries as ROB entries, because the most pessimistic scenario would be the one where each in-flight instruction is a mispredicted branch and they are resolved in age order. However, on average, there is a small number of mispredicted branches in flight and therefore, few BCT entries will be probably needed most of the cycles.

First, we have quantified how the out-of-order resolution of mispredicted branches allows reusing existing BCT entries. Figure 6.9 shows the number of younger resolved bogus regions that valid mispredicted branches would find upon resolution. On average, roughly 5% of the branches do not need to allocate a new BCT entry because they can reuse an existing one due to out-of-order execution, which relieves some pressure from the BCT (7.72% on average for SPECint and 2.18% for SPECfp).

Now, we take a look to the coverage of our technique for different number of BCT entries. Figure 6.10 shows a stacked chart indicating the coverage achieved. Our results show that on average, 1 entry obtains 99.12% and 99.94% *error detection coverage* for SPECInt and SPECfp, respectively. When moving to 2 entries, coverage raises to 99.96% and 99.99%, respectively. A BCT of 3 entries is able to provide 99.99% coverage, on average. Most of the cycles we would not need more than 1 or 2 entries in the tracking mechanism. Hence, the hardware overhead and the table management

complexity can be minimized.

6.6.2 Overheads

This section details the impact of our techniques in terms of delay, power and area (by following the methodology described in Section 4.1.4).

Delay

From an implementation perspective, RAT signature generation is performed in a staged manner during the two halves of the rename cycle. At the beginning of the first half, the current physical register ids are already available, and therefore the computation of $h(I_i.ldst, I_i.pdst)$ can happen in parallel for every instruction i. In addition, a partial computation of the RAT signature takes place, where $S_{i-1} \oplus h(I_i.ldst, I_i.pdst)$ is performed for every instruction i in the rename bundle, generating S'_i and requiring a global delay of $1 + W$ XOR gates, where W is the rename width. During the second half of the rename cycle, the previous physical register ids are already available, and $h(I_i.ldst, I_i.prev)$ is computed in parallel for every instruction i. In addition, $acc_i = \bigoplus_{j=0}^{j=i} h(I_j.ldst, I_j.prev)$ is computed for every instruction i in the rename bundle. The final RAT signature for every instruction i is generated as $S_i = S'_i \ominus acc_i$, needing a global delay of $1 + W$ XOR gates.

We have analyzed any possible impact by means of the area-power-delay framework described in Section 4.1.4. Our Wattch studies indicate that for 2, 3, 4 and 5 bit RAT signatures no impact is introduced into the delay: the reason is that access to the big rename tables during the two halves of the rename cycle dominates the total delay, as noted by Palacharla [140].

Signature re-generation logic piggybacks on the ROB walk logic. As described in Appendix A.2, during the first half of every RAT recovery cycle, the existing logic obtains from the ROB the register mapping of as many instructions as available ROB read ports-banks (in our case up to 4 instructions can be committed per cycle). Furthermore, every f^{-1} block grabs this register mapping information and starts re-computing the signature for every instruction (as described in the previous paragraph). This step starts during the second half of the cycle, and finishes in the first half of the next cycle, in parallel to the frontend RAT updates.

Regarding the `BCT`, it is filled upon branch misprediction. Given that a minimum of one cycle is spent in recovering the frontend RAT (for the case when the checkpoint RAT is copied into the frontend RAT, with no register mapping updates) and that

Table 6.2: Area and power overheads. `n`b SGN stands for `n`-bits RAT state signatures.

Block	Area				Power			
	%	Δ%			%	Δ%		
	Original	3 bits	4 bits	5 bits	Original	3 bits	4 bits	5 bits
`Bypass`	5.28	0.00	0.00	0.00	4.97	0.00	0.00	0.00
`FUs`	17.98	0.00	0.00	0.00	13.73	0.00	0.00	0.00
`L2$`	18.15	0.00	0.00	0.00	2.72	0.00	0.00	0.00
`Rename`	2.33	0.14	0.19	0.24	5.27	0.14	0.19	0.24
`IQ`	3.89	2.44	3.25	4.07	6.73	0.58	0.77	0.96
`RF`	2.92	0.00	0.00	0.00	8.10	0.00	0.00	0.00
`D$`	15.02	0.00	0.00	0.00	12.73	0.00	0.00	0.00
`ROB`	2.54	0.03	0.04	0.05	9.85	0.03	0.04	0.05
`Alloc`	1.69	0.40	0.50	0.63	1.99	0.50	0.66	0.83
`LSQ`	7.11	0.00	0.00	0.00	3.02	0.00	0.00	0.00
`Frontend`	23.09	0.00	0.00	0.00	24.17	0.00	0.00	0.00
`CLK`	N/A	N/A	N/A	N/A	6.72	0.11	0.14	0.18
Total	100.0	**0.11**	**0.14**	**0.18**	100.0	**0.07**	**0.09**	**0.11**

the branch is not allowed to retire until the recovery is complete, we can check and update the `BCT` during the first and second half of the recovery cycle.

Area and Power

We have evaluated the area and power introduced by the hardware needed to implement our runtime validation techniques. To do so, we have extended our power and area models as described in Chapter 4.

Left-hand side of Table 6.2 summarizes the area overhead for the end-to-end RAT state validation mechanism. We show in the first column the contribution of every block to the total processor area. Columns 3 to 5 show the extra area overhead when using different number of bits for the RAT signatures. The results show that the overall area increase is small. When using 3-bit signatures (8 different signatures), the core area increase is 0.11%. Columns 3 and 4 show the area overhead when using 16 and 32 different signatures. With respect to the core, an area overhead of 0.14% and 0.18% is required. Note that increasing the signature size affects the Issue Queue, Rename and ROB area. The issue queue must be enlarged because every instruction holds its own RAT signature. Moreover, every RAT must be extended in order to accommodate its own signature. Adding the signature generation / regeneration control logic does not impact much the area. The rename block sees an area overhead that ranges from 0.14% to 0.24%, primarily caused by the need to store the signatures

Table 6.3: Area and power overheads. ne BCT stands for n BCT entries.

Block	**Area**				**Power**			
	%	Δ%			%	Δ%		
	Original	1e BCT	2e BCT	3e BCT	Original	1e BCT	2e BCT	3e BCT
Bypass	5.28	0.00	0.00	0.00	4.97	0.00	0.00	0.00
FUs	17.98	0.00	0.00	0.00	13.73	0.00	0.00	0.00
L2$	18.15	0.00	0.00	0.00	2.72	0.00	0.00	0.00
Rename	2.33	0.00	0.00	0.00	5.27	0.00	0.00	0.00
IQ	3.89	0.00	0.00	0.00	6.73	0.00	0.00	0.0
RF	2.92	0.00	0.00	0.00	8.10	0.00	0.00	0.00
D$	15.02	0.00	0.00	0.00	12.73	0.00	0.00	0.00
ROB	2.54	1.73	2.59	3.46	9.85	2.35	3.53	4.71
Alloc	1.69	0.00	0.00	0.00	1.99	0.00	0.00	0.00
LSQ	7.11	0.00	0.00	0.00	3.02	0.00	0.00	0.00
Frontend	23.09	0.00	0.00	0.00	24.17	0.00	0.00	0.00
CLK	N/A	N/A	N/A	N/A	6.72	0.04	0.07	0.09
Total	100.0	**0.04**	**0.07**	**0.09**	100.0	**0.23**	**0.35**	**0.47**

for the frontend and checkpoints RATs. The control logic for signature generation is minimal, as well as the control logic for the regeneration at the commit stages. Our studies show that it ranges from 0.03% to 0.05% for 3 to 5 bit signatures (with respect to ROB block, that implements the RAT state recovery functionality). In our particular design, the $+$ and $-$ functions are implemented with the bitwise XOR function (whose inverse is itself) and hence the implementation consists of a shallow and small XOR tree. Results in Table 6.2 also show that 3-bit RAT signatures have a 2.44% area impact over the issue queue. When moving to 4-bit signatures, area overhead moves to 3.25%. Clearly, 3 and 4-bit signatures are the most desirable ones: for larger bit counts error coverage does not increase at the same pace, but area overhead increases almost linearly for some structures.

For the BCT scheme, left-hand side of Table 6.3 also summarizes the area overhead for our baseline processor model. The BCT structure is the new hardware block required to implement the mechanism, with minimal additional modifications added to the ROB block. The area overhead mainly comes from the BCT structure and has been accounted as extra area in the ROB block. For 1, 2 and 3 BCT entries, the area overhead moves to 1.73%, 2.59% and 3.46% respectively. This translates into a global area requirement that spans from 0.04% to 0.09%.

When combining both techniques, the results in Table 6.4 also show that the overall area increase is small.

Table 6.4: Area and power overheads. n*b* SGN stands for n-bits RAT state signatures and n*e* BCT for n BCT entries.

Block	Area					Power				
	%	Δ%				%	Δ%			
	Original	3b SGN 1e BCT	3b SGN 2e BCT	4b SGN 2e BCT	5b SGN 2e BCT	Original	3b SGN 1e BCT	3b SGN 2e BCT	4b SGN 2e BCT	5b SGN 2e BCT
Bypass	5.28	0.00	0.00	0.00	0.00	4.97	0.00	0.00	0.00	0.00
FUs	17.98	0.00	0.00	0.00	0.00	13.73	0.00	0.00	0.00	0.00
L2$	18.15	0.00	0.00	0.00	0.00	2.72	0.00	0.00	0.00	0.00
Rename	2.33	0.14	0.14	0.19	0.24	5.27	0.14	0.14	0.19	0.24
IQ	3.89	2.44	2.44	3.25	4.07	6.73	0.58	0.58	0.77	0.96
RF	2.92	0.00	0.00	0.00	0.00	8.10	0.00	0.00	0.00	0.00
D$	15.02	0.00	0.00	0.00	0.00	12.73	0.00	0.00	0.00	0.00
ROB	2.54	1.76	2.62	2.63	2.64	9.85	2.38	3.56	3.57	3.58
Alloc	1.69	0.40	0.40	0.50	0.63	1.99	0.50	0.50	0.66	0.83
LSQ	7.11	0.00	0.00	0.00	0.00	3.02	0.00	0.00	0.00	0.00
Frontend	23.09	0.00	0.00	0.00	0.00	24.17	0.00	0.00	0.00	0.00
CLK	N/A	N/A	N/A	N/A	N/A	6.72	0.15	0.17	0.21	0.24
Total	100.0	**0.15**	**0.17**	**0.21**	**0.24**	100.0	**0.30**	**0.42**	**0.44**	**0.46**

When using 3-bit signatures (8 different signatures), and 1 BCT entry, the core area increase is 0.15%. For the same number of signatures but using 2 BCT entries, overhead moves to 0.17%. Columns 3 and 4 show the area overhead when using 4-bit and 5-bit RAT signatures (16 and 32 different signatures, respectively) and a BCT of 2 entries. With respect to the core, an area overhead of 0.21% and 0.24% is required. Note that the ROB and the Issue Queue blocks are the ones with bigger changes.

We have also evaluated the total peak dynamic power increase due to the proposed solutions. For a RAT signature scheme that uses 8 different signatures, we obtained a 0.07% power increase with respect to the whole core, as shown in the right-hand side of Table 6.2. Minimal peak power overhead of 0.09% and 0.11% is introduced when using 16 and 32 signatures, respectively.

Regarding the BCT technique, the global peak dynamic power penalties are a bit higher. The current BCT entry is continuously accessed by each instruction being committed and two age comparators are permanently being active. Furthermore, every branch in the worst case needs to access all the BCT entries for both error detection and entry reuse detection. These facts translate into an increase in the ROB dynamic peak power that ranges from 2.35% to 4.71% (Table 6.3).

However, given that the ROB power represents the 9.85% of the total power consumed, the impact is slight at the core level: global power overheads of 0.23%, 0.35% and 0.47% are required for 1, 2 and 3 BCT entries, respectively.

For both techniques combined together, results in the right-hand side of Table 6.4 shows that for a scheme using 8 different signatures and 1 BCT entry, we obtained a 0.30% dynamic peak power increase with respect to the whole core. Peak power overhead of 0.42%, 0.44% and 0.46% is introduced when using 2 BCT entries and 8, 16 and 32 signatures, respectively. As it can be observed, the biggest contributor to the global power are the BCT entries that are continuously accessed.

6.7 Conclusions

Modern processors devote a significant amount of hardware and complex control logic to provide an efficient implementation of control flow recovery required by branch prediction, exceptions, interrupts and other speculative performance solutions. To our knowledge, we have proposed the first solution to protect in a targeted way the whole control flow recovery control logic. It provides continuous runtime error detection for multiple sources of failures (including design bugs). Coupled with the control-flow checking baseline RAS features described in Section 4.2, control flow can therefore be protected in an effective manner.

Our solution exploits high-level microarchitectural invariants to protect the control flow recovery logic in an end-to-end way.

The proposed technique is able to potentially target 9.65% of the SDC SER FIT and cover 3.86% of the baseline processor area against other faults (excluding protected structures). Given these area and potential error rate targets, our evaluations show that the techniques provide an excellent error coverage against all types of injected faults, without polluting the architectural state. For failures affecting the RAT recovery logic, we can achieve different coverage ratios by changing the amplitude of our error detection signatures. We have evaluated signatures sizes varying from 3 bits to 5 bits, and error coverage has proven to span from 87.03% to 95.93%, on average. For failures affecting the identification of wrong-path instructions, coverage is always above 98% on average.

No performance slowdown is introduced and area and dynamic peak power overheads with respect to the core are rather modest (little impact is introduced on the hardware structures implementing control flow). For our technique targeting the detection of errors in the RAT state management logic, area and dynamic peak power overheads for the biggest signature scheme are 0.18% and 0.11%, respectively. Our second technique just requires areas overhead between 0.04% and 0.09%, while dynamic peak power overhead stays around 0.23% and 0.47%.

CHAPTER 7

MEMORY FLOW VALIDATION

7.1 Introduction

The Load-Store Queue is one of the most complex structures in a modern out-of-order processor that is in charge of allowing out-of-order execution of memory instructions while at the same time guarantees that all these memory instructions update the processor state as if they were executed in program order. The LSQ holds in-flight memory instructions and supports simultaneous associative searches to honor memory dependencies.

Unfortunately, most of the existing global hardware (Section 3.1) or software reexecution-based techniques (Section 3.4) cannot protect the memory dataflow because loads and stores from a redundant execution get their data from the original execution, which can be potentially faulty. To our knowledge, no specific solutions exist for protecting the memory dataflow of a processor in a targeted and cost-effective manner. Furthermore, access time of the LSQ is critical because it is a component of the load-to-use latency [144, 172]; therefore, error detection mechanisms that are not intrusive and do not increase the LSQ complexity or delay [193] are needed.

In this chapter we propose a simple yet effective idea for validating that the LSQ logic performs correct out-of-order memory ordering. More important, our solution allows implementing different schemes with varying degree of error coverage, performance overhead and design complexity. Our technique runs in parallel to the LSQ and relies on a small cache-like structure that keeps track of the last producer (store) for each cached address. Our results show that we can achieve up to 99.91% coverage with a very small area increase with respect to the LSQ. Moreover, several sources of failures can be corrected by flushing and re-executing the pipeline because faults are detected before the architectural state is irrevocably updated.

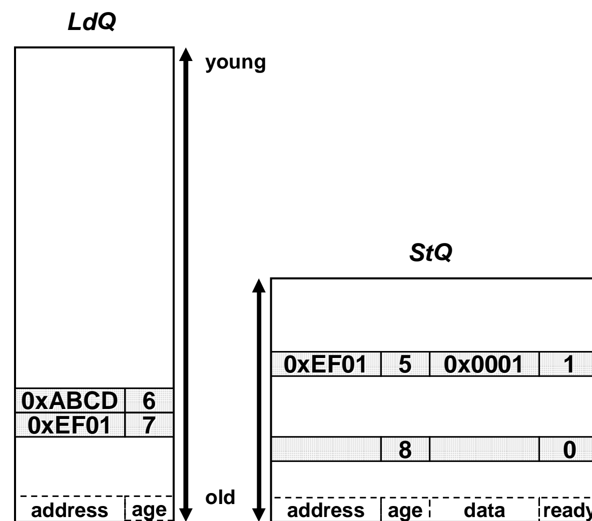


Fig. 7.1: A typical LSQ configuration (simplified)

The rest of the chapter is organized as follows. Section 7.2 reviews the LSQ architecture and Section 7.3 analyzes the kind of failures that it may experience. Section 7.4 presents the general idea for validating the correct functioning of the LSQ. Section 7.5, Section 7.6 and Section 7.7 describe three different implementations of the technique. We discuss some results in Section 7.8. Finally, we offer our concluding remarks in Section 7.9.

7.2 Load-Store Queue: Overview

Load-Store Queues are used in modern out-of-order processors to allow early out-of-order execution of memory instructions to increase overall performance [56]. However, a LSQ must guarantee that all memory instructions will update the architectural processor state as if they were executed in program order. To implement this, the LSQ performs address matching and age prioritization to detect read-after-write memory dependencies among instructions residing in the LSQ, and their possible violations. This process is performed both for loads and stores. Write-after-read and write-after-write dependencies are ensured by committing stores in program order.

Figure 7.1 shows a possible LSQ implementation. LSQs are typically divided in three main components: (i) a CAM & SRAM queue that holds information of in-flight loads (called load buffer or load queue-*LdQ*), (ii) a CAM & SRAM queue that holds information of in-flight stores (called store buffer or store queue-*StQ*), and (iii) a complex control logic that uses address, age and size information (*age* and *size* fields in the load and store queues) to allow out-of-order execution while ensuring correct

memory ordering. A LSQ is often managed as a circular queue. Entries in the *load queue* and *store queue* are allocated to instructions in consecutive and program order. Similarly, LSQ entries are deallocated in program sequential order at commit time.

To achieve out-of-order execution, high-performance LSQ designs support *load bypassing* and *load forwarding*. *Load bypassing* happens when a load is allowed to execute by ignoring older in-flight stores (and loads) in case no store address aliasing exists (conservative memory disambiguation design) or in case it is predicted that there will not be aliasing (speculative memory disambiguation design). *Load forwarding* happens when a load executes and finds a memory dependence with an older store residing in the LSQ. In this scenario the load is directly satisfied from the youngest older matching store buffer entry if the address is valid and the data is available in the store buffer. As a consequence, loads do not need to wait to execute until all previous stores are committed to the cache. Speculative memory disambiguation is the common choice in modern processors: load bypassing or load forwarding are allowed even when *not* all prior store addresses have been resolved. This option can result in a store S being executed and finding younger and already executed loads with memory dependencies (matching addresses). If any of these matching loads was not forwarded by a store younger than store S, then a memory ordering violation is detected and corrected. We refer to *memory ordering violation detection* as *memory ordering violation* for the sake of brevity.

We next detail how *load bypassing*, *load forwarding* are implemented and how *memory ordering violations* are detected and corrected.

Load Bypassing and Load Forwarding Once a *load instruction (load)* is ready (i.e. its source operands have been computed), it is issued to the execution units and as a result its effective address is calculated. This effective address is then written-back into its allocated load queue entry. After that, the LSQ will access the data cache to obtain a value for that load and in parallel it will perform an associative search in the *store queue* among the older stores. If an address match is found in the *store queue*, then the youngest older matching store value is retrieved from the *store queue* and the value coming from the data cache is ignored (it is stale). If the store value is not ready upon a match, then the load operation is frozen until it is available.

Memory Ordering Violation Detection Similarly, when a *store instruction (store)* executes, its effective address is computed and is written-back into its store queue entry. From that moment, and no later than when the store instruction commits/retires, the LSQ will perform an associative search in the *load queue* in order to determine if there has been a read-after-write dependence violation for that store address. If this

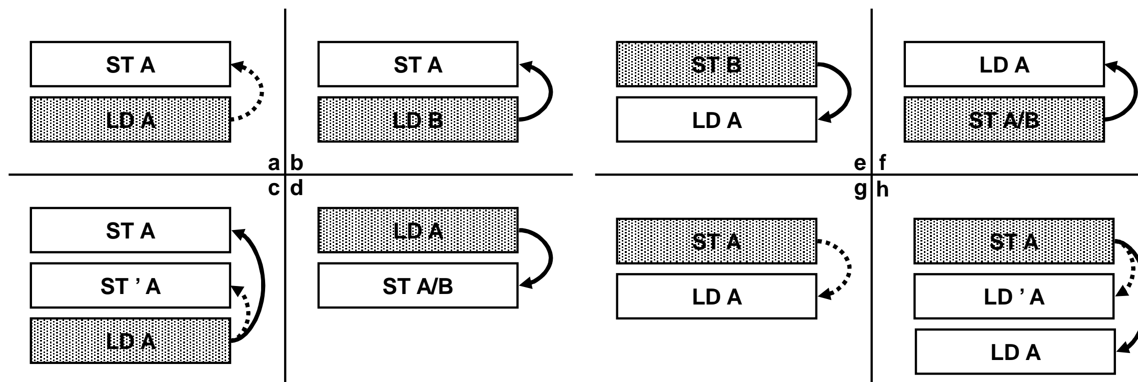


Fig. 7.2: Failure scenarios related to LSQ operation

is the case, younger instructions starting from the oldest matching load (including it) need to be recovered. Recovery is usually implemented with a pipeline flush starting from the oldest matching load. Implementations where all consuming loads and their recursively dependents are replayed are extremely costly from an implementation perspective and not practical due to the rarity of these events. Furthermore, processors implement different flavors of memory dependence predictors to reduce the occurrence of *memory ordering violations* (see Appendix A.2). For the rest of the chapter, and without loss of generality, we will assume a recovery mechanism based on flush and re-execution.

7.3 Load-Store Queue Failures

Techniques like parity, ECC or residues can detect errors in SRAM-like structures (for example, the data value in the *store queue* entries), or even in CAM data (such as the addresses in the *load queue* and *store queue* entries). In fact, the proposed end-to-end register dataflow validation technique described in Chapter 5 is used to protect the LSQ addresses and values in a cheap and unified manner. However, control logic like address and size comparators, matchlines, age priority encoders, or dynamic storage such as the *ready bits* cannot be protected with coding techniques. Incoming bits can match against a CAM entry in the presence of a fault when it should really have mismatched (false-positive case). Alternatively, incoming bits may not match any CAM entry, although they should have really matched (false-negative case), age comparators can operate incorrectly, priority selectors based on the output of age comparators can fail, etc.

Figure 7.2 shows a taxonomy of the different kind of failures that a LSQ may suffer as a result of different sources of failure (including soft errors, design bugs, hard faults, degradation, etc). A broad spectrum of high-level functional failures are

included, resulting from multiple low-level fault scenarios. Instructions are listed in program order top-down. Shadowed instructions are the ones that execute and cause the failure. Straight lines show the actions taken, whereas dashed lines represent the missing actions.

Cases *a* to *d* correspond to an incorrect determination of *store-to-load forwarding*. Cases *a*–*b* arise with address-related faults, whereas cases *c*–*d* appear due to faults in the age prioritization logic, bit flips in the *ready bits*, etc.

(a) A forwardable store-load pair is wrongly ignored. When the LD A resolves the address, it should obtain the data from ST A(already executed). However, the LSQ logic ignores this, and therefore, LD A gets the wrong data from the memory (cache).

(b) A store-load pair that does not access the same address is wrongly matched by the LSQ logic. In this case, LD B resolves the address and LSQ logic decides to forward the wrong data from a previous executed ST A.

(c) A load has a value wrongly forwarded from an older matching store when there is an additional matching store older than the load but younger than the forwarding store. The LD A should get the data from ST' A; however, due to a fault, it gets the wrong data from an older ST A.

(d) A store forwards a value to an older load. The LD A gets the wrong data from a store that appears later in program order.

Cases *e* to *h* correspond to an incorrect determination of *memory ordering violation* that are triggered when stores resolve their address. Cases *e*–*g* appear due to address-related faults, whereas cases *f*–*h* arise with fault in the age prioritization logic, etc.

(e) The LSQ logic wrongly identifies a memory ordering violation between a store with a younger already executed load having a different address. In the figure we show how a ST B invalidates a younger LD A. This case would result in an unnecessary pipeline flush, but it would not cause any failure.[1]

(f) The LSQ logic identifies a memory ordering violation between a store and an older already executed load. Similar to the previous case, this would result in an unnecessary pipeline flush.[2]

[1] Unless recovery is implemented by a mechanism that re-forwards the store value to the wrongly executed load, and re-executes all dependent instructions, committing wrong data.

[2] Unless recovery is implemented by a mechanism that re-forwards the store value to the wrongly executed load, and re-executes all dependent instructions, committing wrong data.

(g) A store matching with a younger already executed load is not detected by the LSQ logic. In this case, ST A should have invalidated LD A, but due to a fault in the LSQ logic, the LD A commits with wrong data.

(h) A store matches with more than one younger already executed loads but the prioritization logic does not perform the recovery for the oldest one. ST A should invalidate both LD A and LD' A. However, LD' A is not invalidated and commits with the wrong data.

7.4 LSQ Memory Ordering Tracking and Validation: General Idea

We present a low-cost solution that provides fine-grain error detection for the LSQ. Our approach is based on *verifying* the microarchitectural behavior of the LSQ by: *(i)* tracking the memory ordering imposed by stores and *(ii)* by validating that loads gets the data generated by the youngest previous matching store (in program order).

At the base of our approach we have a small cache-like structure called *Memory Order Validation Table (MOVT)* that is in charge *memory ordering tracking*. *MOVT* is indexed with memory addresses, updated by stores and read by loads. Each entry keeps a "store identifier" (*StID*), a small id that is written by the last store that updated the corresponding address.

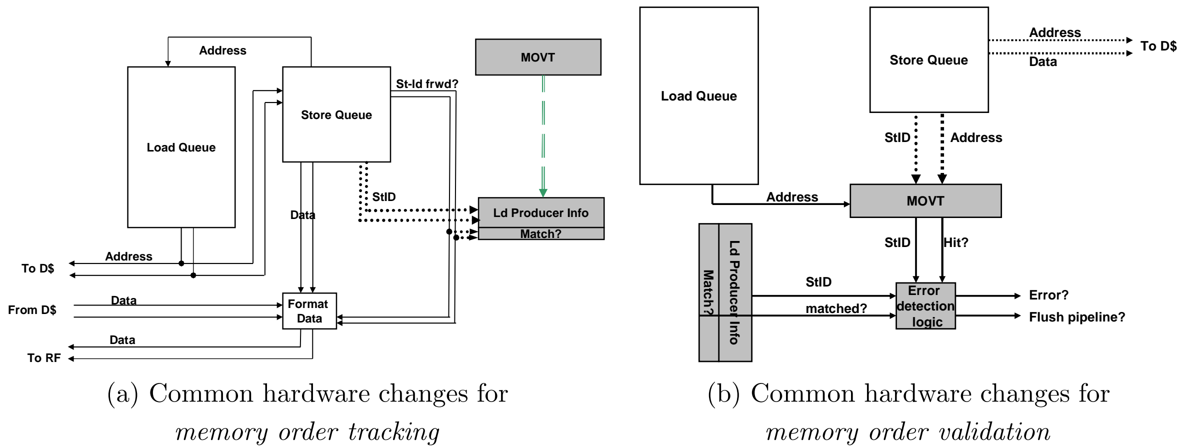
All stores update the *MOVT* when they commit. We leverage the in-order commit for validation purposes, as described in some memory disambiguation approaches [29]. Given that stores update the *MOVT* with their address and *StID* at commit time and in program order, the *MOVT* will hold a set of the references accessed during the program execution together with their last producer id.

During their lifetime, loads will acquire a *StID* from their producer, which is also necessary for *memory ordering tracking*. Depending on the way or stage where loads obtain the *StID* , we will have different implementations with different trade-offs for error coverage, performance, area and design complexity.

Memory ordering validation is checked for loads. Validation is done during commit-time and is common to all of the techniques described here: loads access the *MOVT* with their address and compare the producer *StID* obtained during execution with the one stored in the *MOVT* .

In the next subsections we detail the common hardware changes needed to implement our proposal. Then, we describe how to use the *MOVT* to track the memory order and validate the *LSQ*'s behavior. Finally we discuss error recovery.

Next, in Sections 7.5 to 7.7 we explore three possible different implementations



(a) Common hardware changes for *memory order tracking*

(b) Common hardware changes for *memory order validation*

Fig. 7.3: *MOVT* hardware template

of this global idea. These three designs differ in the way loads acquire the *StID* from their producers, providing different coverage-slowdown-complexity trade-offs. Section 7.5 describes one scheme where loads obtain their producer *StID* at the execution stages, Section 7.6 characterizes a scheme with minimal design complexity and intrusiveness where just forwarded loads obtain a *StID*. Finally, Section 7.7 explores one scheme where address prediction is used to speculate on loads *StIDs* as a way to potentially remove timing constraints when accessing the *MOVT*.

7.4.1 Microarchitectural Changes

Our mechanism requires minimal extensions to the LSQ organization and logic. These hardware changes are global to any implementation, but each particular LSQ implementation may require extra specific changes.

The main hardware involved is shown in Figure 7.3. LSQ original logic is represented by thin lines and white boxes, whereas the new hardware is shown in thick lines, dotted lines and grey boxes.

As mentioned, the core of our technique is the *Memory Order Validation Table (MOVT)* that tracks the memory ordering. For each store queue entry, we add a field to store its identifier (the *StID*). Similarly, each load queue entry holds a field that indicates the producer of the value consumed during the load lifetime (i.e. *StID*). We call this field *Load Producer Info*, or *prodID*. We also add an extra *match? bit* to keep record of whether a store-to-load forwarding has happened (if the existing LSQ implementation does not have it). For visual simplicity Figure 7.3 shows *Ld Producer Info* (*prodID*) and *match?* as separate blocks, but actually they are simply extra fields inside the *load queue*.

Our approach to error detection in the LSQ works in parallel with the current LSQ logic. Therefore, we do not expect any impact in the critical path.

7.4.2 LSQ Memory Ordering Tracking

We now give a global view of how we track the memory ordering through the *MOVT*. Section 7.5, Section 7.6 and Section 7.7 will detail how this general implementation accommodates to particular *MOVT* design choices.

Allocate Stage

At allocate, each store is given a *StID*, which is later stored in its store queue entry. In order to avoid aliasing among different *StIDs*, we only need to guarantee that there are as many different *StIDs* as possible *live stores*. We need as many *StIDs* as the maximum number of entries in the *MOVT* or store queue: few bits suffice to encode the *StID* (e.g. 5 bits). *StIDs* are generated by incrementing a counter each time an instruction is allocated in the LSQ. This step is common across all the implementations.

ProdID Acquisition Stage

At some point during the load lifetime, it will access the *MOVT* to obtain its producer *StID* and will store it in the *Ld Producer Info* field (dashed green line flow in Figure 7.3(a)). This step is implementation dependent: loads can obtain their *prodID* at different pipeline stages, resulting in different instantiations of the technique. Since the *MOVT* size is bounded, it is possible that a load misses in the *MOVT* during *prodID* acquisition. In this situation, the *MOVT* will return a special *NULL value*.

Execution Stage

This is a common step for all the implementations, and has been depicted in Figure 7.3(a) (dashed grey line flow). Once a load is issued, a regular LSQ would access the data cache and perform an associative search in the store queue looking for possible producers.

When a store-to-load forwarding situation is detected by the LSQ logic, the load writes the *StID* of the forwarding store in the load *prodID* field (potentially overriding any previous *prodID*) and also annotates in the *match? bit* that it has been

forwarded.[3]

7.4.3 LSQ Memory Order Validation

We now give a global view of how memory dataflow is validated through the *MOVT*.

Commit Stage

This is a common step for all the implementations, and has been depicted in Figure 7.3(b): when a store commits it updates its address reference in the *MOVT* with its own *StID*, to change the state of the tracking mechanism.

At commit time loads finally validate that they have obtained the data from the expected producer. We do so by comparing the *prodID* obtained during loads lifetime with the information stored in the *MOVT*. If a load's *prodID* field matches the *StID* stored in the *MOVT*, it means that everything went fine. Otherwise, it will indicate that there has been potential error in the memory dataflow.

Assuming an unbounded *MOVT*, comparing the *prodID* field with the *StID* grabbed at commit time is enough to validate the LSQ. However, for a finite *MOVT* some entries may get lost due to evictions. Therefore, it is possible that a load misses in the *MOVT* either during the *prodID* acquisition or at commit time. The *Error detection logic* in Figure 7.3(b) decides if there has been a fault in the LSQ logic. For each particular *MOVT* design choice, this logic will take different actions depending on the load *prodID*, its *match? bit* and the *StID* obtained from the *MOVT* at commit time. Specific details will be given later for each of the different implementations.

Example

We will illustrate how our mechanism works by using an example (shown in Figure 7.4). We depict the state of the load and store queues, as well as the state of the *MOVT* before -upper figures- and after -lower figures- store execution, store commit, load *prodID* acquisition and load commit. Figure 7.4(a) shows the changes in the state when a store executes.

[3] In case the LSQ implements *memory ordering violation detection* where offending loads and their dependents are replayed, further modifications are required. Stores detecting the *memory ordering violation* would forward their value together with their *StID* to the wrongly executed loads. Furthermore, the *match? bits* would be set and any *prodID* would be overriden with the stores' *StIDs*. As a consequence, *memory ordering violations* would ultimately behave as *load forwarding* scenarios. Nevertheless, these LSQ designs are extremely rare.

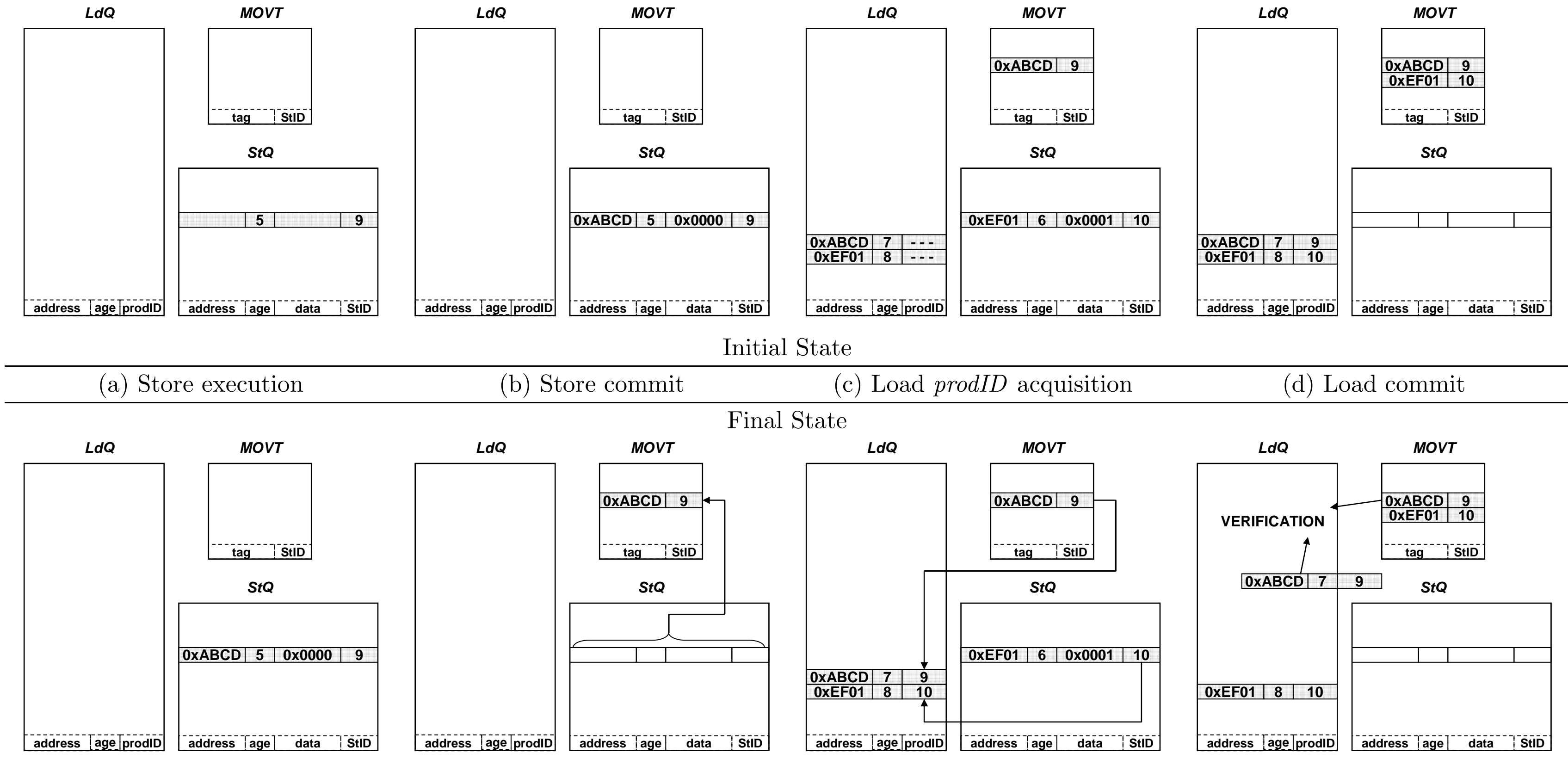


Fig. 7.4: Memory ordering tracking and validation: an example

At allocate, the store was given an entry in the store buffer and the *StID*. Once it executes, it resolves the address and data, updating only the store queue. When a store commits (Figure 7.4(b)), it releases its entry in the store buffer, and fills an entry of the *MOVT* with its *StID*. Note that the *MOVT* works as a cache, and hence, if at commit time it does not keep an entry holding information for the store address, another entry will be used (either a free one or evicting one). Top of Figure 7.4(c) shows that after the store with age 5 committed, one younger store and two younger loads entered the pipeline and computed their effective address. Specifically, the figure shows that the store with age 6 (store 6) wants to update address 0xEF01 with data 0x0001, and the loads 7 and 8 will access addresses 0xABCD and 0xEF01, respectively. Bottom of Figure 7.4(c) reflects that a load can obtain its *prodID* either from the *MOVT* or from the *store queue*: load 7 obtains its *prodID* exclusively from the *MOVT* (because there is no older in-flight store to address 0xABCD), whereas load 8 obtains it from the *store queue* at execution time. As a result, load 7 will receive the *StID* from store 5, and load 8 will get it from store 6. When a load commits, it releases the entry in the load queue and compares the *prodID* value with the corresponding *StID* in the *MOVT*. Figure 7.4(d) shows that after store 6 commits, load 7 hits the MOVT when committing and checks its *prodID*.

7.4.4 Failure Recovery

We rely on flushing the pipeline to restore correct state in the event of an error detection. This mechanism is already used to handle the recovery of wrong memory ordering detection in the load-store queue, or to handle branch misprediction recovery.

By flushing the pipeline we can recover from multiple sources of failures. Re-execution will start from the offending load, and therefore, since the pipeline and *MOVT* will be empty, the load will go directly to cache and *bypass* the LSQ logic.

For permanent / intermittent faults, we would like to disable/replace the affected hardware to avoid performance overheads when exercising a fault repeatedly. How this is done is out of the scope of this thesis. The real challenge is indeed locating the fault and diagnosing it, so that (i) the repair and reconfiguration mechanism can be conducted and (ii) to help validators understand the reason behind the recovered error. The diagnosis of failures in the LSQ will be covered in Chapter 8.

7.5 Design #1: *MOVT* Access at Execute

In this section we describe an implementation of the general design proposed in Section 7.4, that performs the prodID *acquisition* during load execution. This means

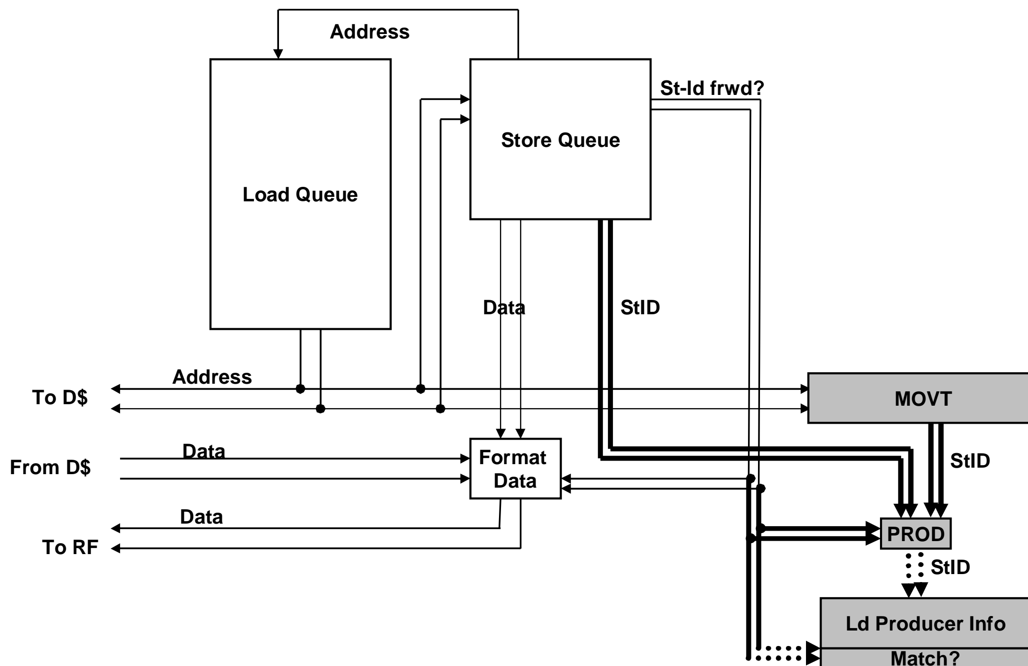


Fig. 7.5: *MOVT* hardware for design #1: prodID *acquisition* at execute time

that after a load has resolved its effective address, it will access the *MOVT* in order to obtain its producer *StID*.

ProdID Acquisition

Figure 7.5 shows how the *prodID* acquisition works in the execute stage. Once a load is issued, a regular LSQ would access the data cache and perform an associative search in the store queue looking for possible producers. The particularity of this design is that the load also accesses (in parallel) the *MOVT*. If there is no forwarding from the store queue, the *prodID* field of the executing load is filled with the *StID* obtained from the *MOVT*.

In case a load does not find a matching store either in the store queue or in the *MOVT* at execution time, its *prodID* field is updated with a special *NULL value*.

Validating Memory Ordering

When loads commit, they check the *MOVT* to see if there is information of the producer (store instruction) that produced their data. In the best case, the load will hit in the *MOVT* and will find the *StID* of the store that produced the data.

Table 7.1: Protocol when loads hit the *MOVT* at commit time (Design #1)

prodID	***match? bit***	**Action**
VALID	N/A	Check
NULL	N/A	Fix

Table 7.2: Protocol when loads miss the *MOVT* at commit time (Design #1)

prodID	***match? bit***	**Action**
N/A	Set	Fix
N/A	Unset	None

However, due to the limited space of the *MOVT* it may happen that a later store evicted the information.

Table 7.1 shows the different actions taken when loads effectively hit in the *MOVT* at commit. Notice that the *match? bit* is not important in this first case:

(i) The *prodID* field holds a valid *StID* if the load hit in the *MOVT* at the execute stage or obtained it through forwarding. We compare the *StID* stored in the *prodID* and the *StID* obtained from the *MOVT*. A mismatch indicates an error.

(ii) It may happen that the load missed in the *MOVT* at execute time. This means that the load checked the stores in the store queue and the *MOVT* without finding any match. If that is the case, the *prodID* field would have the *NULL* special value. At commit time, the *MOVT* holds a subset of the information stored in the *MOVT* and store queue at execute time. Therefore, a load that hits at commit time can only correspond to a failure. The load should have obtained a *StID* during execution, either through store-to-load forwarding case or through the corresponding entry in the *MOVT*.

Due to address aliasing, entries from the *MOVT* may be evicted, or some memory locations may have never been accessed by a store. As a consequence, some loads may miss in the *MOVT* at commit time when they check whether the *StID* they carry in the *prodID* is correct. Table 7.2 shows the different scenarios:

(iii) If the *match? bit* is set, it means that the store producing the data is very close. Thus, we would expect the load to hit when accessing the *MOVT* table. However, it is possible that a forwarded load misses in the *MOVT*, because the producer store could be evicted.

We observe that such scenario when a load misses the *MOVT* after getting the value through forwarding is very rare because most of the store-load matching

pairs are close to each other. Therefore, we consider this scenario as suspicious of a potential error and apply the correction mechanism conservatively (flush the pipeline and restart execution).

(iv) If *match? bit* is unset, the most likely situation is one where it was not forwarding store. In that case, the behavior would be correct in most of the cases.

However, it may be the *rare* case where there was a forwarding store which was not identified by the LSQ logic, the store updated the *MOVT* and that entry was evicted later on. As mentioned, the case where a load gets the data through forwarding and does not hit in the *MOVT* is very unlikely. Therefore, we opt for ignoring this case (at the expense of coverage), and no action is taken.

Potential Issues

For this implementation, the *MOVT* is accessed simultaneously from two different pipeline stages (the commit stage and the execute stage).

One of the possible problems with this approach is that depending on the processor layout it may be hard to accommodate the *MOVT* accesses within the existing processor timing restrictions. For some designs, if the ROB and LSQ are distant in the processor die, design efforts would be required to implement the technique. In Section 7.6 and Section 7.7 we explore two designs to alleviate this potential difficulty.

7.6 Design #2: Minimal *prodID* Acquisition

In this section we describe another instantiation of the general design proposed in Section 7.4, that targets design simplicity. This design exclusively performs *memory ordering tracking* for forwarded loads.

ProdID Acquisition

Figure 7.6 depicts the hardware necessary for *memory order tracking*. The scheme that is proposed here is the simplest implementation of the general idea presented in Section 7.4. For this design, no load will access the *MOVT* to obtain its *prodID* only forwarded loads will obtain the *prodID* from the *StID* entries in the store queue. Therefore, no specific *prodID* acquisition is conducted. A NULL value is kept in the *prodID* field for those loads that are not forwarded.

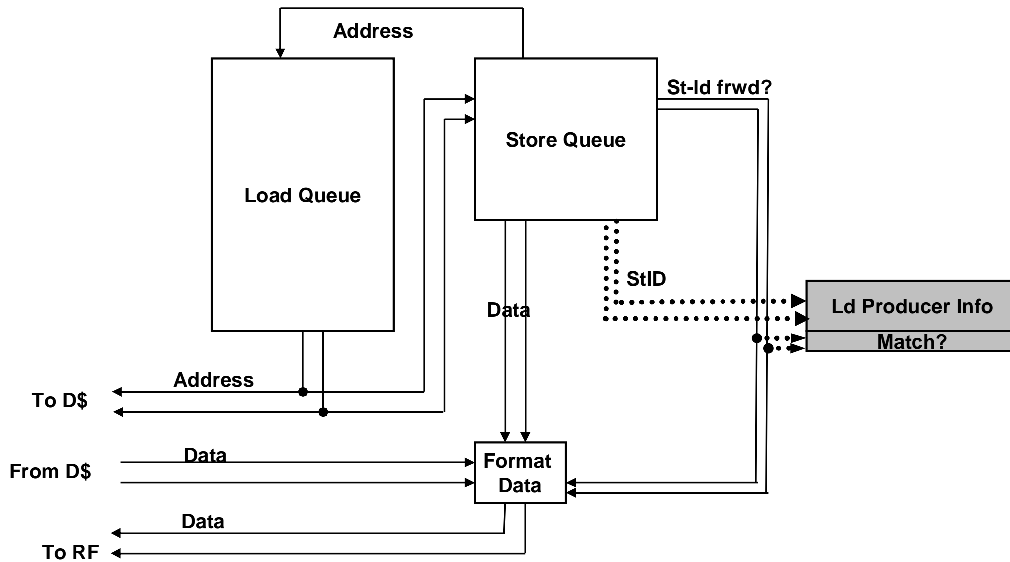


Fig. 7.6: *MOVT* hardware for design #2: minimal prodID *acquisition*

Table 7.3: Protocol when loads hit the *MOVT* at commit time (Design #2)

prodID	***match? bit***	**Action**
VALID	Set	Check
NULL	N/A	None

Validating Memory Ordering

This *minimalist MOVT* design takes a different approach for validating the LSQ. Given that loads will only have a valid *prodID* in case they have been forwarded their values (this is, their *match? bit* is set), we will only be able to check those loads that coexist in the LSQ with a matching store.

Table 7.3 shows the actions taken when loads hit in the *MOVT* at commit.

(i) Same actions and explanations as in design #1 when a load hits the *MOVT* and has a valid *prodID*. Note that a committing load with a valid *prodID* will have its *match? bit* set.

(ii) As opposed to design #1, in case the *prodID* is NULL, we will not flush the pipeline. Since the *MOVT* is not accessed at execution time, we can not reason about the existence of a failure for this case. Hence, on this situation no recovery action will be taken and it will imply coverage loss.

Similarly, loads may miss the *MOVT* at commit, either because they did not obtain a valid *prodID* from the LSQ, or because there were stores operations that

Table 7.4: Protocol when loads miss the *MOVT* at commit time (Design #2)

prodID	***match? bit***	**Action**
N/A	Set	Fix
N/A	Unset	None

evicted the information required by the committing load. Table 7.4 summarizes the heuristic implemented by the *Error detection logic*:

(iii) Same actions and explanations as in design #1 when a load misses the *MOVT* but has its *match? bit* set.

(iv) Same actions and explanations as in design #1 when a load the *MOVT* and has its *match? bit* unset.

Potential Issues

This minimal *prodID* acquisition implies that only *MOVT* accesses are required at commit time, achieving a low complexity design. As a consequence, this option will overcome the potential problem of timing constraints introduced by processor layouts. However, since only those loads that have been forwarded a value will obtain a *prodID*, this design will pay a higher coverage loss.

7.7 Design #3: *MOVT* Access at Allocate

Motivation

In this section we describe an implementation of the general design proposed in Section 7.4, that performs the prodID *acquisition* during load allocation. As commented previously, the rationale behind is to propose an alternative design for scenarios where it may be hard to accommodate due to timing restrictions accesses to the *MOVT* from the commit and execute stages. This is a challenging fact, since with this design loads must obtain their *prodID* before their address generation has been done (it is performed in the execution stages at the backend of the processor).

Our strategy for overcoming the problem of accessing the *MOVT* at the frontend of the processor, consists on using *address prediction*. We provide loads with speculative *prodIDs* and allow an access to the *MOVT* off the critical path, because it moves the prodID acquisition to the allocate stages, which is usually physically close

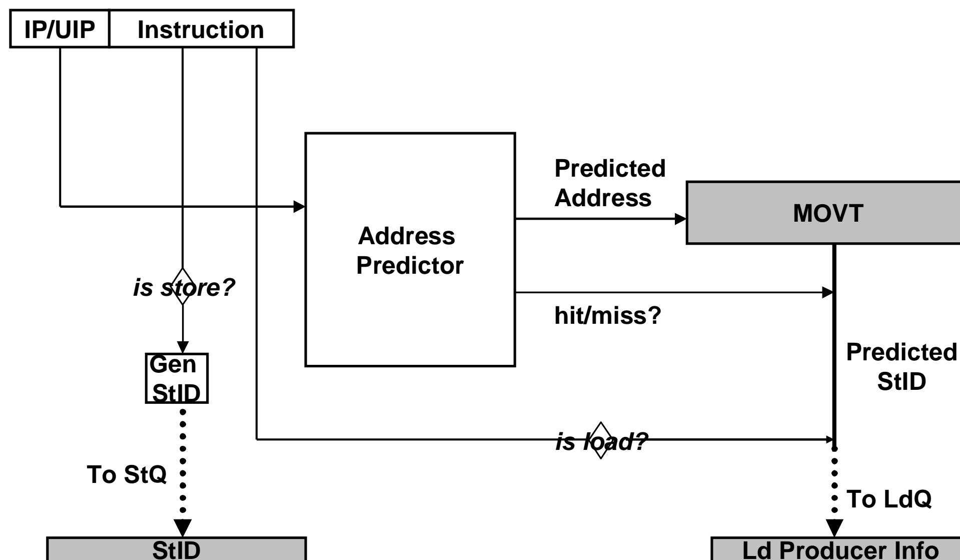


Fig. 7.7: *MOVT* hardware for design #3: prodID *acquisition* at allocate time

to the commit stages. As there are several stages between allocation and execution, this design can tolerate delays when accessing the *MOVT* .

Speculating the memory references of loads and stores has been shown to be very amenable for prediction [101]. In fact, the effective addresses of most memory instructions follow an arithmetic or repetitive progression. Actually, a myriad of effective value predictors have been proposed [168].

ProdID Acquisition

Figure 7.7 shows how the allocate stages are extended by the proposed mechanism. In order to achieve this functionality at the frontend of the processor, our technique provides a predicted *prodID* rather than a real *prodID*. To do so, for each load we build a hash signature using the instruction pointer (program counter). This index is used to access the *address predictor*.

For every predicted load, we use its predicted address to access the *MOVT* table. Upon a hit, a speculative *prodID* is obtained and later allocated to its *Ld Producer Info* field. [4] Those loads not obtaining a speculative *prodID* will set a *NULL prodID* in its corresponding *load queue* entry. Even if a load hits the *MOVT* at the allocate stage, its *prodID* field can be overridden at the execute stage whenever the LSQ logic

[4] Note that the speculative *prodID* accuracy will depend on many parameters, including the type of predictor, its size, its confidence, possible index aliasing, and of course, on the predictability of memory addresses.

Table 7.5: Protocol when loads hit the *MOVT* at commit time (Design #3)

prodID	***match? bit***	**Action**
VALID	N/A	Check
NULL	N/A	None

Table 7.6: Protocol when loads miss the *MOVT* at commit time (Design #3)

prodID	***match? bit***	**Action**
N/A	Set	Fix
N/A	Unset	None

detects a *store-to-load forwarding* scenario. Similarly, the load will set its *match? bit* to be used later for load validation.

Regarding the *address predictor*, we opt to update it at commit time. [5]

Validating Memory Ordering

Each speculative *prodID* is compared to the *StID* stored in the *MOVT*. Note that the real effective address is used to access the *MOVT* at commit, not the predicted one (which is no longer used after the allocate stage). Now, a *StID* mismatch does not necessarily indicate the occurrence of a failure but rather the *possibility*, due to an address misprediction leading to accessing a wrong *StIDs* at allocate time.

Table 7.5 summarizes the actions taken by the *Error detection logic* when a load hits in the *MOVT* at commit time.

(i) If we have a valid *prodID* for the commiting load and have obtained a *StID* from the *MOVT* at commit time, we can compare both *IDs*. If the *match? bit* is set then it cannot correspond to a false positive, because the store's *StID* will override the load speculative *prodID*. Conversely, if it is not set, then a false positive could have happened in case a wrong address was predicted. Upon a mismatch, we perform a corrective action: the pipeline is flushed and re-execution starts from the offending load.

(ii) A load hitting the *MOVT* at commit time but with no valid *prodID* will not always correspond to a failure. It is possible that a load misses the *MOVT* at allocate time. Address misprediction is another reason leading to missing in

[5]Although updating it at the execution stage could potentially achieve faster update-to-use latency, it has the cost of storing the hash index in the *load queue*. Moreover, our studies showed a negligible difference in prediction accuracy.

the *MOVT*. Even if the address is correctly predicted it can happen that the producing store is close enough in the pipeline. In this case, the producing store will not have enough time to update the *MOVT* with its *StID* before the consuming load accesses it.

In this case, the design #3 does not perform any action and ignores this case. The consequence is coverage loss.

Due to conflicts, entries from the *MOVT* can be evicted and loads can miss at commit time. Table 7.6 summarizes the different possible scenarios:

(iii) Same actions and explanations as in previous designs when a load misses the *MOVT* but has its *match? bit* set.

(iv) Same actions and explanations as in previous designs when a load the *MOVT* and has its *match? bit* unset.

Potential Issues

With this design, possible timing constraints are avoided because it moves the *prodID acquisition* to the allocate stages, which are physically close to the commit stages.

On the other hand, since this scheme works with speculative *prodID* it can pay a cost in coverage loss and also in processor performance (due to extra pipeline flushes caused by false positives).

7.8 Evaluation

In this section, we present a detailed evaluation of the three designs presented in Section 7.5, Section 7.6 and Section 7.7. We will evaluate these designs in terms of area, error coverage and performance slowdown (due to pipeline flushes) in the baseline out-of-order processor described in Appendix A.

7.8.1 Fault Coverage Methodology

From a global perspective, our previous studies [205] based on AVF analysis indicate that all the hardware involved in implementing memory dataflow functionalities represents 8.47% of the SDC FIT rate caused by soft errors (excluding protected structures, like caches, TLBs, etc.). Previous studies [212] report similar error rates using fault injection methodologies. Furthermore, 12.73% of the baseline processor

area is exposed to other sources of failures, including wear-out, design and hard faults (again, excluding protected structures). By analyzing the microarchitectural blocks and by means of fault vulnerability studies (as described in Section 4.1.3), we have determined that the proposed technique is able to potentially cover 12.60% of the baseline processor area, and to potentially target 8.39% of the SDC SER FIT.

As opposed to the rest of the chapters where we perform fault injection campaigns to evaluate the actual error detection capabilities of the proposed solutions, here we follow a different methodology. No sampled fault injection is introduced because coverage con be deduced from properties of the producing and consuming memory instructions. We compute it as follows.

We have analytically quantified the error coverage as the number of committing load operations that can be validated completely with our on-line mechanism. For each technique, we classify each committing load based on Tables 7.1–7.2, 7.3–7.4 and 7.5–7.6. Those loads that fall under actions *Check* or *Fix* are counted as protected. For those loads whose action is *None*, we compute an upper bound of the coverage loss. For this situation, we count as not protected those loads whose producer is close enough so that a forwarding would have been possible. We check that there are at most $2 * S$ stores between the load and the producer, where S is the number of entries in the store buffer. The rationale behind is that no *store-to-load forwarding* or *memory ordering violation* scenarios can arise given this producer-consumer distance. Each execution has been allowed to run for 100M instructions, as described in Chapter 4.

7.8.2 Area Overheads

Clearly, the size of the $MOVT$ will determine the degree of coverage that our designs can achieve: bigger tables allow more loads to check their ordering. However, we are faced with the problem of minimizing the area overhead while providing a high coverage.

We have first evaluated the area overhead for different fully-associative $MOVT$ configurations, as shown in Table 7.7. To do so we have extended our area and power models as described in Chapter 4. We have configured the $MOVT$ to have 6 read ports (4 for loads that may commit, 2 for the two loads that may execute), and 1 write port (only 1 non-bogus store may commit since we only have 1 write port in the data cache). Area numbers are relative to those of the LSQ.

The 8-entries fully-associative $MOVT$ has an area overhead over the area of the LSQ of 7.39%. However, the 16-entries fully-associative $MOVT$ incurs in a 21.32% overhead, whereas the 32-entries fully-associative $MOVT$ is almost as big as our

Table 7.7: Area overhead w.r.t. the LSQ, for different *MOVT* sizes. *e* stands for entries, *w* for ways

	32e, 32w	**16e, 16w**	**16e, 8w**	**16e, 4w**	**16e, 2w**	**8e, 8w**
Area overhead	**74.33%**	**21.32%**	14.78%	3.20%	2.34%	7.39%

Table 7.8: Coverage, slowdown and area cost for different *MOVT* configurations that perform *prodID* acquisition at execute time. Results for coverage and slowdown is shown in pairs (μ, σ) across all benchmarks. *e* stands for entries, *w* for ways

	16e, 16w	**16e, 8w**	**16e, 4w**	**16e, 2w**	**8e, 8w**
Coverage	(99.99%, 0.06%)	(99.98%, 0.07%)	(99.91%, 0.20%)	(99.73%, 0.56%)	(98.73%, 3.33%)
Loads flushed	(0.02%, 0.05%)	(0.03%, 0.06%)	(0.09%, 0.14%)	(0.28%, 0.71%)	(0.47%, 0.85%)
Slowdown	(0.06%, 0.13%)	(0.09%, 0.16%)	(0.24%, 0.42%)	(0.89%, 2.52%)	(1.20%, 2.32%)
Area	**21.32%**	14.78%	3.20%	2.34%	7.39%

processor's LSQ. Clearly, we can see in gray that designs as big as a 16-entries fully-associative *MOVT* begin being extremely costly.

In order to further minimize area overheads while having a reasonable capacity, we have then evaluated the area of different *MOVT* designs implemented by means of set-associative caches. Set-associative caches are less complex than fully-associative caches; however, they usually have more conflicts and hence evictions, which in our case turns into higher performance cost due to extra misses. This behavior is particularly radicalized in the face of caches with few sets or strided patterns. We opt to use XOR-based mapping functions [65] to improve the behavior of our set-associative *MOVT* to achieve similar results to those of a fully-associative *MOVT*. Last row of Table 7.8 shows area overheads for different *MOVT* configurations, including fully-associative versions and set-associative versions with XOR-based mapping functions. It can be observed that a *MOVT* holding 16 entries can be implemented with a moderate area overhead when using a 4 or 2-way associative cache. For a 16e, 2w *MOVT* the area overhead is just 2.34% with respect to the LSQ, whereas a 16e, 4w *MOVT* just requires an area overhead of 3.20%.

7.8.3 Evaluation of Design #1: *MOVT* Access at Execute

Coverage Results

We count as not protected the percentage of loads described in case (iv) in Section 7.5; and we compute an upper bound of the coverage loss by tracking the distance between producing stores and consuming loads (see Section 7.8.1).

We have evaluated three different fully associative configurations: *MOVTs* of 4, 8 and 16 entries. On the right axis of Figure 7.8 we show the total coverage (notice

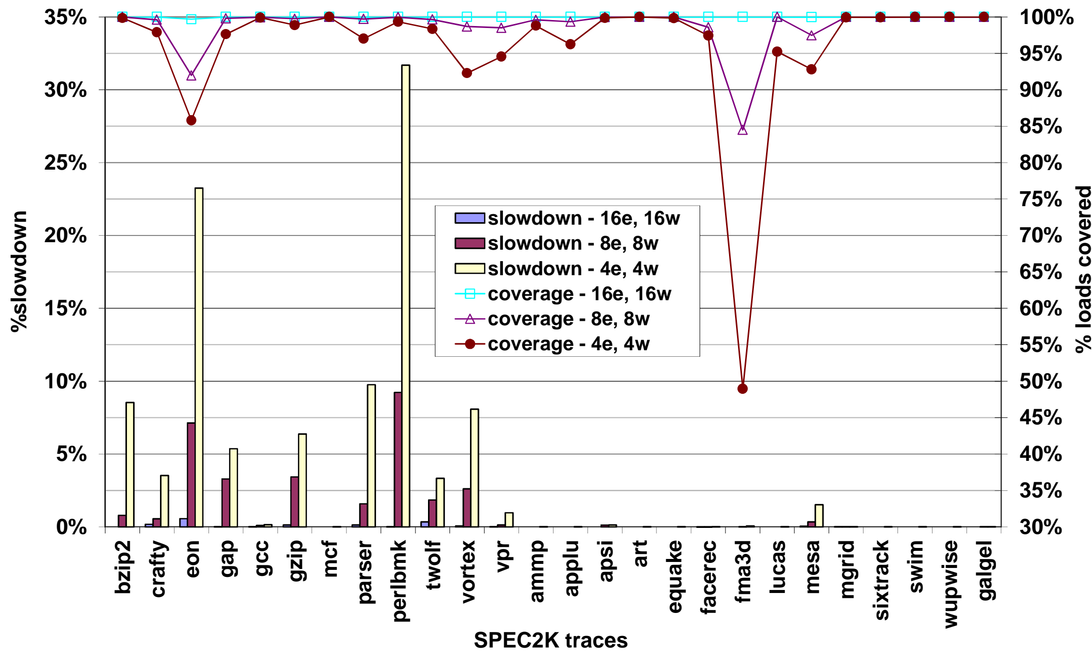


Fig. 7.8: Coverage and slowdown for different fully-associative *MOVT*s based on *prodID* acquisition at execute time

that it starts at 50%). Results show that on average for the SPEC benchmarks, for a 8 fully-associative table, 98.96% of loads would be covered, whereas if we used a 16 fully-associative table, we would cover 99.99% of the loads. However, a 4 entries fully-associative table falls short and only covers 95.79% of the loads on average, with glass-jaw cases like *fma3d*, *mesa* and *eon*. For these benchmarks the producing stores and consuming loads pairs are close enough that they *could* co-exist in the LSQ but the *StID* of the stores are evicted before the consuming loads commit (falling into case (iv) and being computed as coverage loss, as described in Section 7.8.1).

A 16 fully-associative *MOVT* is enough to achieve an excellent coverage. However, its area overhead is huge (recall Table 7.7). In order to understand the implications on coverage of XOR-based mapping functions, we have also conducted several experiments for set-associative *MOVTs*. We summarize the results in Table 7.8. Shadow column shows the best configuration: a 16-entries, 4-way set-associative cache with XOR-mapping, achieves an average coverage around 99.91%, and an area overhead of 3.20%. This configuration achieves better coverage per area than a 16-entries 8-way *MOVT* with XOR-mapping (which needs an area overhead of 15.78% for just 99.97% error coverage - additional 0.06%). Given that this *MOVT* configuration offers the best trade-off in area vs. coverage, we will also evaluate it for the rest of designs.

Detailed evaluation for all benchmarks is depicted in Figure 7.9. Notice that the right axis starts at 97%. It is interesting to note the differences in performance of

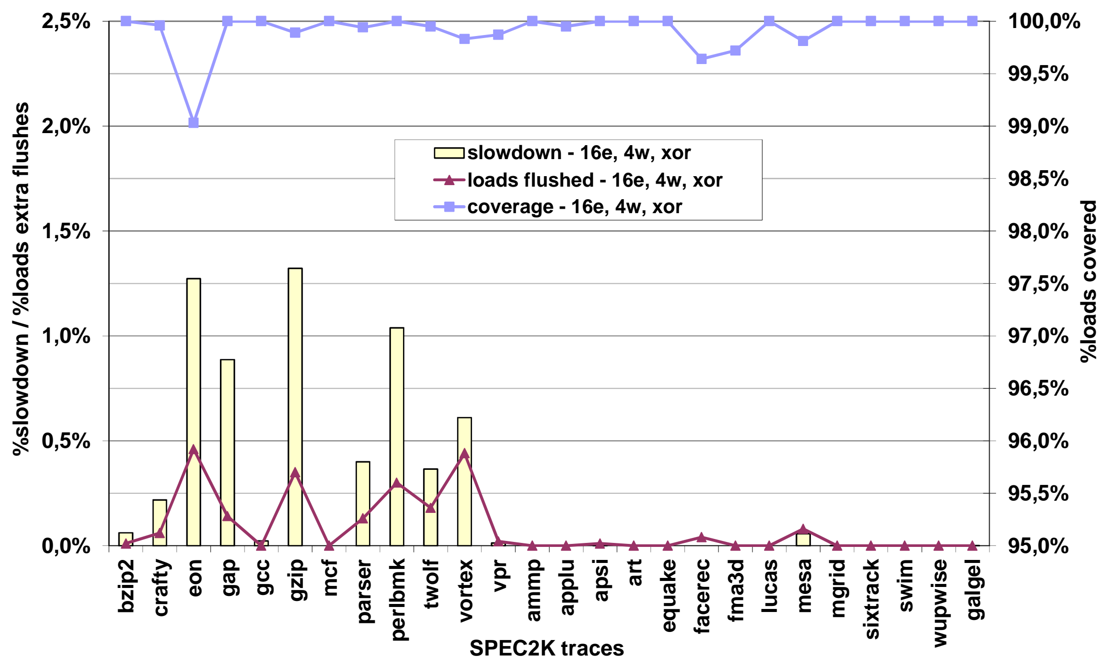


Fig. 7.9: Detailed evaluation of a 16 entries, 4-way *MOVT* based on *prodID* acquisition at execute time

the proposed technique for different benchmarks. The difference in error coverage between benchmarks is basically caused by the distribution in the distance between the producing store and the consuming load. This is common to all three techniques.

Performance Results

As we have explained in Section 7.5, in case (iii) we apply the recovery mechanism conservatively.

Left axis of Figure 7.8 shows the slowdown caused by the proposed design due to the loads that call for a conservative recovery action (for fully associative *MOVT*). Results show that the performance cost is very low. On average, as summarized in Table 7.8, an 8-entries fully-associative table would cause an 1.2% slowdown, whereas performance would drop 0.06% in case of a 16-entries fully-associative *MOVT*. Similar to the coverage, the performance cost incurred by a 4-entries fully-associative *MOVT* is larger: 3.95%, with some programs having over 30% slowdown.

When moving to set-associative and XOR-based mapping functions, a 16-entries 4-way *MOVT* induces just a 0.24% performance overhead (0.52% on average for SPECint and 0.00% for SPECfp). This fact also confirms that XOR-based mapping functions are a good option to reduce the area overhead, while at the same time

providing similar coverage and slowdowns to the ones achieved with fully-associative *MOVTs*. Figure 7.9 details for every SPEC benchmark the performance penalty and the coverage when using a 16-entries 4-way XOR-based *MOVT*. Since performance cost depends much on the number of pipeline flushes, right axis of Figure 7.9 also shows their percentage with respect to the total number of loads. Results show that on average 0.09% of loads are flushed (0.17% for SPECint and 0.01% on average for SPECfp), with few outliers like *eon*, *gzip*, *perlbmk* or *vortex*. It can also be observed that the percentage of flushed loads has a direct (but not exact) correlation with the observed slowdown. The absolute number of loads, the application IPC and other factors also determine the slowdown and the benchmark tolerance to pipeline flushes.

7.8.4 Evaluation of Design #2: Minimal *prodID* Acquisition

Coverage Results

Right axis of Figure 7.10 depicts the coverage achievable with the *minimalist MOVT* design.

If we consider a *MOVT* configuration of 16-entries, 4-way and XOR remapping, the achievable coverage is 91.68% on average. However, for 14 out of 26 benchmarks this scheme is below 95.00%. Specifically, some benchmarks have a rather bad coverage (*eon*, *vpr*, *mesa* and *fma3d* obtain a coverage value of 82.44%, 78.85%, 77.04% and 48.42%, respectively). The reason is the same as for design #1.

However, if we take into account the fact that the *MOVT* is accessed exclusively at commit time, the coverage lower bound suggests that a good part of the coverage can be reaped by exclusively accessing the *MOVT* at commit time.

Similar results are obtained if we use a 32 fully-associative or 16 fully-associative *MOVT*.

Performance Results

For this *minimalist MOVT* design, just one case requires flushing the pipeline in the absence of failures (case (iii)).

Left axis of Figure 7.10 shows the performance slowdown suffered from different configurations. A 32 fully-associative *MOVT* is able to achieve a negligible 0.01% slowdown on average. If we move to a 16 fully-associative *MOVT* slowdown increases slightly: on average, it represents 0.04%. Finally, a 16-entries 4-way *MOVT* renders 0.23% slowdown. The worst cases correspond to benchmarks *eon*, *gzip* and *perlbmk*,

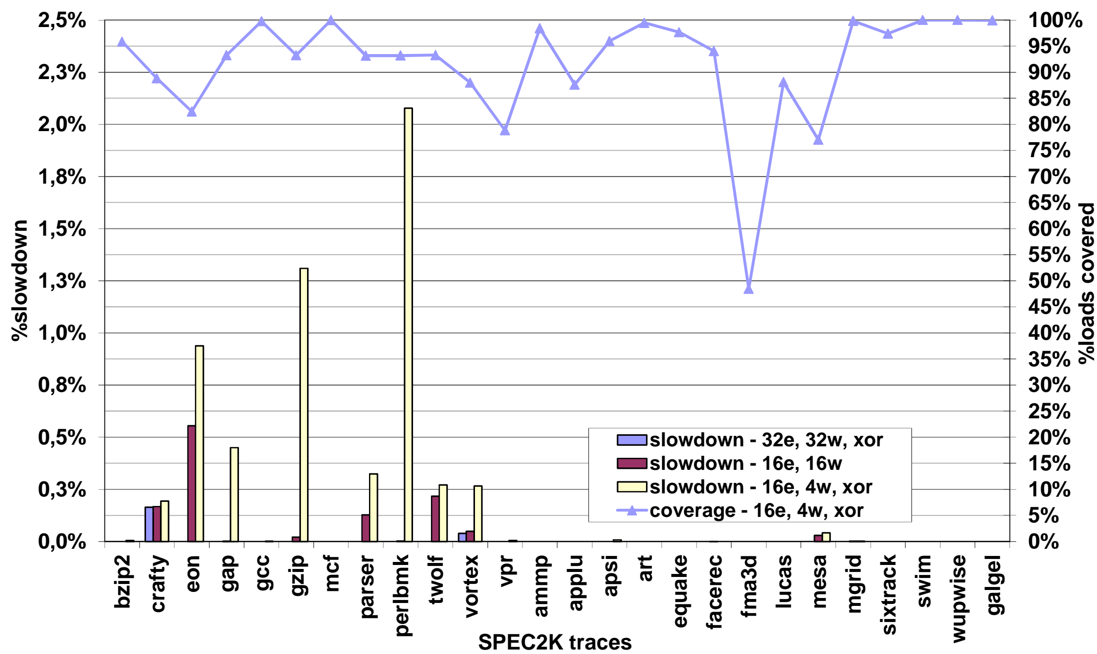


Fig. 7.10: Coverage and slowdown for different *minimalist MOVTs* configurations

which degrade performance in 0.94%, 1.31% and 2.08% respectively (same benchmarks as for design #1). These slowdown results are slightly better than the ones achieved in design #1.

Figure 7.11 exposes the percentage of flushed loads, for different *MOVT* configurations. These results are lower than the ones shown in Section 7.8.3 (design #1). For example, results for a 16 fully-associative *minimalist MOVT* indicate that on average 0.02% of the loads are flushed. For a 16-entries 4-way *MOVT* 0.06% of the loads require a conservative recovery action, whereas design #1 required 0.09% of the loads. The number of flushed loads is a bit lower than for design #1 because no interferences are introduced from the *prodID* acquisition: for design #1, the *prodID* acquisition for loads that are not being forwarded could displace the LRU information of the *StIDs* of those stores that have forwarded the value and are awaiting to be checked.

7.8.5 Evaluation of Design #3: *MOVT* Access at Allocate

Coverage Results

This subsection will evaluate the implications of a speculative *MOVT* scheme on error coverage, and will show whether it is able to achieve a similar coverage to non-speculative designs.

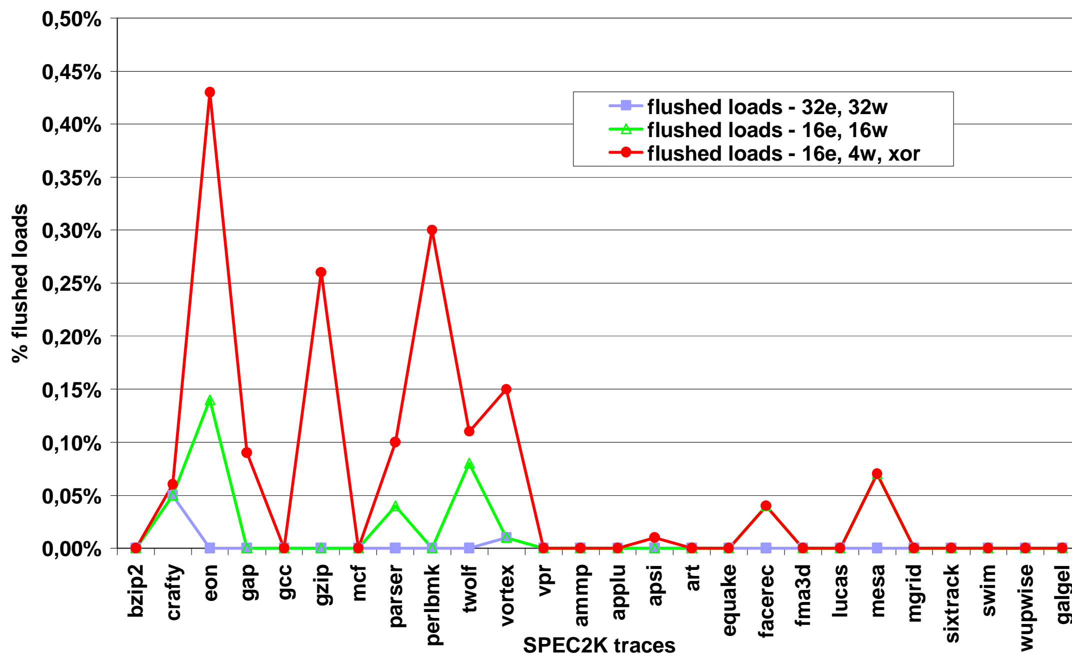


Fig. 7.11: Flushed loads for different *minimalist* *MOVT*s configurations

In order to reduce the effect of address misspredictions on coverage, we have evaluated the technique with a big state-of-the-art value predictor, the *DFCM (Differential Finite Context Method) predictor* [62]. [6]

Clearly, as the predictor introduces big area overheads, this design option is viable in case the predictor is amortized for other purposes, such as for supporting data prefetching [41], or for supporting speculative execution of load and store instructions [66].

We have evaluated the error detection capability for three different *MOVT* configurations (for design #3). Right axis of Figure 7.12 shows that achievable error coverage is much below the coverage provided by design#1 but notably above design #2. When using a 16-entries 4-way *MOVT*, coverage ranges from 84.15% (*eon*) to 100.00% (*lucas*). From the the total set of benchmarks, this *MOVT* configuration ob-

[6] The DFCM predictor is one of the most accurate state-of-art non-hybrid predictors and is able to predict constant, strided and complex memory access patterns. The DFCM is a two-level predictor, just like the FCM. An instruction maps to an entry of the level-1 table, and the entry stores the last value and a hashed history of differences between the recently occurring values (the context). The level-2 table is accessed by means of the context obtained from the level-1 table and contains the next difference, updated by a recurring past history of differences. The prediction is computed by adding the last observed value to the predicted delta. We have configured the DFCM predictor to have 2^{16} entries for the level-1 table and 2^{16} entries for the level-2 table. The length of the history (aka order) has been set to 4, as recommended in [62].

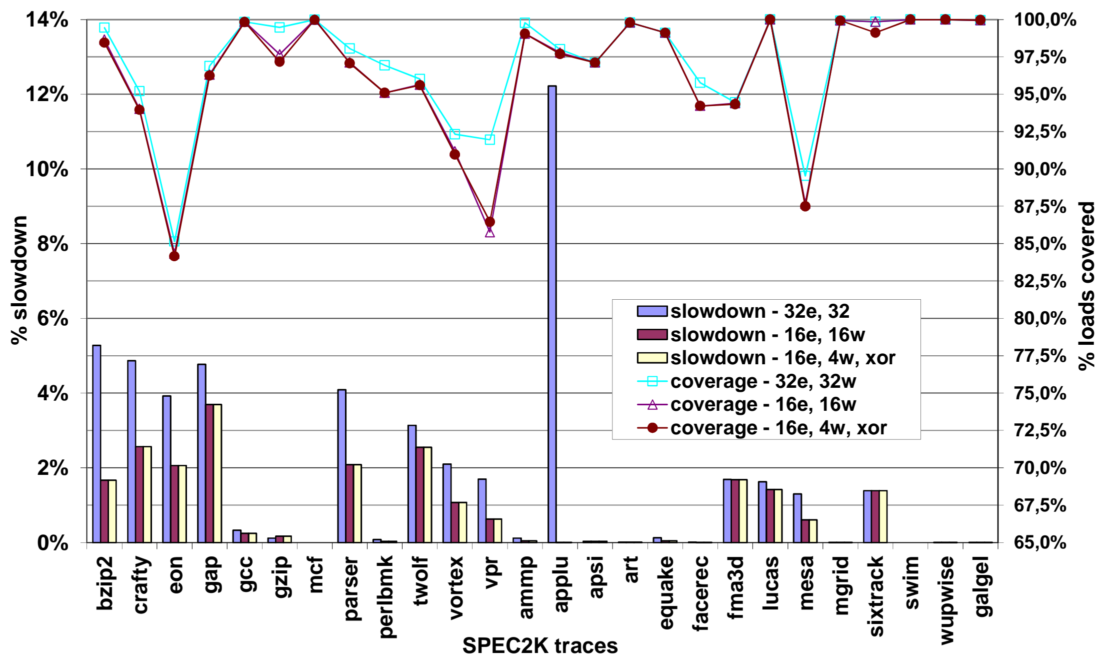


Fig. 7.12: Coverage and slowdown for different *MOVT*s based on *address prediction* (prodID *acquisition* at allocate time)

tains a coverage higher than 95.00% for 19 out of 26 benchmarks. Results show that on average, this configuration can cover 96.26% of the loads against errors. If we use a fully-associative *MOVT* of 16 entries, coverage remains very similar (96.32%). It can also bee seen that if the *MOVT* is configured as a 32 fully-associative table, coverage increases, but is not able to reach a coverage comparable to the one obtained with the scheme that performs *prodID* acquisition at execute time (a 32 entries *MOVT* configuration for design #3 provides 97.10% load coverage on average, whereas design #1 scored 99.99% with a 16 fully-associative *MOVT*). It is worth noting that the coverage of this *MOVT* design suffers from several glass jaws: for 16-entries *MOVT*s, benchmarks like *eon*, *vpr* or *mesa* observe a coverage below 87.5%.

Performance Results

Left axis of Figure 7.12 shows performance results. For a 16-entries 4-way *MOVT* slowdown is quite high: 0.85% on average (1.40% for SPECint and 0.37% for SPECfp). Very similar results are achieved when moving to a fully associative *MOVT* of 16 entries. Furthermore, for both configurations some outliers (5 out of 26 benchmarks) manifest with non-acceptable slowdowns that are above 2%. Comparing with the scheme described in Section 7.5 (design #1), we can see that this design pays a higher overhead (design #1 causes 0.52% and 0.00% slowdown for SPECint and

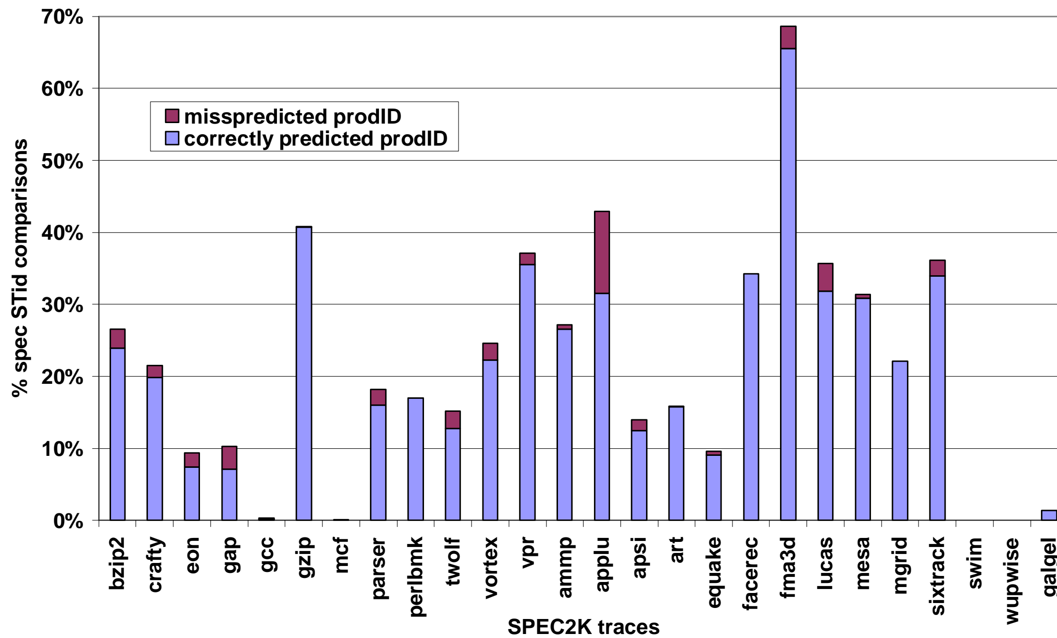


Fig. 7.13: Breakdown of *speculative prodID* comparisons for a 32-entries fully-associative *MOVT* based on *address prediction* (prodID *acquisition* at allocate time)

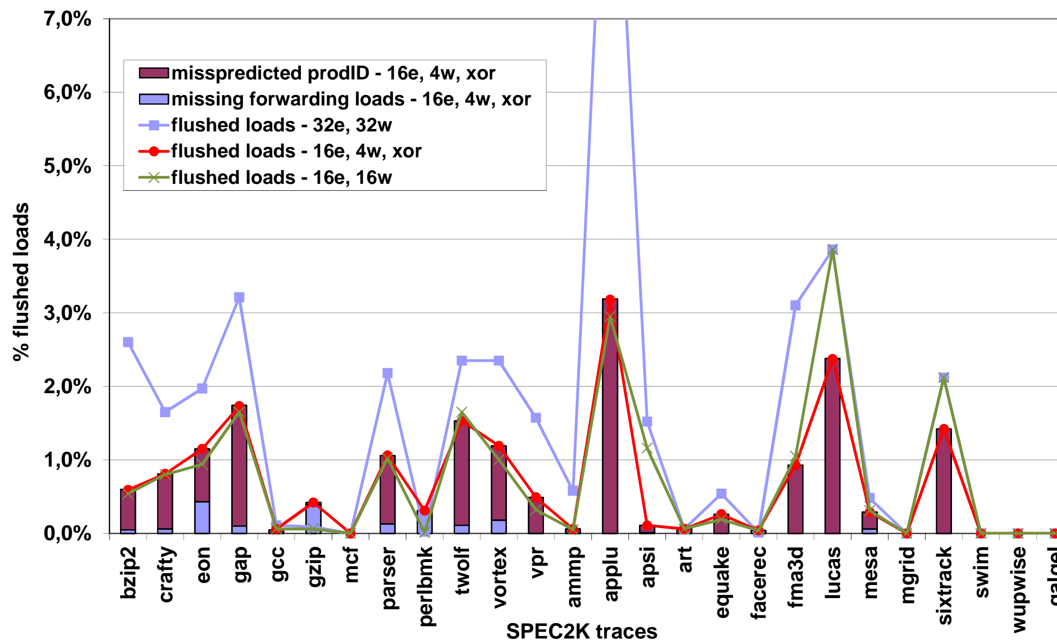


Fig. 7.14: Breakdown of pipeline flushes for different *MOVT*s based on *address prediction* (prodID *acquisition* at allocate time)

SPECfp using a 16 entries 4-way *MOVT*).

However, we observe that as the *MOVT* gets bigger the penalty in performance grows significantly. Specifically, for a 32 fully-associative *MOVT* average slowdown grows to 1.88% (2.53% for SPECint and 1.32% for SPECfp). In order to understand the impact of *prodID prediction* on the processor performance degradation, Figure 7.13 classifies for a 32 fully-associative *MOVT* the percentage of loads that hit at commit time, that do not obtain their *prodID* by means of *store-to-load forwarding* and that have a valid *prodID* (not NULL). Despite many loads are able to obtain a correct *prodID*, there are still many that fail to correctly validate it. A 32 fully-associative *MOVT* will flush 1.60% of all loads, whereas a 16 fully-associative *MOVT* this would decrease to a 0.74%, as shown in Figure 7.14. Note that these percentages do not include the forwarded loads missing the *MOVT* at commit time.

The lines in Figure 7.14 depict the total number of flushed loads for three different *MOVT* configurations. The most important aspect here is that the number of loads that obtain their *prodID* by means of *store-to-load forwarding* and miss the *MOVT* at commit time are negligible with respect the number of loads that have a misspredicted *prodID* and hit at commit time. In fact, the number of forwarded loads that are opportunistically flushed for a 16 entries 4-way *MOVT* (for design #3) is very similar to an equivalent *MOVT* under design #1. The percentage is 0.07% on average and is always below 0.43% (*eon*). The *prodID* mispredictions is therefore the dominating cause to performance loss. Benchmarks with difficult *StID* predictability (such as *gap*, *twolf*, *crafty*, *lucas* or *sixtrack*) are the ones suffering a high slowdown, whereas for design #1 and design #2 the worst performing were the ones with a high percentage of forwarded loads that missed the *MOVT* at commit time.

7.9 Conclusions

The LSQ is one of the most complex structures in a modern out-of-order processor. Unfortunately, most of global hardware or software error detection techniques based on re-execution are unable to protect the LSQ logic because they do not replicate the load-store queue activity across threads.

In this chapter we have proposed a light-weight on-line error detection method that targets the runtime validation of the *memory dataflow logic* implemented by the load-store queue. The proposed technique is able to potentially cover 12.60% of the baseline processor area against faults, and to potentially target 8.39% of the SDC SER FIT (excluding protected structures). Our tecnique (*MOVT*) leverages the microarchitectural knowledge of the runtime behavior, and it is independent of the particular LSQ implementation. The technique relies on a small cache-like structure

that keeps track of the last store id to each cached address. Load are checked to have obtained the data from the youngest older producing store. Non-permanent faults can be corrected by flushing and re-executing the faulting instructions. Moreover, this general technique can be implemented at several forms, offering different trade-offs for error coverage, performance overhead and design complexity.

We have presented and evaluated three different implementations, with different design complexities. The designs differ in how a load obtains its producing store id (*prodID* acquisition): the first design obtains them during execution, the second obtains them exclusively from forwarding stores (minimal acquisition), and the third one obtains them at allocate time by means of address prediction.

Overall, it can be observed that a *MOVT* design that performs full *memory ordering tracking* at the execution stage (design #1) is an excellent design point. On average, this design can validate 99.91% of the loads against errors, with an average negligible performance overhead of 0.24%. Area is increased just by 3.2%.

We have also shown that the other two designs are not able to achieve the error detection coverage and the performance overheads of the design performing *prodID* acquisition during execution. The design with minimal *prodID* acquisition requires minimal extra processor complexity because the *MOVT* is accessed exclusively at commit. It achieves an average error coverage of 91.68% with 0.23% performance degradation. In addition, several glass-jaw benchmarks obtain a rather bad coverage, making the design attractive only for designs where fault tolerance is secondary. Finally, we prove for the third design (*prodID* acquisition at allocate time) that address prediction fails at enabling an efficient *MOVT* design: high misprediction rates sink processor performance while at the same time coverage is below than the achieved for the first design.

We conclude that the design doing *prodID* acquisition at execution time (design #1) configured as a tiny 16 entries 4-way *MOVT* is the option that provides the best coverage per area and the best coverage per performance overhead, while at the same time being extremely amenable for implementation due to its minimal costs.

CHAPTER 8

AUTOMATED FAULT LOCALIZATION AND DIAGNOSIS

8.1 Introduction

In the previous chapters we have presented low-cost solutions to exclusively detect failures during runtime.

However, several facts suggest the possibility of adding extra value to our solutions by extending their applicability to the post-silicon validation phases. The first observation is that since dynamically checking micro-architectural invariants allows detecting multiple sources of failures (including design bugs), the reliance on simulation farms to generate tests and golden outputs could be minimized. Test programs and applications could be directly executed and errors could be detected without needing to perform signal or architectural state comparisons. The second observation is that finding errors by comparing the architectural state against an expected one incurs very high detection latencies. Big latencies lead to time-consuming and complex debugging methods to narrow down relevant execution activity. On the other hand, by dynamically checking micro-architectural invariants errors can be detected as soon as they affect an instruction. The third observation is that minimal interference is required during system validation, and our techniques fulfill this requirement because they run concurrently and transparently with the checked hardware.

However, more problems plague current post-silicon validation practices. Current state acquisition techniques (like scan chains or on-chip tracing buffers) are inefficient when trying to increase the limited internal observability: small buffer capacities and frequent system interruptions to extract internal state require long trial-and-error manual processes and skilled staff. In addition, non-reproducible errors can hardly be

debugged with existing solutions because they cannot continuously trace the internal activity.

In this Chapter we introduce a novel hardware-software solution to locate and diagnose errors during post-silicon validation. We build it by combining the features of our error detection schemes with new logging and diagnosis techniques.

To show the potential of our approach, we have particularly focused on how to apply it to validate a specific functionality of an advanced out-of-order processor: the memory dataflow implemented by the Load-Store Queue. However, the approach could be instantiated to other core functionalities, such as the register dataflow logic or the control flow logic.

The rest of the chapter is organized as follows. Section 8.2 introduces our post-Si and runtime validation technique. Next, Section 8.3, Section 8.4 and Section 8.5 delve into the specific details of the implementation. Afterwards, Section 8.6 evaluates our work in terms of diagnosis coverage, performance, area and power overheads. Previous work is reviewed in Section 8.7. Finally, we summarize the main conclusions in Section 8.8.

8.2 Automated Fault Localization and Diagnosis: Proposed System Overview

Our validation proposal is a hybrid hardware-software system built of several components to achieve automated localization and diagnosis.

We introduce a mechanism that logs the microarchitectural activity for later analysis. This logging mechanism allows continuously storing traces that reflect the system internal activity during program execution, at processor full-speed. We define as an *event* a microarchitectural activity or a change of state related to the circuit under debug. A selected set of *event types* are tracked by the logging mechanism and an aggregate of their possible instantiations constitute the *activity log* used for debugging. With minimal changes in the OS, the log can be stored in one or more pages of the memory space of each application being run. This way, we can store long logs without adding big memory structures in the processor or impacting the performance of applications when stealing large part of the cache. Events generated by the processor are temporally stored in a small hardware buffer while waiting for data cache idle cycles. The data cache acts as a proxy to the rest of the memory hierarchy. This component allows alleviating the problem of reduced internal observability and reproducibility.

Connected to the logging mechanism, we integrate our on-line, concurrent, timely error detection mechanism. Specifically, we choose the *MOVT* mechanism described

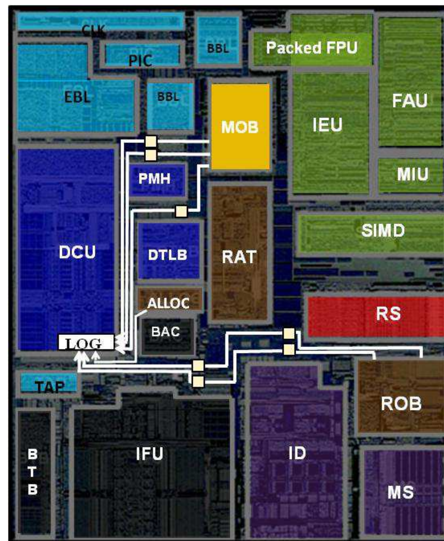


Fig. 8.1: Event driving latches: extensions in the processor

in Chapter 7), because we focus on the debugging of the memory dataflow logic implemented by the Load-Store Queue. The timely feature of our error detection mechanisms allows detecting errors before they cause data corruption, and allows having a precise, unpolluted state in the processor microarchitecture and in the activity logs upon error detection (no events past the error manifestation point are logged, reducing log capacity requirements). The on-line concurrent feature allows detecting failures arising from multiple sources of failures (including design bugs and transient faults) and eliminates the dependence on system-level simulation of RTL models to obtain golden outputs to compare against.

Finally, the last component of our validation method is a software-based diagnosis algorithm. Once an error is detected by the error detection mechanism, this algorithm will examine the log and will root the cause for such error. Validation is performed by analyzing the events stored in the log, and the location and root cause of the error is attempted to be identified in an automatic manner.

Next, we detail how the different components are integrated into the processor and how they interact with each other.

8.3 Event Generation

The first necessary modification consists in forwarding the microarchitectural LSQ activity to the core-level logging mechanism, and collecting it. Every memory operation has associated several types of activity events that may be generated out of program order. The activity log is built incrementally by aggregating the events that

occur within the same cycle. This way, the log reflects the activity introduced in the pipeline in a *timely manner*, not necessarily in program order.

Figure 8.1 shows the layout locations where events are generated. Based on the baseline processor microarchitecture described in Appendix A, we have defined 4 types of events related to the activity of the LSQ that we will use in order to root a fault. Each event carries some important piece of information that is used later by the software diagnosis:

1. `ALLOC` **event**: we generate this event when a load is allocated. The information associated to this event consists of the memory size it reads from memory (3 bits), its position within the *Load Queue* (5 bits), and the *Store Queue* head and tail pointer values upon its allocation (both 5 bits).

2. `COMMIT` **event**: we generate it when a store commits. This event contains the store position within the *Store Queue* (5 bits), and a bogusness bit indicating whether the store belongs to a wrong control path or not (1 bit).

3. `AGEN` **event**: we generate an `AGEN` event for every executed store. This event contains: the store position within the *Store Queue* (5 bits), its effective size (3 bits), its linear address (32 bits) and two extra fields indicating whether the store detected a load introducing a *memory order violation* (1 bit) and the corresponding load position within the *Load Queue* (5 bits).

4. `LDEXEC` **event**: for every executed load, we generate a `LDEXEC` event that indicates the load's *Load Queue* position (5 bits), its linear address (32 bits), the read port used to move the load out of the *Load Queue* (1 bit), and two extra fields to tell whether there was a *store-forwarding situation* (1 bit) and the corresponding forwarded *Store Queue* position (5 bits).

Note that there is no `ALLOC` event for stores and no `COMMIT` events for loads. Store allocation information is implicitly included in loads' `ALLOC` events. Load commit information is exclusively needed by our error detection solution (Chapter 7) to identify the load that observed a failure. Since this information is available at the error detection mechanism, there is no need to continuously log it.

Every event type and its associated information are generated in one pipeline stage and one microarchitectural structure. This means that for the case of LSQ diagnosis, there is no need to gather information from other parts of the core. As a consequence, events are generated locally but stored on a centralized structure, called the '*LOG buffer*'. However, due to layout constraints it may happen that the delay required to move events to the hardware log may vary depending on the event type

(pipeline location). In order to solve this issue, we add latches so that every event type generated during the same clock cycle arrives to the log at the same time. The number of latches to be inserted per event type is determined by the worst delay. Nevertheless, given that it is not necessary to log events on the very same cycle they are generated, inserting latches does not pose any problem to the operating frequency.

It is worth noting that the entries that constitute the log are not meant for specific event types. This means that any event can be written to any position within the log. In order to distinguish among event types, we add decode information to every generated event. This adds 3 bits per event: 2 for the event type and 1 to indicate if it is valid or not.

Hence, an `ALLOC` event requires 21 bits, a `COMMIT` event 9 bits, an `AGEN` event 49 bits and a `LDEXEC` event 47 bits.

Next, we describe two possible optimizations in the design space of event generation: an *event fusing optimization* and an *address hashing optimization*. Whereas the first one reduces the number of generated events by merging some events of the same type, the second one reduces the size of certain event types in order to accommodate more events in the log. Note that both optimizations are incompatible (applying one optimization would not allow applying the other one): the first one increases the size of the fused events, whereas the latter reduces the sizes.

Event Fusing Optimization

The quantity of information required per event type is different. It is clear that `ALLOC` and `COMMIT` events require fewer bits than `AGEN` and `LDEXEC` events, because the size of the addresses dominates over the rest. This means that small events will have spare bits in the log entries.

We make use of this situation and propose to fuse consecutive `ALLOC` events and fuse consecutive `COMMIT` events (not `ALLOC` and `COMMIT` events together). This allows us reducing the number of events to be written to the log per cycle.

`ALLOC` events are fused by storing the number of loads allocated in the same cycle. Our baseline microarchitecture can allocate a maximum of 4 loads in the same cycle (as described in Chapter 4), so 2 extra bits are needed. On top of this, it necessary to add their corresponding sizes (12 bits), the first load *Load Queue* position (5 bits), the *Store Queue* head pointer and the tail *Store Queue* pointer values observed during their allocation (20 and 20 bits, respectively). This optimization makes an `ALLOC` event 62 bits long.

Our architecture only allows one non-bogus store commit per cycle. However, we

compress `COMMIT` events for bogus stores into a single event. To do so, we indicate the number of bogus stores retired (2 bits, because at most 4 instructions can be retired per cycle), the initial store's *Store Queue* position (5 bits) along with the bogusness bit set to true (1 bit). As a consequence several `COMMIT` events can be fused to 11 bits.

After applying this optimization, every event stored in the log will require around 64 bits of space (the maximum event size across all event types). Moreover, a maximum of 6 events can be generated per cycle in our baseline processor: 1 `ALLOC`, 1 `AGEN`, 2 `LDEXEC` and 2 `COMMIT` events (whereas if no optimization is applied, 11 events can be generated per cycle in the worst case).

Address Hashing Optimization

A second possible optimization, consists in reducing the size of the larger events so that more event entries can fit in a given area budget. To do so, we compress addresses of `AGEN` and `LDEXEC` events. In this case, a full 32 bit address would be reduced to a smaller number of bits by means of *address hashing*. Depending on the selected hash size, we may have `AGEN` events ranging from 18 to 48 bits and `LDEXEC` events ranging from 16 to 46 bits (for an interval between 1-bit and 31-bit addresses hashes, respectively).

Clearly, this design alternative allows minimizing the required size of an event in the log, at the expense of some loss in the diagnosis coverage.

When using a reasonable hash size (like 8-bit), every event in the log will require 32 bits of space. We will later show that we discard this option for coverage reasons.

8.4 Diagnosis Algorithm

The logging of the processor activity is done in parallel to processor operation. When an error is detected a failure is flagged and we insert into the log the information of the committing load that observed a failure. For the case of LSQ diagnosis, the position of the load in the *Load Queue* suffices.

Different levels of precision may be implemented by the diagnosis algorithm, depending on the amount of information that designers want to obtain as feedback. We have identified two possible diagnosis levels. For example, the diagnosis algorithm may signal a failure case where "*a load at LDQ position 2 with address 0x82ba1700 has been nullified by an older store with address 0x92ba1700*" or it may even extend it with the information that "*the load should actually have been forwarded from*

the store at STQ position 8 with address 0x82ba1700". Clearly, the second output provides much more valuable information for debuggers because besides determining the failure that actually happened during the processor operation, it also allows to determine the expected behavior. However, it is clear that as we increase the diagnosis precision, the bigger will be the number of events to be analyzed. The log subsequence used to conduct the diagnosis is called the *analysis window*.

In order to conduct a theoretical coverage study, we consider several common failure scenarios in the LSQ logic. These bugs include the failure scenarios described in Chapter 7, that mimic bugs found during the validation phases of modern processors. The 19 failure scenarios are described in Table 8.1. The first column corresponds to the failure name, the second column describes the failure scenario and the third column indicates the size of the analysis window required to identify the actual failure scenario. Specifically, two different window sizes are required to diagnose the considered failure scenarios. The first group of failures can be diagnosed by considering an analysis window starting at the failing load `ALLOC` event (and ending in the last logged event for the load observing the failure). The second group of failures can only be diagnosed when increasing the analysis window up to the farthest `AGEN` event belonging to an older store and whom the processor did not `COMMIT` before the failing load `LDEXEC` time. This analysis window is bigger than the previous mentioned one, and is the one that allows determining the expected failure-free case.

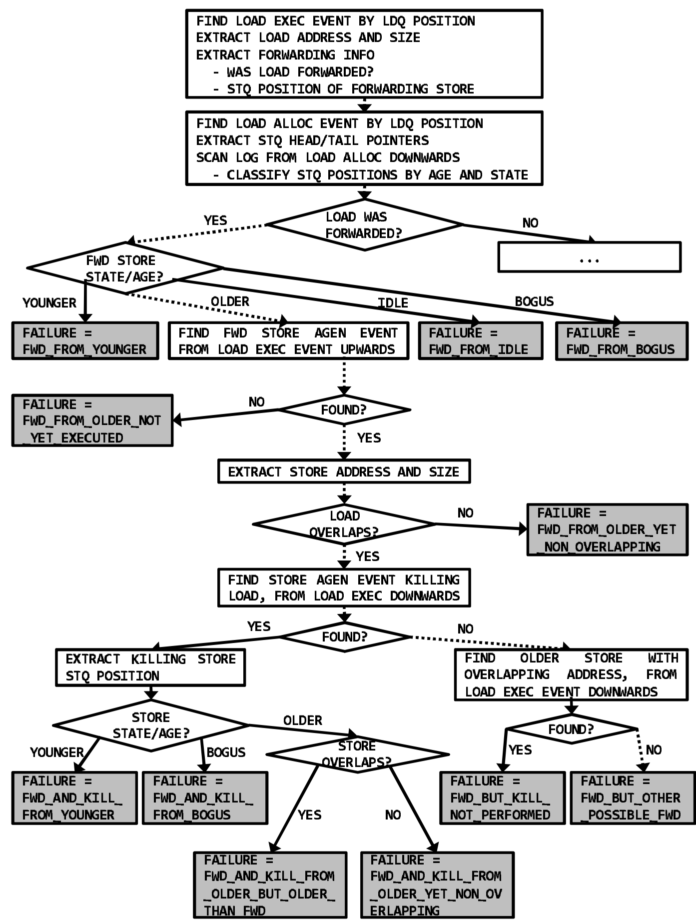
We implement a localization and diagnosis algorithm based on classifying failures depending on a *decision tree*. As nodes are visited (groups of failure scenarios), the failure scenarios are refined depending on the outcomes of different tests. Figure 8.2(a) and Figure 8.2(b) depict at a very high level the implemented diagnosis algorithm for the failure scenarios described in Table 8.1. Note that this code snippet does not provide the expected error-free LSQ behavior (only the faulty one).

The algorithm is constructed in such a way that the first failure types to be considered are those who require the smallest analysis window. Later, if these failure types do not correspond to actual failure case, the rest of failure types are considered (increasing the analysis window). Hence, the number of events to examine in order to identify what went wrong is not fixed a priori. It depends on the actual failure case, and the degree of diagnosis precision desired by debuggers.

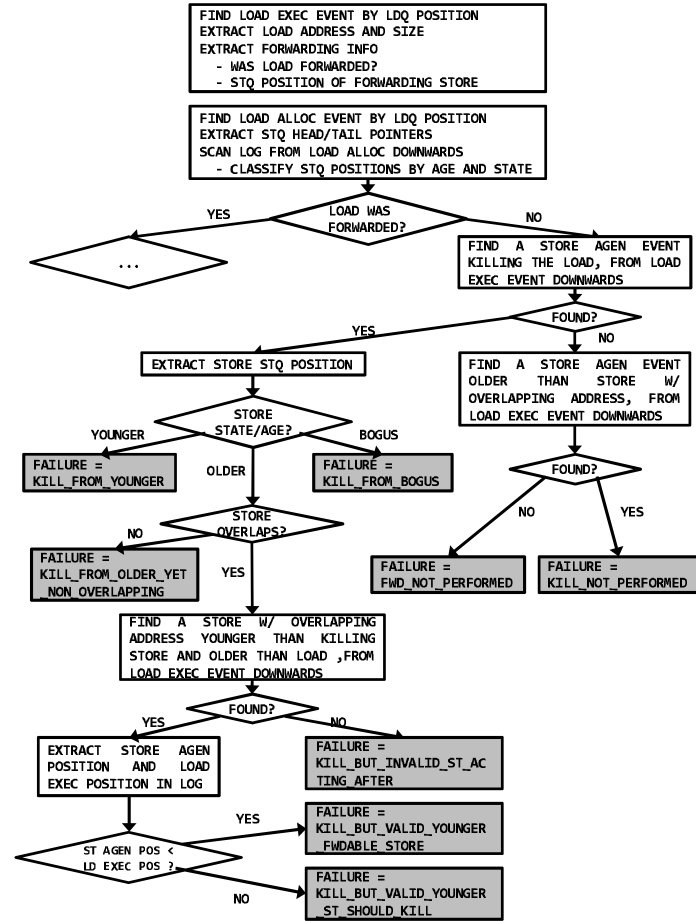
Figure 8.3 shows an example of a short log capturing a failure in the LSQ operation. To clarify things, events marked as `---` are not captured in the log but have been added to clarify the temporal evolution of the microarchitectural activity. The diagnosis algorithm will determine that the failing load is the one in slot 7 in the *Load Queue* (load 7). This information is provided by the error detection mechanism.

Table 8.1: Diagnosable LSQ failure scenarios: descriptions and required analysis window size

Failure Scenario	**Description**	**Analysis Window**
FWD_FROM_YOUNGER	Load was forwarded from a younger store	From ALLOC
FWD_FROM_IDLE	Load was forwarded from an idle STQ position	From ALLOC
FWD_FROM_BOGUS	Load was forwarded from a bogus store	From ALLOC
FWD_FROM_OLDER_NOT_YET_ EXECUTED	Load was forwarded from a previous store that did not compute its address	From farthest AGEN event
FWD_FROM_OLDER_YET_NON_ OVERLAPPING	Load was forwarded from an older non matching store	From farthest AGEN event
FWD_AND_KILL_FROM_YOUNGER	Load was forwarded but there was a younger store that wrongly killed the load	From ALLOC
FWD_AND_KILL_FROM_BOGUS	Load was forwarded but then was killed by a bogus store	From ALLOC
FWD_AND_KILL_FROM_OLDER_ BUT_OLDER_THAN_FWD	Load was forwarded from an older matching store but there was a matching store older than the forwarding one which wrongly killed the load	From ALLOC
FWD_AND_KILL_FROM_OLDER_ YET_NON_OVERLAPPING	Load was forwarded but a previous non matching store later killed the load	From ALLOC
FWD_BUT_KILL_NOT_PERFORMED	Load was forwarded but there was an older store, younger than the forwarding one which should have invalidated the load	From ALLOC
FWD_BUT_OTHER_POSSIBLE_FWD	Load was forwarded from an older store but should have been forwarded from a store older than the load but younger than the wrong forwarding store	From farthest AGEN event
KILL_FROM_YOUNGER	Load was killed by a younger store	From ALLOC
KILL_FROM_BOGUS	Load was killed by a bogus store	From ALLOC
KILL_FROM_OLDER_YET_NON_ OVERLAPPING	Load was killed by an older store but its address did not match	From ALLOC
KILL_BUT_INVALID_ST_ ACTING_AFTER	Load was killed by the correct store, but afterwards another store performed an invalid kill or fwd action	From ALLOC
KILL_BUT_VALID_YOUNGER_ FWDABLE_STORE	Load was killed by an older matching store, but there was an older store younger than the killing one that should have performed a forwarding	From farthest AGEN event
KILL_BUT_VALID_YOUNGER_ SHOULD_KILL	Load was killed by an older matching store, but an older store younger than the killer one should have killed it	From ALLOC
FWD_NOT_PERFORMED	Load should have been forwarded	From farthest AGEN event
KILL_NOT_PERFORMED	Load should have been killed by an older matching store	From ALLOC



(a) Decision-tree when failing load was forwarded at execution time



b) Decision-tree when failing load was not forwarded at execution time

Fig. 8.2: Diagnosis algorithm showing failure type determination: high-level code

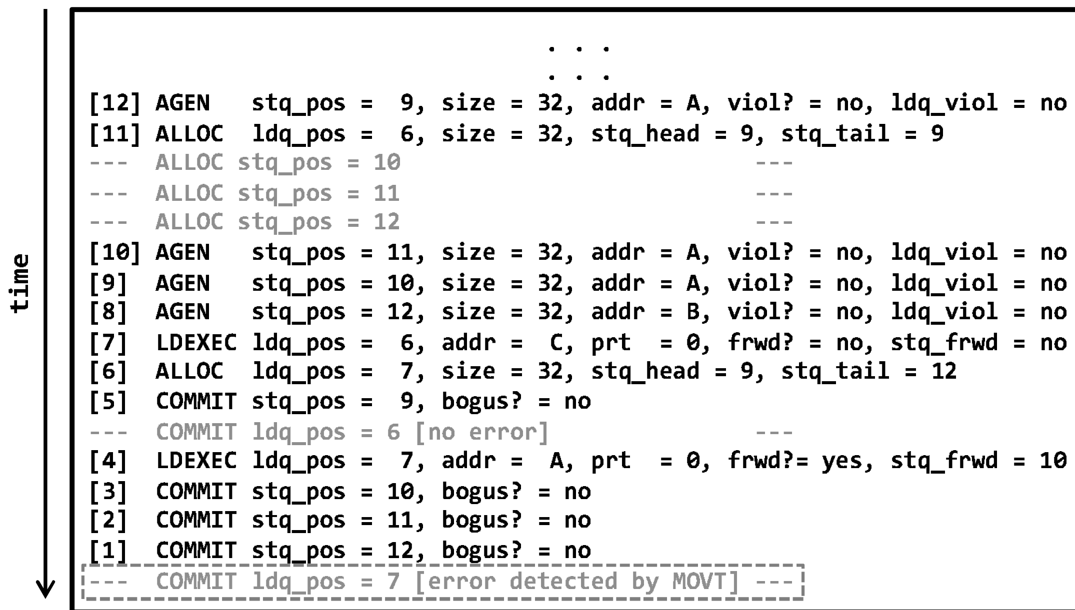


Fig. 8.3: Log of a LSQ failure: an example

The failing load LDEXEC event occupies position [4] in the log, and it indicates that the load has address A and was forwarded by the store in slot 10 of the *Store Queue* (store 10). The failing load ALLOC event occupies position [6] and it indicates that upon its allocation, stores 9/10/11/12 were already in the *Store Queue* and were older.

Scanning the log from this event down to event in position [4], we can refine all store ages and states. Since store 9 was COMMITed in position [5], before the failing load executed, it is now considered as idle (it disappeared from the *Store Queue*). Also, older stores 10/11/12 are determined to be not bogus. Hence, the algorithm follows the edge called OLDER, as shown with dashed lines in Figure 8.2(a). Then, the forwarding store AGEN event is found (analysis window is extended to position [9]) and its address and size are obtained. For this example, store 10 overlaps with the failing load (it has the same address and size) and was executed before. Next, the algorithm finds no AGEN event killing the failing load (from event [4] downwards). Finally, the analysis window is extended to event [10]> to find that there is an AGEN event from store 11, younger than store 10, and it is overlapping. Hence, the diagnosis algorithm concludes that load should have been forwarded by store 11 (FWD_BUT_OTHER_POSSIBLE_FWD failure scenario).

Diagnosis Coverage versus Log Size

The diagnosis algorithm described is able to identify faults for an ideal scenario where the log is unbounded, there is no limit on the number of events that can be logged per cycle and addresses are not compressed. Figure 8.4 shows the average required number of logged events to locate and diagnose a fault for the SPEC benchmarks. In this case, we have used the highest level of diagnosis precision, and have also applied the '*event fusion*' optimization. As one can see, if our log keeps the last 180 events, we are able to root almost all possible faults (99.96%). Note that we are considering the log as unbounded and with no implementation restrictions. In case the log is bounded to a fixed size, a failure will not be diagnosable if the algorithm runs out of events in the log and has not taken any decision. This may happen because of a structural limitation (buffer size, number of writable events per cycle) some events may not be appended to the log, and hence would get lost.

It is important to note that the *address hashing* optimization has implications on the achievable diagnosis coverage. A failure observed for a load will not be able to be diagnosed in case there is *more than one* store whose *address hash* matches the load address hash. Similarly, in case there is *more than one* store whose *full address* matches the load address, then the load will not be diagnosable for any hash size

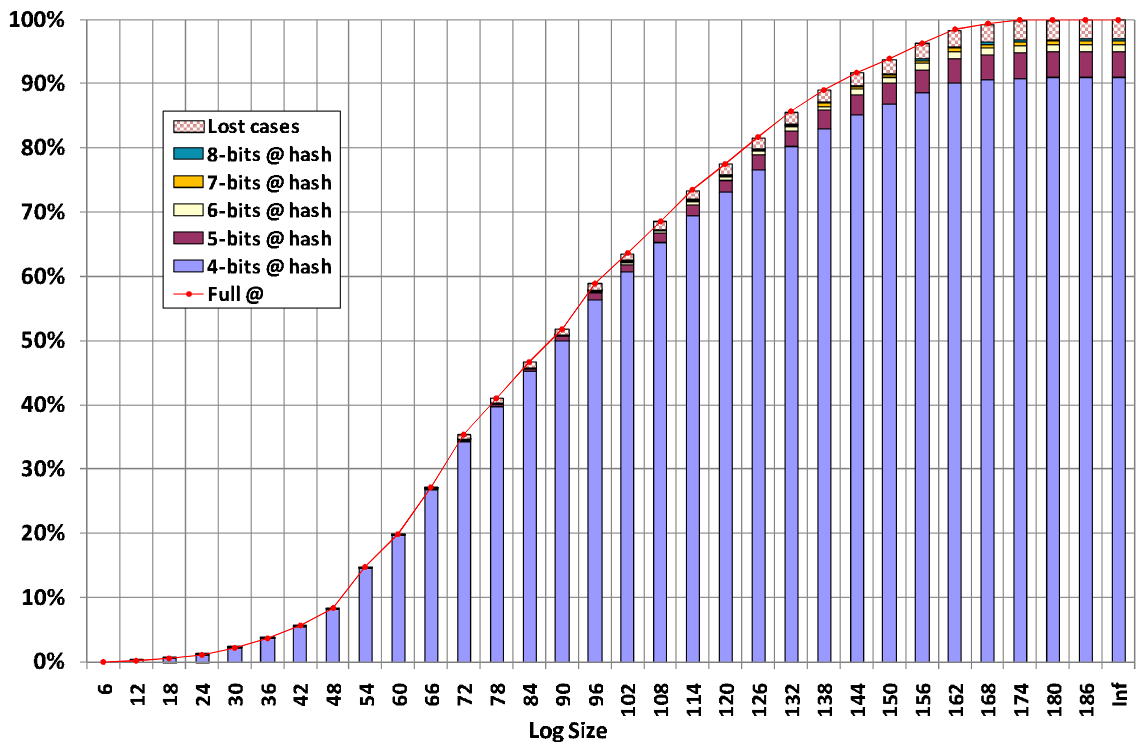


Fig. 8.4: Accumulated diagnosis coverage versus log size

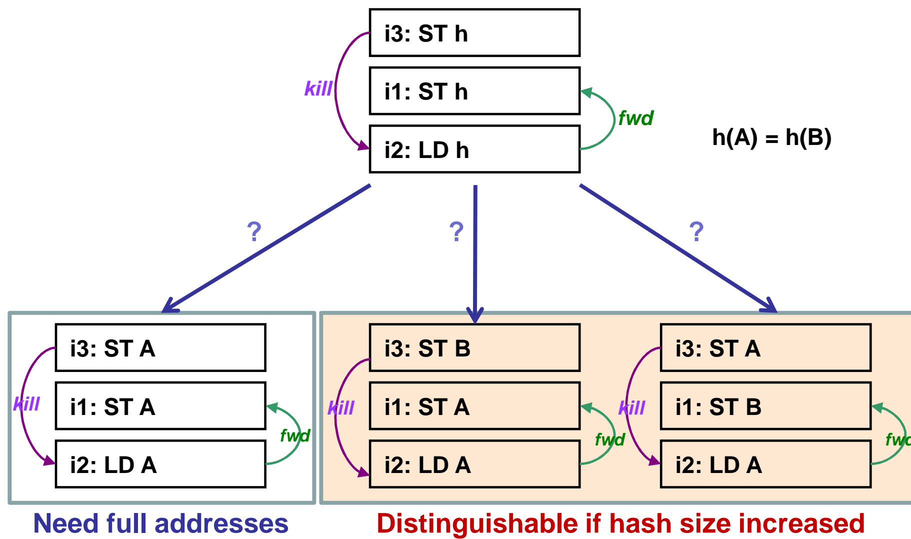


Fig. 8.5: Address hashing undistinguishable failure scenarios: an example

smaller than the length of the address. Whereas in the first case this can be alleviated by increasing the hash size, in the latter case this can only be solved by avoiding *address hashing*. Figure 8.5 shows one of the cases where the diagnosis algorithm could not choose the correct failure scenario among KILL_BUT_VALID_YOUNGER_FWDABLE_STORE, FWD_AND_KILL_FROM_OLDER_YET _NON _OVERLAPPING and FWD_FROM_OLDER_YET_NON_OVERLAPPING. Lines marked as *'fwd'* denote store-to-load forwarding, whereas lines marked as *'kill'* represent memory ordering violation (kill) detection.

Figure 8.4 also depicts the overall diagnosis coverage loss for different hash sizes for our benchmark suite. It is interesting to note that 8-bit hashes addresses shows the best trade-off since they represent the 99.84% diagnosability potential of the *address hashing* optimization. However, the percentage of faults that cannot be diagnosed when using this technique ('*Lost cases*') is 2.91%, which is pretty high for a diagnosis method. Moreover, some specific failure scenarios would never be detected when using the *address hashing* optimization (such as FWD_BUT_OTHER_POSSIBLE_FWD, KILL_BUT_VALID_YOUNGER_FWDABLE_STORE, etc.). Since the '*event fusion*' and the '*address hashing*' optimizations are exclusive, **we opt to use the '*event fusion*' optimization**.

8.5 Logging System Implementation

The upper-bound coverage results from the previous section show that in order to achieve a coverage near to 100%, around 180 hardware log entries would be necessary

for diagnosing errors in the LSQ control logic. Clearly, it is not practical to implement them in a hardware log, because big area overheads would be introduced into the processor. Instead, we propose a new mechanism to minimize the required extra hardware and design effort, while keeping a good diagnosis coverage.

To solve these issues, we propose a more adaptable hybrid software-hardware solution. We modify the OS to sequester one or more physical pages from the application being run (each one being 4KB) to work as a circular buffer for the events. Notice that a page gives enough room to store the required 180 events to fully diagnose a large percentage of faults. Connected to the (first-level) data cache, we introduce a small hardware buffer to temporally keep the events generated by the processor. This buffer sends the events to the main log (in memory) through the data cache, whenever it is idle (otherwise it would be necessary to steal to the running application both cache ports, causing a potential decrease in performance). Hence, the data cache is used as a proxy to the bigger logging storage space. Events generated by the processor will be stored in specific cache lines (cache line events) and will be treated as any regular memory access and be stored on any way, controlled by the cache replacement policy. Moreover, cache line events can be replaced as needed by the application in an adaptive manner and can move through the memory hierarchy. This is not possible in previous scheme that sequester ways or sets: cache line events cannot disappear from the data cache since they are directly dumped out of the data cache upon failure detection.

8.5.1 Microarchitectural Changes

From a hardware perspective, the required changes introduced in the processor are depicted in Figure 8.6 and have been tagged as '*LOGGING system*'.

The inputs to the logging component are two: (*i*) the events that have been generated in their corresponding pipeline stages and (*ii*) a signal indicating whether the data cache ('*D$*') is going to be available (idle) during the current cycle. Next, we will detail the different hardware components that form our logging system.

Merging line: Given that events are offloaded to consecutive positions in memory, we use a special buffer called '*merging line*' which is as big as a cache line (64 bytes). The main purpose of the '*merging line*' is to: (*i*) offload as many events whenever the cache is idle, (*ii*) cluster the events in the minimum number of cache lines and (*iii*) reduce power by using less idle cache write cycles.

The '*merging line*' is dumped to the data cache in bulk mode whenever the '*Dump logic*' determines so. Different decision mechanisms can be implemented to decide when to dump the '*merging line*'; one extreme option is to dump it only when it

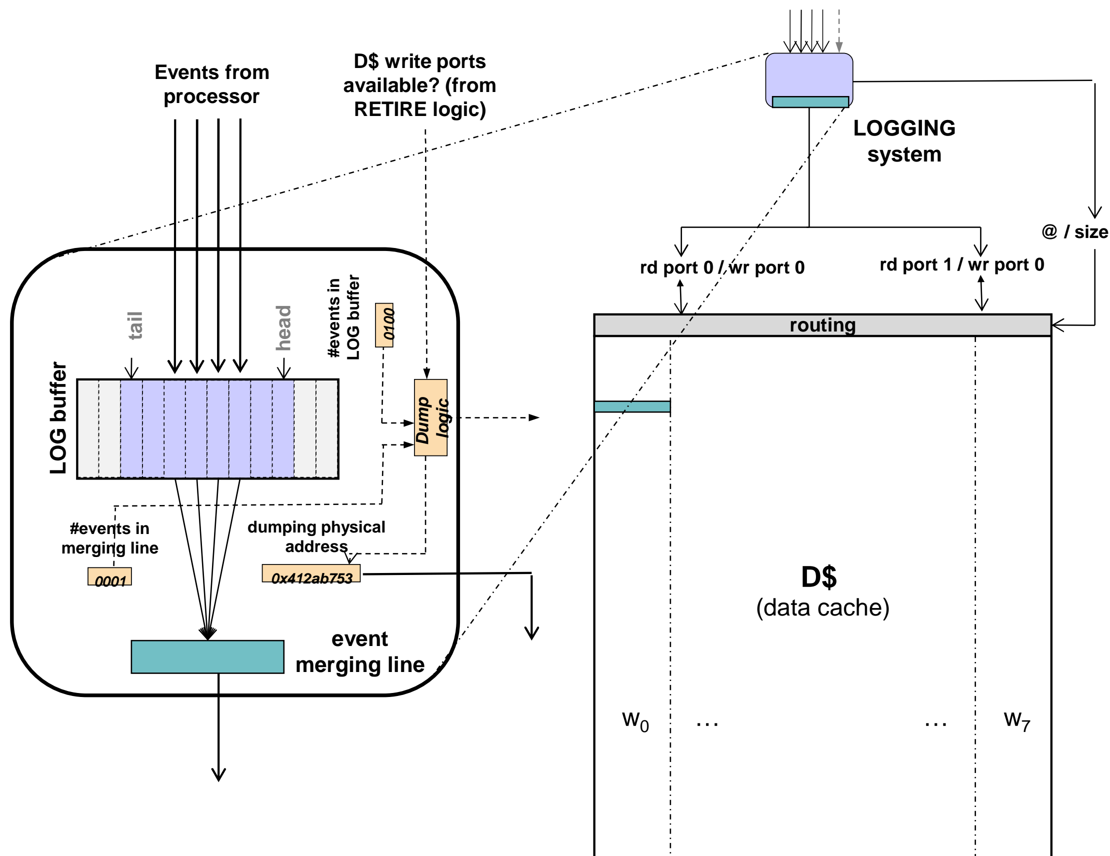


Fig. 8.6: Activity logging mechanism: hardware design and integration

is full. Another extreme option is dumping it whenever the cache is idle. For our experiments, we dump the '*merging line*' whenever the cache is idle and there is at least one event in the '*merging line*'. Note that when using the 64-bit events, the '*merging line*' is able to store 8 events.

LOG buffer: The '*LOG buffer*' stores the events generated by the processor and is the interface with the rest of the processor. The '*LOG buffer*' is designed in such a way that every cycle it can store a fixed number of generated events, unless it is full.

Events from the '*LOG buffer*' are moved out to the '*merging line*' each cycle if enough space is available.

As Figure 8.7 shows, the '*LOG buffer*' is organized as a two-banked structure (instead of a multi-ported structure). Events generated by the processor in the same cycle are stored together in an '*event row*' (a group of as many latches as the number of writable events per cycle). Every '*bank*' is organized as multiple chains of '*event rows*', and every cycle the events inside an '*event row*' advance and are latched into the next '*event row*', if the destination is idle and the source has some events.

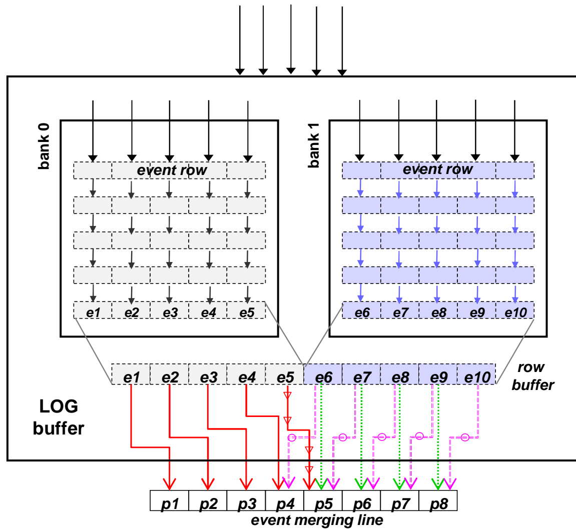


Fig. 8.7: Log buffer: hardware organization

Events from the same cycle are written into the banks in a rotative manner. As a consequence, '*event rows*' in the same positions in different banks must advance simultaneously, so that time ordering across banks is maintained.

In order to increase the read bandwidth when moving events from the '*LOG buffer*' into the '*event merging line*', two '*event rows*' are read every cycle out from the two banks into a buffer called '*row buffer*' whenever it is empty. The rationale for using two banks and reading two '*event rows*' is to exploit the common case where four or less events are generated per cycle, being able to fit into the '*event merging line*' two '*event rows*' per cycle. Note that a multi-ported configuration would allow reading events from as many different cycles as the number of read ports. Some empty events may be present in the '*row buffer*' and in the '*event merging line*', so we mark them at write time so that the diagnosis algorithm may identify them.

The events in the '*row buffer*' advance and are finally moved into the '*event merging line*', once the 'merging line' has been dumped to the memory hierarchy (i.e. it is empty). A clean '*event merging line*' allows associating fixed positions among the events in the buffer and the positions in the '*event merging line*', avoiding full shuffling trees and wide multiplexors. On the other hand, this restricts using the full capacity of the '*event merging line*' and can introduce cycles where the dumping cannot be performed.

Given that the number of non-empty events in the buffer may surpass the capacity of the '*event merging line*', it may be necessary to dump the events in a two-step manner (in two different cycles). However, the '*LOG buffer*' is designed in such a way that a single-step dump can be performed for the common case where 4 or less

events are generated per cycle (a total of 8 events, the size of the '*event merging line*'). The '*dumping control logic*' manages this and the rest of situations, as the pseudo-code in Figure 8.7 shows. This average-case design requires fewer event multiplexors than the full shuffling tree. Please note that even though the 'Dumping Control Logic' described in Figure 8.7 is the particular implementation for the case where 'event rows' hold 5 events each, it can be generalized to any configuration with bigger 'event rows'.

Due to bandwidth reasons, it may happen that the '*merging line*' can not accommodate events from the '*LOG buffer*'. In case the cache is busy for several cycles, the '*LOG buffer*' may also end up being full ('*#events in LOG buffer*' saturates). As a consequence, some events may not be logged. Also, it may happen that during a cycle more events are generated than the number of events that potentially can be written. To address these issues we propose that whenever an event cannot be added to the log, the next successfully written event will be extended with a '*barrier*' bit. From the diagnosis perspective this means that *in case a* 'barrier*' bit is found by the algorithm, the failure is not diagnosable.*

We will analyze the trade off between hardware complexity and diagnosis coverage in Section 8.6.

Physical memory addressing: The logging system also contains a pointer to the physical memory position (aligned to a cache line boundary) where the '*merging line*' will be offloaded. The '*dumping physical address*' is incremented after the '*merging line*' has been moved to the data cache, and the increment is triggered by the '*Dump logic*'.

The allocated physical page (or pages) is used as a circular buffer. The hardware logging system knows when the '*dumping physical address*' is going to point to a physical location past the allocated physical page boundary, and hence also has head and tail pointers to physical addresses.

An interesting aspect of our mechanism is that given that the dumping address is a physical address, there is no need to perform a TLB translation. This eludes the cost of introducing complex design changes in order to deal with TLB misses that are not caused by the application itself.

8.5.2 System-Level Interaction

From a software perspective, few changes must be introduced in our logging system. We opt to have the OS responsible of sequestering the physical page and providing the '*dump physical address*' to the hardware logging component.

Whenever the OS is going to create a new process it obtains from the free pages pool as many consecutive physical pages as required by the structure to be diagnosed, and then assigns it to the logging system. For diagnosing failures in the LSQ control logic, one physical page suffices to achieve a good diagnosis coverage (worst case would be 3 4KB physical pages when storing a single event per cache line). In order to simplify the hardware design these physical consecutive pages are pinned by the OS (cannot be swapped). Once the pages are allocated, the OS must communicate the physical address to the hardware component. The easiest option to implement so is considering the '*dumping physical address register*' as a memory mapped register and accessing it by regular I/O (IN and OUT) instructions.

Application switching: Whenever a new task is going to be switched in by scheduler, the OS will update the '*dumping physical address*' register value. When a task is switched out, the OS reads the corresponding physical address pointer and stores it in the process OS structure for next use.

Clearly, the logging physical pages will be invisible to any process and their addresses will not be stored in any translation table (hence, not accessible). Only the OS will know about their existence.

From a system-level perspective, once a failure is detected the following steps are taken:

1. Pending events residing in the '*LOG buffer*' and the '*event merging line*' are drained off to the physical page, through the data cache.

2. The information gathered by the error detection mechanism (the *load queue* position of the load that raises the error), the log's head and tail physical address pointers are then dumped to the logging physical page. This information is stored in the first 64 bytes of the logging physical page (hence, the logging of events would start on the second cache line boundary of that page).

3. A `MACHINE CHECK` exception is thrown. In order to do so, we rely on existing features to report hardware errors [45]. The processor modifies the respective control and status registers from the corresponding error-reporting register banks in order to indicate that non-diagnosed LSQ error has been reported.

Once the exception has been thrown, the OS takes the final steps.

4. The OS exception routine will identify that an error in the LSQ operation has been detected and then would dump the logging physical pages to a file dump for later analysis in a fault-free processor/core.

It is important to note that our scheme allows having a log per process and allows continuing building a log across different context switches. Furthermore, this logging scheme is not restricted only for diagnosis errors in the LSQ structure. In fact it could be adapted for diagnosing errors for other processors components or logic.

8.6 Evaluation

This section evaluates our technique in terms of diagnosis coverage, area, power and performance overheads.

8.6.1 Diagnosis Coverage Results

Diagnosis coverage has been performed by means of error injection, as described in Chapter 4. For every failure scenario and SPEC benchmark we have simulated the injection of 1000 effective faults. Each fault has been randomly injected during the ten first million executed instructions, after the warm-up period (see Chapter 4). We have allowed faults to propagate, manifest and being catched by the *MOVT* error detection mechanism (during a maximum period of 100M instructions). Once every failure has been detected, we have frozen the simulation and have run the diagnosis algorithm to determine the diagnosis coverage. To do so, we have considered the highest diagnosis precision level (the one that allows diagnosing the actual failure scenario and provide the expected failure-free one).

Using the highest level of diagnosis precision allows diagnosing any failure scenario because we are using the biggest possible analysis window. The same fault injections have been performed across the different considered configurations, so that we can compare in a fairly way.

First, we have evaluated the number of generated events per cycle. Figure 8.8 shows this statistic for each benchmark. As it can be observed, 3 or less events are generated for 97.29% of the cycles, on average. Despite the maximum number of events generated per cycle is 6, this situation happens seldom. Our analysis shows that allowing 5 writable events accounts for 99.96% of the cycles.

We have also run a sensitivity analysis for 27 different '*LOG buffer*' configurations. These configurations are organized in 9 different groups. The 9 groups vary in the number of maximum writable events per cycle and the total number of '*event rows*', but all of them having two logical banks. Results are summarized in Figure 8.9; each configuration `Xwr,Yrows` stands for number of writable events per cycle (size of an '*event row*'), and *total* number of '*event rows*', respectively. As an example, the

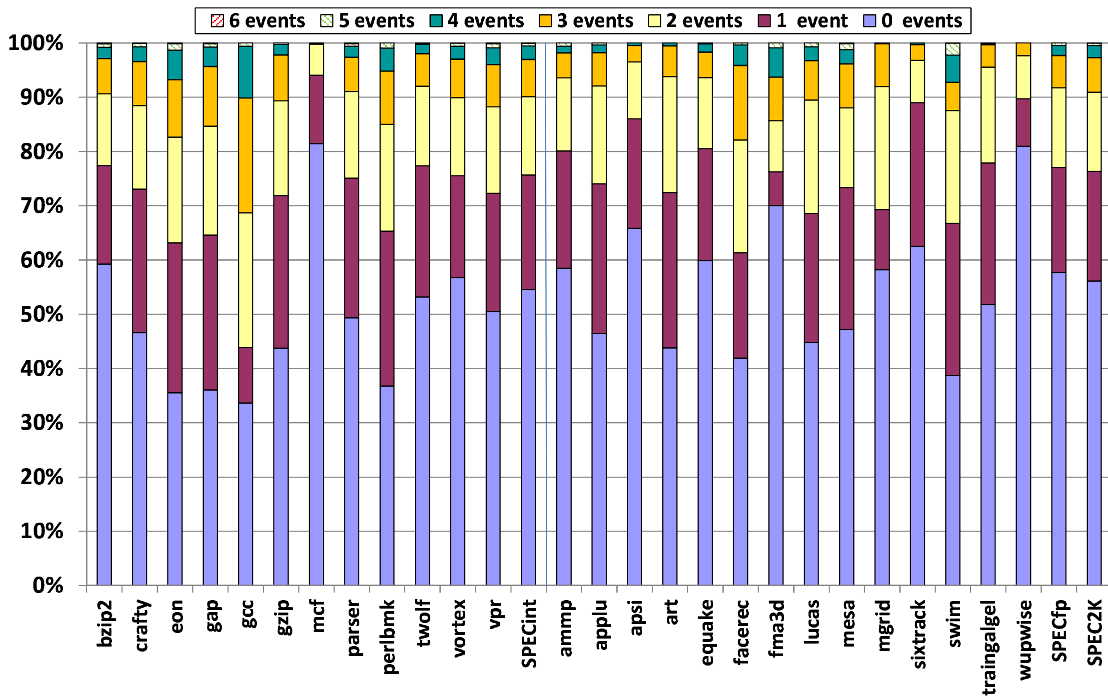


Fig. 8.8: Breakdown of number of LSQ log events generated per cycle

4`wr,12rows` has 12 '*event rows*' and each one of them is able to store up to 4 events generated by the processor in the same cycle. This means that potentially the '*LOG buffer*' can keep up to 48 valid events.

The 9 configuration groups have been formed by considering 6, 5 and 4 writable events per cycle and 12, 10 and 8 '*event rows*'. Note that an '*event row*' of 6 events is able to store the maximum number of generated events per cycle (for the case of the LSQ diagnosis). Hence, only capacity hazards may arise when using this configuration (lost events will be dropped just because the '*LOG buffer*' is full).

It can be observed that even when avoiding buffer write structural hazards (`6wr` configurations), the diagnosis coverage does not reach 100%. This is caused because the buffer has finite size and in some situations it cannot be emptied timely to the '*event merging line*', because the data cache experiences bursts of very busy phases, or because the '*Dumping Control Logic*' spends too many cycles in '*Two-Step*' mode. One observation worth highlighting is that configurations with a lower number of writable events per cycle are able to achieve a similar diagnosis coverage, when keeping constant the number of '*event rows*'. This is the case for `5wr` vs `7wr` configurations (but not `4wr` vs `5wr`). Hence, the best choice is a `5wr` configuration.

For each of the 9 groups we have considered three different '*LOG buffer*' designs (rendering a total of 27 different configurations). The '*Avg Shuffle*' design exploits

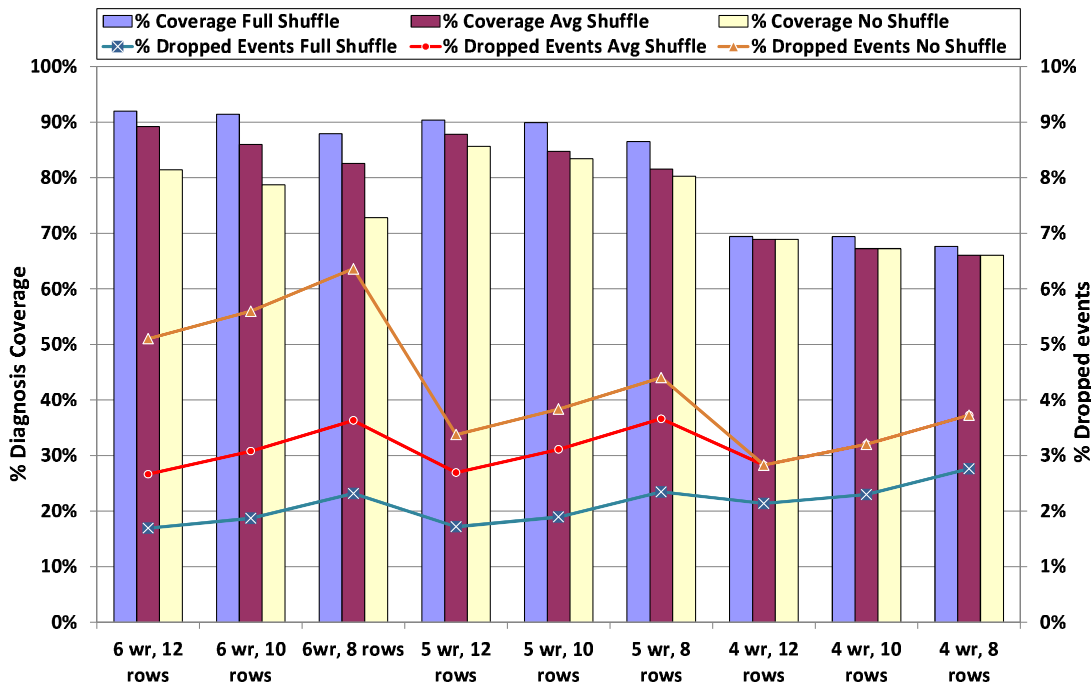


Fig. 8.9: Diagnosis coverage and dropped events for different '*LOG buffer*' configurations. '`Xwr,Yrows`' stands for number of writable events per cycle, total number of '*event rows*'

the common case where 4 events are generated per cycle, as Section 8.5.1 details. Two extreme designs have also been considered: '*Full Shuffle*' and '*No Shuffle*': the first one allows moving any event from the '*row buffer*' to any position in the '*event merging line*', whereas the second has fixed mappings among events and positions. For a '*No Shuffle*' configuration this means that for a Two-step Dump, the first 8 events from the '*event merging line*' would be first dumped, and during another cycle the last 8 events (having marked as 'empty' those events that were dumped the previous time). Note both of them rely on a 'two-step dump' process for the worst case, but the first one allows using the full capacity of the '*event merging line*' and flexible packing of events.

Results in Figure 8.9 show that the '*Avg Shuffle*' configuration is able to bridge the gap between the '*No Shuffle*' and '*Full Shuffle*' designs. This means that a diagnosis coverage similar to '*Full Shuffle*' can be achieved with simpler control logic. This is specially notable for the '`6wr`' configurations, because the '*No Shuffle*' design would always fall into a 'two-step dump' process whenever the number of non-empty events in the second bank is bigger than two. This translates into a higher pressure in the '*LOG buffer*', which in turns translates into a bigger percentage of dropped events. For the '`6wr`' configurations, when using 12 '*event rows*', the '*Full Shuffle*' achieves a diagnosis coverage of 91.98%, the '*Avg Shuffle*' achieves 89.20% and the '*No Shuffle*'

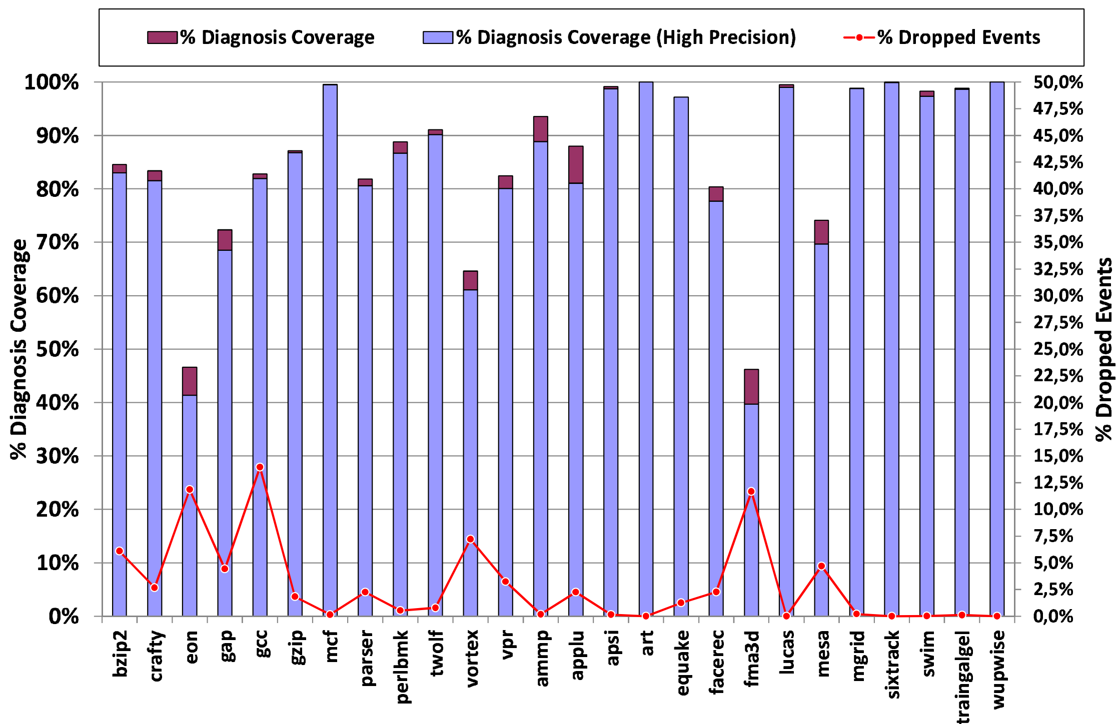


Fig. 8.10: Diagnosis coverage for a '`5wr,12rows` *LOG buffer*' configuration

81.45%, on average. On the other hand, for the `4wr` configurations both the '*Avg Shuffle*' and '*No Shuffle*' designs behave identically, because no two-step dumping is performed and all events from the two '*event rows*' fit in the '*event merging line*'.

For an '*Avg Shuffle*' or '*Full Shuffle*' configuration where the number of '*event rows*' is fixed, the percentage of dropped events minimally grows when reducing the number of writable events per cycle from 6 to 5. For `4wr` writable events, these increases start growing noticeably. For a '*No Shuffle*' configuration, the percentage of dropped events follows an inverse trend: as more events can be written per cycle, a higher percentage of the cycles the '*event merging line*' needs to be dumped in a two-step mode, leading to contention in the '*LOG buffer*'.

Finally, average diagnosis coverage does not increase linearly as we increase the number of '*event rows*'. In fact, it slightly increases from 10 '*event rows*' on: for `6wr` configurations, a '*Full Shuffle*' configuration with 10 '*event rows*' achieves an average coverage of 91.40% whereas if we use 12 '*event rows*' the average diagnose coverage just increases to 91.98%. For these configurations, the availability of the data cache write ports for dumping events ends up limiting the achievable coverage. From these results, a '*Avg Shuffle*' design configured as `5wr,12rows` seems the best design choice. On average, it is able to offer a diagnosis coverage of 87.79%.

The diagnosis capability shown by our technique varies from one application to another. This fact is shown in Figure 8.10 where we show the achieved diagnosis coverage when running the whole `SPEC` benchmark suite on a logging system configured as our best choice (`5wr,12rows` design with '*Avg Shuffle*'). Recall that coverage loss would be caused by dropped events when having a full '*LOG buffer*' or when the number of generated events in a cycle is bigger than 5.

Figure 8.10 also shows the diagnosis coverage that can be achieved when using a lower precision in the diagnosis algorithm. Specifically, this precision level allows the algorithm to pinpoint the actual failure type, not the expected failure-free scenario. Results show that for this *LOG buffer* configuration, the average diagnosis coverage can be increased to 89.84% (from 87.79%). It is interesting to highlight that from a post-silicon validation standpoint, even when the diagnosis algorithm is not able to pinpoint the root cause for a *given* fault injection, if we permanently allow the same fault to manifest again and again, then the system is able to diagnose all the failures and failure types (at least on one application). However, for runtime validation, the localization of faults will not be possible for undiagnosed errors, and the system would have to resort to coarse-grain mechanisms for recovery (rather than flushing the pipeline and re-executing) because the architectural state would be already reflecting a wrong state.

There is not a clear linear relationship between the percentage of dropped events and the diagnosis coverage. As an example, `gcc` has a higher percentage of dropped events compared to `gap`, `eon` or `vortex`, but `gcc` has a higher diagnosis coverage. Similarly, `mcf` and `perlbmk` have a similar percentage of dropped events, but `mcf` obtains a much bigger diagnosis coverage. We also notice that applications such as `eon` and `fma3d` obtain poor diagnosis coverage due to the fact that a huge amount of consecutive cycles (15 or more) the data cache is used by the application. During the length of this period, more events need to be allocated than the number of events that can be dumped from the '*event merging line*' to the memory hierarchy (a maximum of 8). When these busy periods dominate the execution of the program a large percentage of events are lost and diagnosis coverage decreases. For these glass-jaw cases even a '*LOG buffer*' of 32 '*event rows*' would not be able to provide diagnosis coverage above 60%, which is already too expensive for a post-Si or runtime validation technique.

8.6.2 Overheads

We have also quantified the performance impact introduced by our activity logging technique. Area, power and delay overheads have also been computed with respect

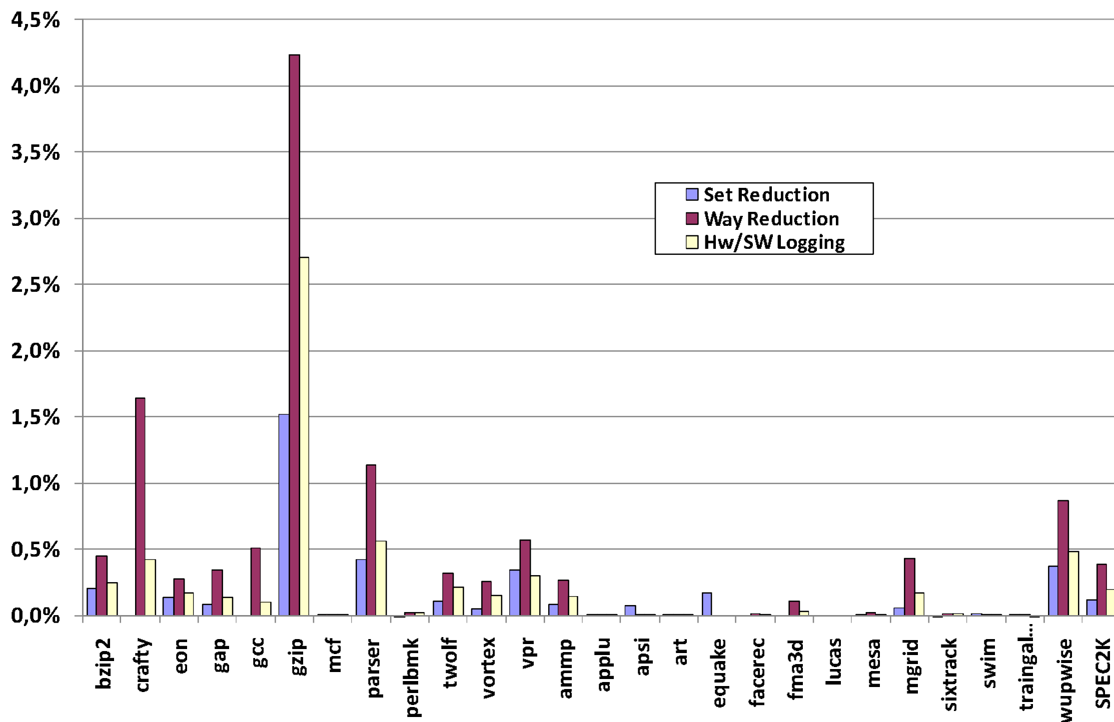


Fig. 8.11: Slowdown induced by a '`5wr,12rows` *LOG buffer*' configuration

to the data cache of our baseline processor.

Performance overhead

We have compared the performance impact with respect to approaches that sequester a cache way or a group of adjacent sets with an equivalent storage capacity (4KB). Furthermore, these two approaches are modeled in such a way that they do not compete for the data cache port availability. Results in Figure 8.11 show that in the worst case, a slowdown of 2.71% is introduced when using our hybrid hardware-sofware logging approach. When using an approach that reduces a way (1 out of 8), the worst performance slowdown is 4.23%. Reducing the cache an equivalent number of sets (8 out of 64) introduces less performance overhead (worst case is 1.52%), assuming set re-mapping [65] is enabled . However, way-reduction and set-reduction pose a problem: as the required log size increases, the performance overhead introduced would rapidly surpass the performance overhead introduced by our hybrid approach. This is due to the fact that cache lines devoted to store logging information cannot be evicted from the data cache for way-reduction and set-reduction approaches, and therefore less effective cache lines can be used by the application. On the other hand, our approach allows any log size and cache lines can be evicted from the data cache to

Table 8.2: Area, peak dynamic power and cycle time overhead for different '*LOGGING systems*'

Configuration	Area	Peak dynamic power	Cycle time
`6wr,12rows`	2.74%	5.82%	17.60%
`6wr,10rows`	1.64%	5.47%	15.36%
`6wr,8rows`	1.54%	5.10%	14.99%
`5wr,12rows`	1.65%	5.47%	15.36%
`5wr,10rows`	1.56%	5.17%	15.06%
`5wr,8rows`	1.46%	4.86%	14.75%
`4wr,12rows`	1.54%	5.10%	14.99%
`4wr,10rows`	1.46%	4.86%	14.75%
`4wr,8rows`	1.39%	4.61%	14.50%

upper level caches. On average, the set-reduction, way-reduction and our approach suffer a slowdown of 0.12%, 0.39% and 0.20%, respectively. As it can be seen, the average performance slowdown introduced by the logging component is very close to a set reduction approach that does not compete for data cache ports.

Area, Power and Delay overheads

We have quantified the area, peak dynamic power and the cycle time overhead for the '*LOGGING system*', extending our power and area models as described in Chapter 4. Table 8.2 shows the relative overheads with respect to the data cache for several configurations. The results clearly show that there is not an impact in the processor cycle time. Also, area and power costs are small. The area, peak dynamic power and cycle time ratio with respect to the data cache are 1.65%, 5.47% and 15.36%. When comparing the area against the whole core, our selected configuration (`5wr,12rows`) requires an area overhead of 0.24%. It is worth mentioning that if our diagnosis system is used exclusively during post-silicon validation, the power and slowdown penalties are only paid during these phases. Once the processor has been verified, the logging system would be deactivated. However, the low power and performance overheads of our technique makes it extremely amenable for 'runtime validation', providing continuous error detection, localization and diagnosis against faults or undiscovered bugs.

8.7 Related Work

To our knowledge, few works have attempted to increase the efficiency of diagnosis in microprocessors. Table 8.3 summarizes the features and pros/cons of each of them.

Bower et. al [26] proposed a pure hardware mechanism to locate and repair hard faults for some selected processor structures. To do so, it relies on a global error

detection mechanism (DIVA [10]) and small saturating error counters associated to every deconfigurable unit present in the processor. Also, the scheme requires tracking the instruction occupancies across different pipeline stages. Upon the detection of an error, the counters associated to each resource affected by the mismatching instruction is incremented, and they only include functional units and buffers (no control logic). Furthermore, the technique is just meant to pinpoint the fault location and is unable to provide validation information such as the reason that caused the error manifestation. Hence, it is a method more suited for run-time availavility and repairability, rather than for validation.

Trace Based Fault Diagnosis (TBFD) [97] uses a software-based fault localization mechanism, but does not perform diagnosis. The scheme relies on a cheap software-anomaly error detection mechanisms [98] to flag errors. Those errors that do not manifest as anomalies at the system level cannot be detected nor diagnosed, hence offering limited coverage. Furthermore, TBFD requires a state checkpointing mechanism to roll back the faulty core to a clean state upon an error detection. A detailed log (trace) is generated in the faulty core to record the execution trace that activated the fault. The trace tracks the usage of microarchitectural-level resources. Then, a golden trace is also generated on a fault-free core. Both traces are compared by software to achieve fault localization. Despite this feature allows paying the overhead only in the infrequent case when a fault is detected, faults that elude their manifestation during the re-execution (non repeatable errors) cannot be diagnosed. Design bugs are not diagnosable, and hence TBFD has limited usefulness during post-silicon validation: it would just help in identifying the location of just hard faults. A critical drawback of TBFD is the big latency of the underlying error detection mechanisms. The big error detection latency negatively impacts the required storage area required to store the fault-free and faulty traces, and the diagnosis complexity. An on-chip buffer is used to dump the trace into memory, but no specific details are given regarding the implementation.

IFRA (Instruction Footprint Recording and Analysis [145]) is a scheme similar to our approach. It overcomes the limitation of the previous diagnosis works by extending support to post-silicon bug localization and diagnosis. IFRA does not perform activity logging into the memory hierarchy. Instead, special distributed hardware circular buffers concurrently record microarchitectural information, in parallel to normal execution. As the recorders run in parallel with the normal execution, IFRA can diagnose non-reproducible bugs. Upon the detection of an error, this information is scanned out and analyzed off-line for bug localization. Like in our proposal, the self-consistency checks implemented in the diagnosis algorithm eliminate the need for full system-level simulation and re-execution However, the diagnosis coverage is limited by the size of the recorders and by the big latencies of the error detection mechanism.

Table 8.3: Comparative table for fault localization, logging and diagnosis techniques

	Reliability Aspects	**Sources Of Failures**	**Non-Reproducible Faults?**	**Log Based?**	**Observability Benefits?**	**Concurrent?**	**ISA Changes?**	**Area Costs**	**Performance Costs**
Distributed counters [26]	Localization	Hard errors + bugs	No	No	No	Yes	No	Minimal	No
TBFD [97]	Localization	Hard errors	No	Yes	Yes	No	No	Low	No (checkpoint)
IFRA [145]	Localization, Cause	Soft + hard errors, bugs	Yes	Yes	Yes	Yes	No	High	No
BulletProof [176]	Localization	Hard errors	No	No	No	No	No	High	Minimal
SW scan-chains [44]	Localization	Hard errors	No	No	Yes	No	Yes	Very High	Very High (Post-Si)
DACOTA [50]	Localization, Cause	Soft + hard errors, bugs	Yes	Yes	Yes	Yes	No	Minimal	Very High (Post-Si)
Our approach	Localization, Cause	Soft + hard errors, bugs	Yes	Yes	Yes	Yes	No	Very Low	Minimal

As a consequence, a big area overhead must be paid (authors report a requirement of 50KB of storage: a 2% overhead with respect to the core).

In Bulletproof [176] the components of a simple VLIW processor are periodically checked by BIST circuits in order to perform fault localization. A checkpointing mechanism creates speculative computation epochs during which the distributed BIST circuits analyze the processor components integrity during the idle component cycles. If no problem is found, the computation epoch is flagged as correct and a new fault-free checkpoint is created. Otherwise, the faulty component is deconfigured and execution is reverted back to a prior fault-free checkpoint. As opposed to other techniques, Bulletproof does not require logs, hardware recorders or software to pinpoint the fault location. However, protection is just limited to stuck-at hard faults for simple blocks. Localization coverage is in the range of 80% to 90%, whereas area overhead is significant: around 6% with respect to the whole core.

Constantinides' et al. [44] technique is aimed at the detection and localization of hard faults. It leverages the existing scan-chain DFT infrastructure to minimize costs and relies on a checkpointing mechanism for recovery. The ISA is extended in such a way that the scan-chains are visible and controllable at the software-level. A firmware periodically interrupts processor execution and uses the new instructions to inject test patterns, obtain the component outcomes and compare them against the expected ones (stored in memory). The achieved localization coverage is very high, but comes at a cost in performance (5% for a simple stuck-at fault model). The area overhead is significant, around 6%, because the scan-chains are re-organized into a tree structure. Furthermore, extending the ISA comes at a high cost and imposes compatibility requirements.

DACOTA [50] is a post-silicon technique aimed at validating the memory coherence and consistency of multi-core designs. DACOTA reconfigures a portion of the cache to log memory accesses. The cache is statically partitioned introducing performance overheads and it does not rely on a timely error detection mechanism. Instead, DACOTA performs periodic execution-diagnosis phases (enabled by a checkpointing mechanism), that introduce big performance overheads during post-silicon validation. DACOTA is able to detect errors by finding cycles among memory accesses. This work targets does not target uniprocessor correctness.

8.8 Conclusions

We have presented a novel hybrid hardware-software solution to diagnose failures during post-silicon validation and runtime operation. To show the potential of our approach, we have particularly focused on how to apply it to validate a specific func-

tionality of an advanced out-of-order processor: the memory dataflow implemented by the Load-Store Queue.

It incorporates three components: a lightweight error detection mechanism, a simple low-cost logging mechanism that that observes selected system activity during normal program execution, and a diagnosis algorithm that determines the location and the nature of the fault.

First, we have added extra value to the proposed error detection mechanisms, by extending their applicability to the post-silicon and runtime validation phases. Our error detection mechanism allows eliminating the costly simulations required to obtain the golden output to compare against, and reduces to some degree the monetary costs of the big simulation farms. In addition, the timely nature of the mechanism enables pristine logs where just relevant internal activity is captured.

The log is temporally stored in a small buffer and is progressively dumped to the data cache whenever it is idle. Architecturally, the log is stored in one or more pages of the memory space of the application being run. Hence, our logging mechanism alleviates the problem of existing state acquisition techniques, by (*i*) increasing the observability through lightweight expandable activity logs, and (*ii*) without relying on expensive validation equipment or big die overheads. Given that our logging mechanism continuously sniffs the internal activity, diagnosis coverage includes non-reproducible bugs (as opposed to most of state-of-art solutions that rely on re-execution or periodic testing). By opportunistically exploiting available hardware during idle periods, minimal system interference is introduced: no interrupts to scan out the internal data are needed and performance is minimally affected.

Upon error detection, the log is dumped from the memory hierarchy for later analysis. The diagnosis algorithm automatically analyzes the traced log and automatically diagnoses the failure. Not only the fault location is determined (as most of state-of-art solutions do), but also the wrong behavior and the failure-free expected one. Our results show that very high diagnosis coverage can be obtained at very low costs. On average, we can achieve a high-precision diagnosis coverage of 87.79% with just a 0.24% area overhead with respect to the core. Moreover, the performance slowdown introduced (due to logging purposes) is just around 0.20%, on average.

With the proposed solution, we embrace a paradigm where resilient microarchitectures assume online testing and validation functionalities to combat the diminishing effectiveness of testing and validation. The net result is a simplification of the current debugging practices, that are extremely costly, manual, time consuming and cumbersome.

CHAPTER 9

CONCLUSIONS

The increasing design complexity and the inevitable transistor vulnerability introduced with technology scaling is making fault-tolerance and post-silicon validation a concern for all processor market segments. The high overheads and the limited effectiveness of traditional solutions call for advancements to sustain the growth of the cost-sensitive microprocessor industry.

In this thesis we have embraced a paradigm where resilient microarchitectures assume online error detection and debugging functionalities to deal with these problems.

We have decomposed the basic functionalities of processors into high-level tasks and have proposed novel runtime verification solutions that when combined together can ensure the correct behavior of the processor. The proposed error detection solutions represent a departure from existing approaches by showing that re-execution is not the only way to provide fault tolerance: by exploiting high-level end-to-end microarchitectural invariants that are reusable across designs we can comprehensively protect against multiple sources of failures (including bugs) during processors' lifetime. We have made the case that light-weight error detection solutions can satisfy the requirements of minimal performance, power and area costs while at the same time offering very high reliability guarantees that can be modulated to suit design needs. Altogether, the proposed error detection solutions can potentially target 88.41% of the SDC SER FIT of a processor, and cover 77.02% of the processor area against other sources of errors (excluding protected structures).

Then, this thesis has also addressed the challenges of current post-silicon validation methodologies. As a working example, we have focused in the debugging of the

memory dataflow logic. We have shown the helpfulness of our error detection mechanisms during the post-silicon validation phases. Since our error detection methods can also catch design bugs, we minimize the need for slow system-level RTL simulation to perform bug discovery / golden output generation. Errors can therefore be detected without needing to perform architectural state comparisons or unexpected behavior sighting.

We have also advocated that new transparent continuous logging techniques combined with flexible on-chip buffer capacities allow debugging non-reproducible errors, amplifying the internal observability and reducing the dependence on costly external tools. By exploiting hardware-software synergies, our hybrid logging approach incurs in negligible area costs and causes little intrusiveness or interference to the processor regular activity. Finally, we have dealt with the problems of current debugging practices by introducing a post-failure analysis software tool that analyzes the captured traces in order to reason about the location, the temporal manifestation and the root causes behind errors.

9.1 Publications

The following is a list of all publications (subject to peer review) that are part of this thesis.

Register Dataflow Validation

- "End-to-End Register Data-Flow Continuous Self-Test", **Javier Carretero**, Pedro Chaparro, Xavier Vera, Jaume Abella, Antonio González. *Proceedings of the International Symposium on Computer Architecture (ISCA'09)*, 2009
- "Implementing End-to-End Register Data-Flow Continuous Self-Test", **Javier Carretero**, Pedro Chaparro, Xavier Vera, Jaume Abella, Antonio González. *IEEE Transactions on Computers Vol. 60 Issue 8*, 2011

Memory Flow Validation

- "On-line Failure Detection in Memory Order Buffers", **Javier Carretero**, Xavier Vera, Pedro Chaparro, Jaume Abella. *Proceedings of the International Test Conference (ITC'08)*, 2008
- "Microarchitectural Online Testing for Failure Detection in Memory Order Buffers", **Javier Carretero**, Xavier Vera, Pedro Chaparro, Jaume Abella. *IEEE Transactions on Computers Vol. 59 Issue 5*, 2010

Control Flow Recovery Validation

- "Control-Flow Recovery Validation Using Microarchitectural Invariants", **Javier Carretero**, Jaume Abella, Xavier Vera, Pedro Chaparro. *International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'11)*, 2011

Automated Fault Localization and Diagnosis

- "Hardware/Software -Based Diagnosis of Load-Store Queues Using Expandable Activity Logs", **Javier Carretero**, Xavier Vera, Jaume Abella, Tanausú Ramírez, Matteo Monchiero, Antonio González. *International Symposium on High-Performance Computer Architecture (HPCA'11)*, 2011

9.2 Open Research Directions

The results presented in this thesis open a number of interesting new research paths which we detail now:

- Even though we have deeply studied how to detect faults in the register dataflow logic, our solutions could be further enhanced or extended. It would be interesting to study methods that adaptively switch between signature generation policies based on the dynamic usage of the microarchitecture from the application being run. Given that certain signature generation policies can better handle certain failure scenarios, micro-architectural awareness of the application being run could be exploited to increase fault detection rates. In another axis, further enhancements to increase signature sizes by means of using idle hardware sub-blocks (such as data-paths with narrow values) or by means of using wider signatures in some selected blocks (such as the bypasses and not in the register file) seem very appealing.

- We believe that control flow errors in the fetch and decode logic could be detected by extending our register dataflow checking approach, rather than relying on the specific ad-hoc techniques described in Section 4.2. With the advent of hardware-software co-designed processors (like Transmeta's or NVIDIA's), ours solution could be integrated into a software control flow checker (Section 3.4) to provide register dataflow and control flow validation in an unified way. Defect tolerance could be seen as a software feature rather than a pure hardware responsibility. The software layer would be aware to some degree about the

expected encoding and sequencing of fetched instructions, and therefore could compute the expected source signatures based on the encoding of producing instructions.

- Regarding memory dataflow validation, the next natural step would be to address memory checking for multicore and multiprocessor systems, where interconnects, cache controllers, buffers, etc. are prone to faults and bugs.

- Regarding architectures for debug, future work will need to investigate methods to improve logging systems by using less-stressed and bigger caches (such as L2 or last-level caches) to reduce performance overheads and to reduce the number of non-logged events due to structural hazards. Furthermore, it is necessary to improve the scalability of these mechanisms to support the simultaneous logging of events belonging to different structures. A processor design enabling validators to choose what to trace into the logging system would significantly improve the post-silicon phases. Finally, it is also worth exploring methods to automatically derive diagnosis algorithms based on micro-architectural specifications, non-synthesizable behavioral RTL constructs (such as assertions) and RTL descriptions.

APPENDIX A

BASELINE PROCESSOR MICROARCHITECTURE

This Appendix describes the microarchitecture of the processor model that has been used in our evaluations. The objective is two-fold: first, provide details that will allow better understanding how our techniques are integrated, and second, show the complexity required to implement the different functionalities (register, control, memory flows) in an efficient out-of-order processor.

In the next two sections, we give a a high-level description of the different microarchitectural blocks and functionalities that constitute our baseline simulated core. Block descriptions are grouped depending on the part of the processor where they reside: the frontend or backend.

A.1 Processor Frontend

The processor frontend is implemented as follows:

Instruction Cache

The Instruction Cache (I$) holds *macro-instructions*. The cache is indexed with virtual addresses, and the address translation is performed by an Instruction Translation Look-aside Buffer (I-TLB) that is accessed in parallel to the cache access. The in-

struction tags are generated from the physical address and are checked against the I-TLB translation.

Multiple instructions or fragments of instructions are fetched per cycled by consecutively reading 16B from the cache that are part of the same cache line. A single read-write port is used for this purpose. Our processor does not implement a *Uop-cache*.

The fetched block of data is then placed into a *fetch buffer* to wait for decoding.

Branch Predictors, Branch Target Buffer and Return Address Stack

A branch direction (taken / not taken) is predicted with an associated confidence for conditional branches, by means of a 2-bit bimodal prediction. The processor implements a correlating GShare [112] predictor, and a global branch history that tracks the direction of the last 16 branches. This history is combined with the program counter of the branch through a XOR function to generate an index to a Pattern History Table (PHT) that contains 2-bit saturating counters. The entry is then used to make the prediction on the branch direction.

Another table, the Branch Target Buffer (BTB) is accessed in parallel to obtain the predicted target address (for taken branches). The prediction is just the target address of the previous time the branch was executed. The target is correct for direct jumps as long as the prediction is correct. Indirect branches also access this structure and the prediction accuracy depends on the regularity of the target addresses.

Special care is taken for return-from-subroutine instructions by means of a Return Address Stack (RAS).

Macro Instruction Decoders

Data from the *fetch buffer* is moved to the pre-decoding and decoding stages. The pre-decoder marks the instruction boundaries, decodes any prefixes and checks for certain properties (e.g. branches). Those instructions that lie at the end of a chunk and need further data to complete its decoding, trigger new fetch requests to the instruction cache. When pre-decoded, marked *macro-instructions* are moved to the real decoders.

Pre-decoded *macro-instructions* are emitted to the macro instruction decoders. The decoders read in the x86 instructions and emit regular, fixed length *micro-ops* which are natively processed by the underlying hardware. Up to 4 *micro-ops* are generated per cycle. Even though Intel x86 ISA is a *register memory* architecture,

macro-instructions are decoded into a set of *micro-instructions* that are meant for a load/store microarchitecture (memory is only accessed by load or store operations). Once decoding is finished, *micro-ops* are sent to the back-end for renaming, allocation, out-of-order execution and commit.

A.2 Processor Backend

The processor backend is implemented as follows:

Rename Tables and Free Lists

A group of decoded *micro-ops* are then renamed atomically and in parallel in a single cycle. The group of instructions is referred to as a *rename bundle*. The Free Lists, Rename Tables (RAT) and logic take care of removing all false register dependencies (write-after-read and write-after-write) while preserving true dependencies (read-after-write). This will enforce the register dataflow specified by the programmer/compiler even though instructions may be executed not following original program order.

Every logical destination of each instruction in the rename bundle is given an exclusive physical register at the first half of the rename cycle, that is identified (marked) through a physical *tag* or *pdst*. We refer to an allocated physical register as the *current pdst*. These *pdsts* are obtained in FIFO-order from centralized pools of available registers, called the *free lists*. There is one free list for integer (*INT*) registers and another one for floating-point (*FP*) and *SIMD* registers.

A rename table, also known as *RAT* or *Register Alias Table*, is a SRAM structure that keeps the latest translation (latest allocated physical register *pdst*) for every logical register. During the first half of the rename cycle, a thread's rename table is atomically accessed to retrieve the latest renamed *pdst* for each logical source operand in the rename bundle. Furthermore, the rename table is also accessed at this stage to obtain the previous physical registers mapped to the logical destinations (these *previous pdsts* will eventually be released back to their free list). At the end of the rename cycle (second half), the RAT table is updated to reflect the latest mappings assigned to the logical register destinations.

However, true and false register dependencies can appear internally in the rename bundle. Two types of intra-bundle dependence checks are performed. The first one identifies those logical source operands whose producing instruction is within the rename bundle. For each of those source operands, the physical register mapping

obtained from the RAT is overridden by the physical register *pdst* allocated to the closest but preceding instruction in the rename bundle producing the same logical register. The second checking identifies instructions whose logical destinations are also logical destinations of younger instructions. The first check allows enforcing correct true register dependencies. The second check allows: (i) correctly updating the rename table atomically for each logical destination with the latest *pdst* and (ii) determining for every instruction the *pdst* that from that point will have no consumers and therefore has to be eventually released to the free list. This logic is implemented as multiple set of comparators, priority encoders, and multiplexors, as described in [20].

While the thread's RAT table is updated with the new *pdsts*, the renamed instructions are moved to the next stage together with their logical destinations, *current pdsts*, *previous pdsts* and the sources' *pdsts*.

In another axis, to assist mispeculations, faults, exceptions and interrupts the renaming stages include a set of *checkpoint RATs*. A thread's rename table is backed-up into an available checkpoint RAT at regular instruction intervals. In the *Reorder Buffer* subsection, the use of these rename tables will be described.

Allocator

Renamed instructions are buffered in an *allocation buffer* where they wait for allocation. The allocation stage reserves the resources that *micro-instructions* will use during execution. This includes entries in the issue queue, the reorder buffer and the load-store queue. *Micro-instructions* are allocated in order, and if any of the required resources is not available, the allocation is stalled for all instructions.

Entries in the ROB and the load-store queue are allocated in a FIFO manner, and the identification of available entries is done by regular head and tail pointers. Since entries in the issue queue can be released in an out-of-order way, allocation to arbitrary entries in the issue queue is supported.

Even though we focus on single threaded configurations, it is worth noting that some resources are statically partitioned by thread when several thread contexts are active. This is the case of the load-store queue, the reorder buffer and the TLBs. Caches are competitively shared between threads, and the issue queue is dynamically shared between threads, based on demand. Instruction fetch, decoding, renaming and commit are time-multiplexed across the different thread contexts.

Issue Queue

Micro-ops are allocated in the Issue Queue (IQ), also known as *scheduler*. Our simulator models an issue queue based around a physical register file, where speculative and architectural registers coexist. No data-capture is performed: register values are always read upon execution.

The issue queue has a fully-associative CAM array (holding the source physical mappings) and a payload SRAM array holding other information [1]. The CAM search operation serves as the wake-up logic. An instruction is ready whenever it is notified that all its source operands have been produced or will be available once the instruction starts executing. Ready instructions send a request signal to the select logic. The select logic selects among multiple ready instructions. It follows a pseudo older-first [28] policy, and it implements a *select binding* approach to reduce the selection complexity. This means that, during allocation time, an instruction is assigned (in a balanced manner) an execution port where it will execute. Based on this pre-assignment, a group of *decentralized* select blocks operate independently by managing exclusively each of them a group of execution ports [109]. As a consequence, the complexity of a full N-of-M select scheme [55, 140] is reduced to some degree. However, a centralized select block manages that no hazard exists in the write-back (and wake-up tag broadcast) buses.

Result availability notification is implemented by means of two mechanisms. Upon instruction allocation, source operands read their readiness information from a register scoreboard, implemented as a regular bit-vector SRAM structure. In addition, upon allocation instructions clear the availability of their *current pdsts*. Instructions selected for execution notify dependents that the dependency has been resolved by means of their *current pdst*. This dependency resolution notification is implemented through *delayed tag broadcast* by means of a group of shift registers that support multi-cycle operations. As many shift registers as the issue width are available [2], and each shift register is as wide as the maximum execution latency [3]. A position in a shift register holds the *pdst* of a producing instruction. Upon instruction issue, its *current pdst* is written into the shift register associated to the execution port-stack

[1] Information such as the opcode, thread id, execution ports, destination physical mapping, immediate, LSQ position, ROB position, latency, predictions, masks, etc.

[2] Actually, as many shift registers as the total number of stacks across all execution ports (see *Functional Units* subsection).

[3] Variable latency instructions, as well as uncommon and slow operations, are handled by means of a direct tag notification signal from the execution units. This allows reducing the cost of the shift registers.

where it has been scheduled, and the *pdst* occupies the position corresponding to the instruction latency. Each cycle, all shift registers are shifted one position and the *pdsts* at the head of the shift registers are propagated through the *tag wake-up broadcast* buses to the issue queue CAM memory, and marked as ready in the register scoreboard.

The select-wakeup critical loop operates in a single cycle to support back-to-back execution for all instruction latencies. When ready and selected, instructions are issued from the issue queue and their CAM and payload RAM entries are read during the next cycle. These issued instructions are sent towards their execution ports and functional units for execution.

An entry is deallocated once it has been issued and the *load replay* mechanism guarantees that no replay event will be necessary. The processor implements a selective and parallel *load replay* scheme: the wavefront of issued instructions that depend on a missing load is stopped in a single cycle (in parallel) so that no other dependent instructions are allowed to be issued. Independent instructions are unaffected [89]. However, instructions in the wavefront are nullified as they reach the execution stages.

Functional Units

During the execution stages, Functional Units (FUs) operate on the sources of the instructions and produce the results of these computations. The pool of functional units is organized into execution ports, to avoid a big clutter of data and opcode buses going to/from every individual functional unit. Each execution port contains several functional units, supporting different types of operation. Specifically, integer arithmetic-logical units (ALUs), integer multiplication units, integer division units, address generation units (AGUs), branch-jump execution units (JEUs), floating-point units (FPUs) and SIMD units (SIMDs) are scattered across the execution ports. The rationale behind is to increase the chances that the select logic will maximize the number of execution ports busy on those situations where few instruction types are available for execution. Each execution port includes a subset of these functional unit types, and functional units of the same data type (and width) are organized as parallel *stacks* inside an execution port. A stack is a set of execution units of the same type, and only one of the functional units within a stack can be selected for execution each clock cycle (a stack has as many input multiplexors as the number of source operands a single instruction has).

Execution ports interface with the issue queue as follows. An issue queue read port output is connected to a specific execution port and hence, there is no need for a full cross-bar to route instructions (selected issue queue CAM and payload RAM entries)

to specific execution ports. Each select block owns one or several specific read ports inputs (as many as the number of execution ports it manages). As a consequence, simple routing hardware is needed to drive the instructions when issued.

All functional units of the same stack and execution port share an output multiplexor that will forward the generated value to the stack write-back bus and bypass datapath. Therefore, the select logic in the issue queue must guarantee that no structural hazards (write-back conflict) exists in the output multiplexor: only one non-bogus value will be arriving to the multiplexors inputs.

Bypass Network

Since several stages are needed to transfer the data to the register file and to update its memory cells, our processor has a bypass network with as many levels as the drive and write-back latencies. This is needed in order to avoid delaying the issuing of consumers. Our baseline processor does not implement a complete bypass network, though, to reduce the impact on cycle time, area and power. Back-to-back bypassing is allowed across (and within) execution ports as long as the consumer and producer belong to the same stack type (integer, floating-point or SIMD). One cycle of delay penalty is introduced for bypasses across stack types.

The wide and complex bypass network complicates the issue queue design. Let's retake the issue queue design description to provide details on how bypasses are handled. Control signals must be generated for the layers of multiplexors feeding the execution ports and functional units, so that the expected datapath is selected before an instruction starts executing. Furthermore, the issue queue must manage any structural hazard in the bypass network: an instruction cannot be issued if a source operand is being written-back and there is not a bypass path to obtain it.

Each scoreboard entry is replaced with a a small shift register, with as many bits as the number of bypass levels plus one. A shift register encodes a one-hot value, where the 1 indicates the bypass level or register file where the instruction could obtain it if it was selected and issued. Similarly, each CAM entry in the issue queue also holds this shift register. Upon a CAM match, the register is enabled and can start shifting. These shift registers within a CAM memory entry are initialized from the value read from the scoreboard, at allocate time. Non-existent bypass paths are deduced by the select logic blocks from the sources and destination types. The scoreboard is extended to track for each physical register the execution port where it was scheduled for production. No CAM-based bypass control is used, in order to avoid power consumption and to avoid propagating the *pdsts* across the bypass network, even though they need to be routed to the register file for write-

back purposes. Furthermore, this implementation allows avoiding unnecessary reads from the physical register file for those operands whose values are alive in the bypass network, and hence, it enables a *read port reduction* enhancement in the register file.

Register Files

Two register files (RF) are supported: one for integer values, and another for floating-point and SIMD registers. The latter register file is organized as a set of slices to support varying operand width. The physical register files holds both speculative and architectural data, and is termed as *merged physical register file*.

Each execution stack that produces a result owns a register file write port, with a width of 64 bits for integer and 128 bits for the rest. SIMD operations of 256-bits merge the output write-back ports of the FP and 128-bit SIMD stack [4].

The number of read ports for each register file is not sized as in the worst-case scenario of full issue utilization. The register files are designed to exploit the fact that many operands [31] are obtained through a bypass, and the issue queue select logic implements a *read port reduction* enhancement. The select logic knows the number of read ports that will be used by each selected instruction; if the number of required read ports by an instruction exceeds the available ones, the cascaded arbiters will not grant it a select response. Therefore, some of the register file read ports are not connected to a single specific execution port, but actually are routed to several of them. This fact complicates the issue queue design: the centralized component of the select logic is in charge of feeding the read port inputs the correct physical register ids and to swizzle the read port outputs to the correct execution port and source position.

The registers are read upon instruction issue, and hence, the schedule-to-execution latency is bigger compared to a data-capture scheme.

Load-Store Queue (Memory Order Buffer)

The Load-Store Queue (LSQ) is also commonly known as the Memory Order Buffer (MOB). Our load-store queue implements a speculative memory disambiguation policy to boost performance. Specifically, loads and stores can be executed out of order and a memory operation can be executed even if previous memory operations have

[4] FP and 128-bit SIMD stacks from the same execution port operate in a parallel and lockstep manner.

not resolved their addresses (i.e. dependencies are potentially unknown) or have not produced their data.

Two separate structures constitute the LSQ: one circular queue to keep track of all in-flight load instructions and another to track in-flight store instructions, both in program order. The LSQ supports associative searches to honor memory dependencies.

The issue queue holds the memory instructions until their operands are ready. These are: addresses for loads, and data and addresses for stores [5]. When a load instruction is ready in the issue queue and is issued, it proceeds to generate its address in an address generation unit. Right after, the address is used to access in parallel the data cache and to perform an associative search in the store queue to find a potential older in-flight producing store. Meanwhile, the load address is written into its associated load queue entry. Upon a store match, the producing store data (if available) is provided to the load, overriding the stale data obtained from the cache. This situation is called *store-to-load forwarding*. The obtained value is written-back to the register file (but not into the LSQ) and is forwarded through the bypass network. The issue queue always assume that the load latency is the data cache access latency, even if data is obtained through *store-to-load forwarding*.

A store data and a store address simply update their store queue position with the data and address, respectively. When retired, they update the memory in program order. Upon a store address generation, the load queue is associatively scanned to detect previously issued younger loads for potential *ordering violations*. Recovery from *ordering violations* is achieved by means of flushing all instructions starting from the mispeculated load, and refetching again [6].

Our processor implements a memory dependence predictor to reduce ordering violations. Specifically, a Collision History Table (CHT) predictor is used [52]. A static load instruction that suffered from *ordering violations* in the past is forced to 'wait' until all previous store addresses have been resolved. The load is marked as such in its load queue and issue queue entries. The LSQ is aware for each load if all previous stores have resolved their addresses. When the waiting is over, the LSQ notifies the issue queue.

[5]Actually, a store *macro-instruction* is frequently decoded into two *micro-instructions*: a store address instruction, and a store data instruction. The reason is that disambiguation can be performed as soon as the addresses are known and therefore, the store operation does not have to wait for the producer of the data to complete in order to compute its address.

[6]A recovery scheme that forwards the value to the offending loads is extremely complicated: it requires identifying and re-allocating dependent instructions in the issue queue for re-execution.

The CAM logic to support associative searches and to support load waiting is far more complex than the one in the issue queue because it requires age information. Being circular queues, age information cannot be efficiently deduced from a queue position because head and tail pointers advance and wrap-around. To solve this, each load and store queue entry holds a sequence number and the CAM logic operates with address, size and age information [7].

Data Cache and Second-Level Cache

The Data Cache (D$) serves requests to the load-store queue. The data cache has two read/write ports of 32B each, and can sustain up to two 256-bit loads or can operate in a lock-step way to provide 64B writes, bringing the aggregate bandwidth of 64B/cycle.

The cache is indexed with virtual addresses, and the address translation is performed by a Data Translation Look-aside Buffer (D-TLB) that is accessed in parallel to the cache access. The instruction tags are generated from the physical address and are checked against the D-TLB translation.

The data cache includes several miss status handling registers (MSHR) [91]. Missing read/write requests allocate an entry in the miss buffer (if it does not have them) and wait until the memory hierarchy obtains its data. Meanwhile, the cache allows servicing other requests and hence, does not block and exploits memory-level parallelism. Fill buffers are used to gather all consecutive data that maps to the same cache line. After all bytes arrive, the fill buffer assembles them into a cache line and then writes it into the cache. The data cache tracks cache misses using 10 line fill buffers.

In addition, the data cache controller incorporates a combining writeback buffer (WBB) for retiring stores in the load-store queue (to avoid stalls if they miss), and a writeback buffer to store evicted dirty lines. These buffers are part of the architected state, and loads must check them (in addition to the store queue and data cache).

The Second-Level Cache (L2$) can provide a full 64B line to the data or instruction cache every cycle, while allowing 16 outstanding misses. The data cache incorporates a stride prefetcher and a streaming prefetcher. Similarly, the L2$ has a streamer prefetcher.

[7] Other information such as address, size, ROB and issue queue position, program counter, and the 'wait' bit is kept in a load queue position. A store queue position holds information such as its address, size, data, ROB and issue queue position, and program counter.

Reorder Buffer

The Reorder Buffer (ROB) is a circular SRAM queue that holds all in-flight instructions. It irrevocably commits finished and speculative correct instructions in bundles, updating the architectural state (memory hierarchy, program counter and visible registers) following the program sequential execution semantics.

Our processor does not implement an architectural rename table. A group of RAT checkpoints and a ROB walking logic are used to perform mispeculation recovery, including branch mispredictions [164].

A thread's frontend RAT is checkpointed at regular distance intervals. In need of reconstructing the register mappings for a mispeculated instruction, the closest checkpoint RAT is deduced from the instruction ROB position, and is flash-copied into the thread's frontend RAT. Then, the *ROB walk logic* traverses the ROB entries and updates the frontend RAT to either undo or apply register mappings. Walk begins at the RAT checkpoint creation point and finishes at the desired instruction point. This is possible because each ROB entry holds register mapping information such as the logical destination, *previous pdst* and *current pdst* [8]. The thread's load queue and store queue tail positions are moved to ignore memory instructions past the recovery point. In parallel, the frontend is redirected to a given program counter value and renaming is stalled until the frontend RAT is reconstructed. Faults and exceptions treatment is delayed until their instructions become the head of the ROB, in order to filter the wrongly speculated ones. LSQ ordering violations are also recovered when they reach the head of the ROB, due to their relatively low occurrence. On the other hand, branch mispredictions trigger a recovery action right after they are detected.

Ready instructions at the head of the ROB also release microarchitectural resources in program order. Previous physical register tags are returned to their proper free lists for those instructions that have not been squashed [9]. Otherwise, for squashed instructions, the current physical register tag is returned to the free list [179]. Current physical registers are not freed by the *ROB walk logic* upon mispredictions, because otherwise the latency to to traverse the whole ROB would be paid in the worst case and would unnecessarily stall the renaming of the instructions from the corrected path. Checkpoint RATs are recycled when its associated instruction commits.

Our baseline processor supports multiple in-flight corrected control-flow paths

---

[8] A ROB entry holds other information such as the program counter, a ready bit, a squashed bit, a fault/exception mask, a fault/exception bit-vector, LSQ position, etc.

[9] Our processor reclaims a physical register allocated by instruction A when another instruction B that writes the same logical register and is younger than A commits.

and out-of-order branch resolution, rather than performing control path redirection at commit time. Mispeculated instructions in the shadow of a mispredicted branch are identified and squashed by means of an instruction *squashing mechanism*. Specifically, our processor implements this through branch coloring [9, 100, 109, 113] bitvectors (also called branch tagging). Branch colors ids are assigned during allocation time for conditional and indirect branches. All subsequent instructions inherit the colors of all previous branches. Upon jump verification, if the prediction was correct the branch color is released and broadcasted, so that all younger instructions reset the branch color position in their bitvector. If the prediction was incorrect, the branch color is broadcasted to identify all control-flow dependent instructions in the issue queue and ROB. Instructions in the shadow of a mispredicted jump release their entry in the issue queue (are nullified) and are forced to mark their 'squashed' bit (also known as 'bogus' bit) in their ROB position.

In any case, once an instruction is committed, its ROB entry is also released and the thread's ROB head pointer advances. Memory instructions deallocate their LSQ entries, and the ROB advances their load or store queue head pointer. Furthermore, for stores, the ROB notifies the LSQ to move its data into an available write-back combining buffer line.

Bibliography

[1] Miron Abramovici, Paul Bradley, Kumar Dwarakanath, Peter Levin, Gerard Memmi, and Dave Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *Proceedings of the 43rd Annual Design Automation Conference (DAC)*, DAC '06, pages 7–12, New York, NY, USA, 2006. ACM.

[2] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. Wiley-IEEE Press, 1994.

[3] Actel. Understanding soft and firm errors in semiconductor device. http://www.actel.com/documents/SER_FAQ.pdf, December 2002.

[4] Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon, Michael Vinov, and Avi Ziv. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design and Test of Computers*, 21(2):84–93, 2004.

[5] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO'03)*, pages 423–434, Dec. 2003.

[6] Muhammad A. Alam. A critical examination of the mechanics of dynamic NBTI for PMOSFETs. In *IEEE International Electron Devices Meeting 2003*, pages 14.4.1–14.4.4, 2003.

[7] AMD. *Revision Guide for AMD Opteron^{TM}Processors.*

[8] Hisashige Ando, Yuuji Yoshida, Aiichiro Inoue, Itsumi Sugiyama, Takeo Asakawa, Kuniki Morita, Toshiyuki Muta, Tsuyoshi Motokurumada, Seishi Okada, Hideo Yamashita, Yoshihiko Satsukawa, Akihiko Konmoto, Ryouichi Yamashita, and Hiroyuki Sugiyama. A 1.3ghz fifth generation SPARC64 microprocessor. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 702–705, New York, NY, USA, 2003. ACM.

[9] Creigton Asato. Circuit and method for tagging and invalidating speculatively executed instructions, September 11 2001. US Patent 6,289,442.

[10] Todd M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 196–207, Washington, DC, USA, 1999. IEEE Computer Society.

[11] Algirdas Avizienis. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. *IEEE Transactions on Computers*, 20(11):1322–1331, November 1971.

[12] Algirdas Avizienis. Arithmetic algorithms for error-coded operands. *IEEE Transactions on Computers*, 22(6):567–572, June 1973.

[13] Algirdas Avizienis. Arithmetic algorithms for error-coded operands. *IEEE Transactions on Computers*, 22(6):567–572, June 1973.

[14] Robert. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. In *IEEE Reliability Physics Tutorial Notes*, 2002.

[15] Robert Baumann. The impact of single event effects on advanced digital technologies - ieee eds distinguished lecturer series. http://ewh.ieee.org/r5/central_texas/eds/files/UTIBMBaumann2006.pdf, December 2006.

[16] Bob Bentley. Validating the Intel Pentium 4 microprocessor. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 244–248, New York, NY, USA, 2001. ACM.

[17] J.M. Berger. A note on error detection codes for asymmetric channels. *Information and Control*, 4(1):68–73, 1961.

[18] Keith H. Bierman, David R. Emberson, and Chen Liang T. Method and apparatus for accelerated post-silicon testing and random number generation. *Patent US7133818 B2*, 2003. Asignee: Sun Microsystems.

[19] Ronald H. Birchall. Apparatus for performing and checking logical operations, 1971, Patent Number 3,624,373.

[20] Benjamin Bishop, Thomas P. Kelliher, and Mary Jane Irwin. The design of a register renaming unit. In *Proceedings of the Ninth Great Lakes Symposium on VLSI (GLS'99)*, GLS '99, pages 34–, 1999.

[21] Maarten Boersma and Juergen Haas. Residue-based error detection for a processor execution unit that supports vector operations, 2014, Patent Number US 20140164462 A1.

[22] Darrell D. Boggs, Shlomit Weiss, and Alan Kyker. Branch ordering buffer. *Patent US 67992681 B1*, 2004. Asignee: Intel Corporation.

[23] Shekhar Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[24] Raj Chandra Bose and Dwijendra Kumar Ray Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1):68–79, 1960.

[25] Douglas C. Bossen. b-adjacent error correction. *IBM Journal Research and Development*, 14(4):402–408, July 1970.

[26] Fred A. Bower, Daniel J. Sorin, and Sule Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 197–208, Washington, DC, USA, 2005. IEEE Computer Society.

[27] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*, ISCA '00, pages 83–94, New York, NY, USA, 2000. ACM.

[28] Alper Buyuktosunoglu, Ali El-Moursy, and David H. Albonesi. An oldest-first selection logic implementation for non-compacting issue queues. In *Proceedings of the International ASIC/SOC Conference*, 2002.

[29] Harold W. Cain and Mikko H. Lipasti. Memory ordering: A value based approach. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA'04)*, 2004.

[30] Theodor Calin, Michael Nicolaidis, and R. Velazco. Upset hardened memory design for submicron CMOS technology. *IEEE Transactions on Nuclear Science*, 43(6):2874 –2878, dec 1996.

[31] Ramon Canal, Joan-Manuel Parcerisa, and Antonio Gonzlez. Dynamic cluster assignment mechanisms. In *Proceedings of the High Performance Computer Architecture (HPCA'00)*, pages 133–142, 2000.

[32] Javier Carretero, Isaac Hernández, Xavier Vera, Toni Juan, Enric Herrero, Tanausú Ramírez, Matteo Monchiero, Antonio González, Nicholas Axelos, and Daniel Sánchez. Memory controller-level extensions for GDDR5 single device data correct support. *Intel Technology Journal*, 17:102–116, 2013.

[33] Javier Carretero, Xavier Vera, Jaume Abella, Pedro Chaparro, and Antonio González. A low-overhead technique to protect the issue control logic against soft errors. In *Proceedings of the 5th IEEE Workshop on Silicon Errors in Logic - System Effects*, SELSE'09, Stanford (California), 2009. IEEE Computer Society.

[34] Jonathan Chang, George A. Reis, and David I. August. Automatic instruction-level software-only recovery. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 83–92, Washington, DC, USA, 2006. IEEE Computer Society.

[35] Kai-Hui Chang, Igor L. Markov, and Valeria Bertacco. *Functional Design Errors in Digital Circuits - Diagnosis, Correction and Repair*, volume 32 of *Lecture Notes in Electrical Engineering*. Springer, 2009.

[36] Pedro Chaparro, Jaume Abella, Javier Carretero, and Xavier Vera. Issue system protection mechanisms. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'08)*, pages 599–604, Oct. 2008.

[37] Pedro Chaparro, Jaume Abella, Xavier Vera, and Javier Carretero Casado. On-line testing for decode logic, November 2011.

[38] Saugata Chatterjee, Chris Weaver, and Todd Austin. Efficient checker processor design. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 87–97, New York, NY, USA, 2000. ACM.

[39] Chin-Long Chen. Symbol error correcting codes for memory applications. In *Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing*, FTCS '96, pages 200–, Washington, DC, USA, 1996. IEEE Computer Society.

[40] Chin-Long Chen and M. Y. (Ben) Hsiao. Error-correcting codes for semiconductor memory applications: a state-of-the-art review. *IBM Journal Research and Development*, 28(2):124–134, March 1984.

[41] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. In *Proceedings of the 21st Annual International Sympo-*

sium on Computer Architecture, ISCA '94, pages 223–232, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[42] Cristian Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, 2003.

[43] Kypros Constantinides, Onur Mutlu, and Todd Austin. Online design bug detection: RTL analysis, flexible mechanisms, and evaluation. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 282–293, Washington, DC, USA, 2008. IEEE Computer Society.

[44] Kypros Constantinides, Onur Mutlu, Todd Austin, and Valeria Bertacco. Software-based online detection of hardware defects mechanisms, architectural support, and evaluation. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 97–108, Washington, DC, USA, 2007. IEEE Computer Society.

[45] Intel®Corporation. Intel® 64 and IA-32 architectures software developer's manual (volume 3a). pages 623–674, 2010.

[46] Son T. Dao, Juergen G. Haess, Michael K. Kroener, Silvia M. Mueller, and Jochen Preiss. Distributed residue-checking of a floating point unit, 2013, Patent Number US US8566383 B2.

[47] Shidhartha Das, Carlos Tokunaga, Sanjay Pant, Wei-Hsiang Ma, Sudherssen Kalaiselvan, Kevin Lai, David M. Bull, and David. Blaauw. RazorII: In situ error detection and correction for PVT and SER tolerance. *IEEE Journal of Solid-State Circuits*, 44(1):32–48, 2009.

[48] Timothy J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. *IBM Microelectronics Division*, 1997.

[49] Andrew DeOrio, Adam Bauserman, and Valeria Bertacco. Chico: An on-chip hardware checker for pipeline control logic. In *Proceedings of the 8th International Workshop on Microprocessor Test and Verification*, MTV '07, pages 91–97, Washington, DC, USA, 2007. IEEE Computer Society.

[50] Andrew DeOrio, Ilya Wagner, and Valeria Bertacco. DACOTA: Post-silicon validation of the memory subsystem in multi-core designs. In *Proceedings of the IEEE International Conference on High-Performance Computing Architecture (HPCA'09)*, pages 405–416. IEEE Computer Society, 2009.

[51] Anand Dixit and Alan Wood. The impact of new technology on soft error rates. In *IEEE International Reliability Physics Symposium (IRPS), 2011*, pages 5B.4.1–5B.4.7, 2011.

[52] Jack Doweck. Inside Intel Core Microarchitecture and Smart Memory Access: An in-depth look at intel innovations for accelerating execution of memory-related instructions. *Intel - White Papers, Webcasts and Case Studies*, 2006.

[53] Mike Ebbers, Pilar G. Adrados, Frank Byrne, Rodney Martin, and Jon Veilleux. *Introduction to the New Mainframe: Large-Scale Commercial Computing*. IBM Form Number SG24-7175-00. IBM Redbooks, January 2007.

[54] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 7–, Washington, DC, USA, 2003. IEEE Computer Society.

[55] James A. Farrell and Gieseke Bruce A. Arbiter system for central processing unit having dual dominoed encoders for four instruction issue per machine cycle, June 2001.

[56] Manoj Franklin and Gurindar Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers (TC)*, 45(5), 1996.

[57] Daniel D. Gajski. Modular modulo 3 module, 1980, Patent Number 4,190,893.

[58] Oscar N. Garcia and Thammavarapu R. N. Rao. On the methods of checking logical operations. In *Proceedings of the Second Princeton Conference on Information Sciences and Systems*, 1968.

[59] Bradley Geden. Understand and avoid electromigration (EM) and IR-drop in custom IP blocks. *Synopsis White Paper*, November 2011.

[60] Balkaran Gill, Michael Nicolaidis, Francis Wolff, Chris Papachristou, and Steven Garverick. An efficient BICS design for SEUs detection and correction in semiconductor memories. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '05, pages 592–597, Washington, DC, USA, 2005. IEEE Computer Society.

[61] Balkaran S. Gill, Chris Papachristou, Francis G. Wolff, and Norbert Seifert. Node sensitivity analysis for soft errors in CMOS logic. In *Proceedings of the IEEE International Test Conference*, ITC'05, pages 9 pp. –972, nov. 2005.

[62] Bart Goeman, Hans Vandierendonck, and Koen de Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the 7th International Symposium on High-Performance Architecture (HPCA'01)*, volume 00, page 0207, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[63] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 98–109, New York, NY, USA, 2003. ACM.

[64] Mohamed Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 172–183, Washington, DC, USA, 2005. IEEE Computer Society.

[65] Antonio González, Mateo Valero, Nigel Topham, and Joan Manel Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *Proceedings of the International Conference on Supercomputing (ICS'97)*, 1997.

[66] José González and Antonio González. Speculative execution via address prediction and data prefetching. In *Proceedings of the 11th International Conference on Supercomputing (ICS'97)*, ICS '97, pages 196–203, New York, NY, USA, 1997. ACM.

[67] Gary D. Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[68] Juergen Haess, Michael K. Kroener, Silvia M. Mueller, and Kerstin Schelm. Exponent flow checking, 2014, Patent Number US 20140164463 A1.

[69] Mark D. Hammig. The design and construction of a mechanical radiation detector. In *Proceedings of IEEE Nuclear Science Symposium*, pages 803–805, 1998.

[70] Richard W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 26(2):147–160, 1950.

[71] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Allan Kyker, and Patrice Roussel. The microarchitecture of the Pentium®4 processor. *Intel Technology Journal*, 1, 2001.

[72] Alexis Hocquenghem. Codes correcteurs d'erreurs. *Chiffres Journal*, 2:147–156, 1959.

[73] M. Y. (Ben) Hsiao. A class of optimal minimum odd-weight-column SEC-DED codes. *IBM Journal Research and Development*, 14(4):395–401, July 1970.

[74] Sorin Iacobovici. Residue-based error detection for a shift operation, 2007, Patent Number US 2007/0043796 A1.

[75] Sorin Iacobovici. End-to-end residue based protection of an execution pipeline, 2009, Patent Number US 7,555,692 B1.

[76] Sorin Iacobovici. End-to-end residue-based protection of an execution pipeline that supports floating point operations, 2010, Patent Number US 7,769,795 B1.

[77] Sorin Iacobovici. Residue based error detection for integer and floating point execution units, 2014,Patent Number US 20140188965 A1.

[78] Intel. *Using the Intel®ICH Family Watchdog Timer (WDT).*

[79] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, September 2014.

[80] International Technology Roadmap for Semiconductors ITRS. Critical reliability challenges for the international technology roadmap for semiconductors (ITRS). Technical report, ITRS, 2003.

[81] International Technology Roadmap for Semiconductors ITRS. International technology roadmap for semiconductors executive summary. Technical report, ITRS, 2007.

[82] International Technology Roadmap for Semiconductors ITRS. International technology roadmap for semiconductors 2012 update overview. Technical report, ITRS, 2012.

[83] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[84] Doug Josephson. The good, the bad, and the ugly of silicon debug. In *Proceedings of the 43rd Annual Design Automation Conference (DAC)*, DAC '06, pages 3–6, New York, NY, USA, 2006. ACM.

[85] David Kanter. Intels Sandy Bridge Microarchitecture. http://www.realworldtech.com/sandy-bridge/, September 2010.

[86] Tanay Karnik, Sriram Vangal, Venkat Veeramachaneni, Peter Hazucha, Vasantha Erraguntla, and Shekhar Borkar. Selective node engineering for chip-level soft error rate improvement. In *Digest of Technical Papers of the Symposium on VLSI Circuits*, pages 204 – 205, 2002.

[87] Jagannath Keshava, Nagib Hakim, and Chinna Prudvi. Post-silicon validation challenges: How EDA and academia can help. In *Proceedings of the 47th Design Automation Conference (DAC'10)*, DAC '10, pages 3–7, New York, NY, USA, 2010. ACM.

[88] Jagannath Keshava, Nagib Hakim, and Chinna Prudvi. Post-silicon validation challenges: How EDA and academia can help. In *Proceedings of the 47th Design Automation Conference (DAC'10)*, DAC '10, pages 3–7, New York, NY, USA, 2010. ACM.

[89] Ilhyun Kim and Mikko H. Lipasti. Understanding scheduling replay schemes. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA'04)*, HPCA '04, pages 198–, Washington, DC, USA, 2004. IEEE Computer Society.

[90] Philip Koopman and Tridib Chakravarty. Cyclic redundancy code (CRC) polynomial selection for embedded networks. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '04, pages 145–, Washington, DC, USA, 2004. IEEE Computer Society.

[91] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th International Symposium on Computer Architecture (ISCA)*, ISCA '81, pages 81–87, 1981.

[92] Simeon J. Krumbein. Metallic electromigration phenomena. *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, 11(1):5–15, 1988.

[93] Sumeet Kumar and Aneesh Aggarwal. Speculative instruction validation for performance-reliability trade-off. In *In Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture*, HPCA'08, pages 405–414, 2008.

[94] Ravishankar Kuppuswamy, Peter DesRosier, Derek Feltham, Rehan Sheikh, and Paul Thadikaran. Full Hold-Scan Systems in Microprocessors: Cost/Benefit Analysis. *Intel Technical Journal*, 8(1):63–72, February 2004.

[95] Anand Lal Shimpi. The source of intel's cougar point sata bug. http://www.anandtech.com/show/4143/, January 2011.

[96] Glen G. Langdon and C. K. Tang. Concurrent error detection for group look-ahead binary adders. *IBM Journal Research and Development*, 14(5):563–573, September 1970.

[97] Man-Lap Li, Pradeep Ramachandran, Swarup K. Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC*, pages 22–31, 2008.

[98] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 265–276, New York, NY, USA, 2008. ACM.

[99] Sheng Li, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. McPAT 1.0: An integrated power, area, and timing modeling framework for multicore architectures. Technical report, HP Labs, 2009.

[100] Mikko H. Lipasti. ECE/CS 752 Advanced Computer Architecture I Course. Slides, University of Wisconsin-Madison, 2015.

[101] Mikko Herman Lipasti. *Value Locality and Speculative Execution*. PhD thesis, Pittsburgh, PA, USA, 1998. UMI Order No. GAX98-06874.

[102] D. Lipetz and E. Schwarz. Self checking in current floating-point units. In *IEEE Symposium on Computer Arithmetic (ARITH)*, pages 73–76, July 2011.

[103] T. Litt. Support for debugging in the alpha 21364 microprocessor. In *Proceedings of the International Test Conference (ITC), 2002*, pages 584–589, 2002.

[104] Jien-Chung Lo. Reliable floating-point arithmetic algorithms for Berger encoded operands. In *Proceedings of the IEEE International Conference on Computer Design on VLSI in Computer & Processors*, ICCD '92, pages 110–113, Washington, DC, USA, 1992. IEEE Computer Society.

[105] Jien-Chung Lo. Reliable floating-point arithmetic algorithms for error-coded operands. *IEEE Transaction on Computers*, 43(4):400–412, April 1994.

[106] Jien-Chung Lo, Suchai Thanawastien, and Thammavarapu R. N. Rao. Berger check prediction for array multipliers and array dividers. *IEEE Transactions on Computers*, 42(7):892–896, 1993.

[107] Jien-Chung Lo, Suchai Thanawastien, Thammavarapu R. N. Rao, and Michael Nicolaidis. An SFS Berger check prediction ALU and its application to self-checking processor designs. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 11(4):525–540, 1992.

[108] Jien-Chung. Lo, Suchai Thanawastien, and Thammavarapu R.N. Rao. Concurrent error detection in arithmetic and logical operations using Berger codes. In *Proceedings of 9th Symposium on Computer Arithmetic*, pages 233 –240, sep 1989.

[109] Gabriel Loh. CS8803: Advanced Microarchitecture Course. Slides, Georgia Institute of Technology, 2005.

[110] Aamer Mahmood and Edward J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2):160–174, February 1988.

[111] Ritesh Mastipuram and Edwin C. Wee. Soft error's impact on system reliability. *Electronics Design, Strategy, News (EDN)*, pages 69–74, September 2004.

[112] Scott McFarling. Combining branch predictors. Technical Report WRL TN-36, Western Research Laboratory, June 1993.

[113] Michael S. McIlvaine, James N. Dieffenderfer, and Thomas A. Sartorius. Method and apparatus for managing instruction flushing in a microprocessor's instruction pipeline, June 2011.

[114] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 210–222, Washington, DC, USA, 2007. IEEE Computer Society.

[115] Albert Meixner and Daniel J. Sorin. Error detection using dynamic dataflow verification. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 104–118, Washington, DC, USA, 2007. IEEE Computer Society.

[116] Albert Meixner and Daniel J. Sorin. Detouring: Translating software to circumvent hard faults in simple cores. In *Proceedings of the Conference on Dependable*

Systems and Networks (DSN'08), pages 80–89. Proceedings of the Conference on Dependable Systems and Networks (DSN'08), IEEE Computer Society, 2008.

[117] Sarah E. Michalak, Kevin W. Harris, Nicolas W. Hengartner, Bruce E. Takala, and Stephen A. Wender. Predicting the number of fatal soft errors in los alamos national laboratory's ASC Q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, 2005.

[118] Sun Microsystems. OpenSPARC T2 system-on-chip (SoC) microarchitecture specification. 2008.

[119] Subhasish Mitra and Edward J. McCluskey. Which concurrent error detection scheme to choose? In *Proceedings of the IEEE International Test Conference*, ITC '00, pages 985–, Washington, DC, USA, 2000. IEEE Computer Society.

[120] Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. Robust system design with built-in soft-error resilience. *IEEE Computers*, 38(2):43–52, February 2005.

[121] Subhasish Mitra, Sanjit A. Seshia, and Nicola Nicolici. Post-silicon validation opportunities, challenges and recent advances. In *Proceedings of the 47th Design Automation Conference (DAC'10)*, DAC '10, pages 12–17, New York, NY, USA, 2010. ACM.

[122] Kartik Mohanram and Nur A. Touba. Cost-effective approach for reducing soft error failure rate in logic circuits. In *Proceedings of the IEEE International Test Conference*, volume 1 of *ITC'03*, pages 893 – 901, 30-oct. 2, 2003.

[123] Mark Moir, Kevin Moore, and Dan Nussbaum. The adaptive transactional memory test platform: A tool for experimenting with transactional code for Rock (poster). In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 362–362, New York, NY, USA, 2008. Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA'08), ACM.

[124] Robert Carl Moncsko. Method and apparatus for modulus error checking, 1998, Patent Number 5,742,533.

[125] P. Monteiro and Thammavarapu R. N. Rao. A residue checker for arithmetic and logical operations. In *Proceedings of 2nd Fault Tolerant Computing Symposium*, 1972.

[126] Shubhendu S. Mukherjee, Joel Emer, and Steven K. Reinhardt. The soft error problem: An architectural perspective. In *Proceedings of the 11th International*

Symposium on High-Performance Computer Architecture (HPCA), HPCA '05, pages 243–247, Washington, DC, USA, 2005. IEEE Computer Society.

[127] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 99–110, Washington, DC, USA, 2002. IEEE Computer Society.

[128] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 29–, Washington, DC, USA, 2003. IEEE Computer Society.

[129] Shubu Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[130] Matthew Murray. Sandy Bridge: Intels Next-Generation Microarchitecture Revealed. http://www.extremetech.com/computing/83848-sandy-bridge-intels-nextgeneration-microarchitecture-revealed, September 2010.

[131] Masood Namjoo. Techniques for concurrent testing of VLSI processor operation. In *Proceedings of the International Testing Conference*, pages 461–468. IEEE Computer Society, 1982.

[132] Egas Henes Neto, Ivandro Ribeiro, Michele Vieira, Gilson Wirth, and Fernanda Lima Kastensmidt. Using bulk built-in current sensors to detect soft errors. *IEEE Micro*, 26(5):10–18, September 2006.

[133] Hang Nguyen. Resiliency challenges in future communications infrastructure. In *Proceedings of the IEEE Comunications and Reliability Workshop (CQR'14)*, CQR'14. IEEE Computer Society, 2014.

[134] Michael Nicolaidis. Carry checking/parity prediction adders and alus. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(1):121–128, February 2003.

[135] Michael Nicolaidis and Ricardo O. Duarte. Design of fault-secure parity-prediction Booth multipliers. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '98, pages 7–14, Washington, DC, USA, 1998. IEEE Computer Society.

[136] Michael Nicolaidis, Ricardo O. Duarte, Salvador Manich, and Joan Figueras. Fault-secure parity prediction arithmetic operators. *IEEE Design and Test*, 14(2):60–71, April 1997.

[137] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51:111–122, 2002.

[138] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error detection by duplicated instructions in superscalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

[139] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Quantifying the complexity of superscalar processors. Technical Report Tech. Report 96-1308, Dept. of CS, Univ. of Wisconsin,, 1996.

[140] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA'97)*, ISCA '97, pages 206–218, New York, NY, USA, 1997.

[141] Abhisek Pan, James W. Tschanz, and Sandip Kundu. A low cost scheme for reducing silent data corruption in large arithmetic circuits. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, DFT '08, pages 343–351, Washington, DC, USA, 2008. IEEE Computer Society.

[142] D.B. Papworth, A.F. Glew, M.A. Fetterman, G.J. Hinton, R.P. Colwell, S.J. Griffith, S.R. Gupta, and N. Hedge. Entry allocation in a circular buffer. *Patent US 5584037*, 1996. Asignee: Intel Corporation.

[143] Angshuman Parashar, Anand Sivasubramaniam, and Sudhanva Gurumurthi. SlicK: slice-based locality exploitation for efficient redundant multithreading. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XII, pages 95–105, New York, NY, USA, 2006. ACM.

[144] Il Park, Chong Liang Ooi, and T.N. Vijaykumar. Reducing design complexity of the load/store queue. In *Proceedings of the International Symposium on Microarchitecture (MICRO-36)*, 2003.

[145] Sung-Boem Park and Subhasish Mitra. IFRA: Instruction footprint recording and analysis for post-silicon bug localization in processors. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 373–378, New York, NY, USA, 2008. ACM.

[146] Praveen Parvathala, Kaila Maneparambil, and William Lindsay. FRITS : A microprocessor functional bist method. In *Proceedings of the 2002 IEEE International Test Conference*, ITC '02, pages 590–, Washington, DC, USA, 2002. IEEE Computer Society.

[147] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'04)*, MICRO 37, pages 81–92, Washington, DC, USA, 2004. IEEE Computer Society.

[148] Priyadarsan Patra. On the cusp of a validation wall. *IEEE Design & Test of Computers*, 24(2):193–196, 2007.

[149] Michael D. Powell, Arijit Biswas, Shantanu Gupta, and Shubhendu S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*, ISCA '09, pages 93–104, New York, NY, USA, 2009. ACM.

[150] Vaughan Pratt. Anatomy of the pentium bug. In *Theory and Practice of Software Development (TAPSOFT)*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107. Springer Berlin Heidelberg, 1995.

[151] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings. 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 111–122, 2002.

[152] Thammavarapu R. N. Rao. *Error Coding for Arithmetic Processors*. Academic Press, Inc., Orlando, FL, USA, 1974.

[153] Thammavarapu R. N. Rao and Eiji Fujiwara. *Error-Control Coding for Computer Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[154] Vimal K. Reddy, Eric Rotenberg, and Ahmed S. Al-Zawawi. Assertion-based microarchitecture design for improved reliability. In *Proceedings of the 24th International Conference on Computer Design*, ICCD'06. IEEE Computer Society, 2006.

[155] Vimal Kodandarama Reddy. *Exploiting Microarchitecture Insights for Efficient Fault Tolerance*. PhD thesis, 2007.

[156] Kevin Reick, Pia N. Sanda, Scott B. Swaney, Jeffrey W Kellington, Michael J. Mack, Michael S. Floyd, and Daniel Henderson. Fault-tolerant design of the IBM Power6 microprocessor. In *Symposium on Hot Chips*, 2007.

[157] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 25–36, New York, NY, USA, 2000. ACM.

[158] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[159] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 148–159, Washington, DC, USA, 2005. IEEE Computer Society.

[160] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(4):366–396, December 2005.

[161] Leonard R. Rockett Jr. An SEU-hardened CMOS data latch design. *IEEE Transactions on Nuclear Science*, 35(6):1682 –1687, dec 1988.

[162] Eric Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, FTCS '99, pages 84–, Washington, DC, USA, 1999. IEEE Computer Society.

[163] Hemant Rotithor. Post-silicon validation methodology for microprocessors. *IEEE Desing and Test*, 17(4):77–88, October 2000.

[164] Elham Safi, Patrick Akl, Andreas Moshovos, Andreas Veneris, and Aggeliki Arapoyianni. On the latency, energy and area of checkpointed, superscalar register alias tables. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design (ISPLED'07)*, ISLPED '07, pages 379–382, New York, NY, USA, 2007. ACM.

[165] Giacinto P. Saggese, Nicholas J. Wang, Zbigniew T. Kalbarczyk, Sanjay J. Patel, and Ravishankar K. Iyer. An experimental study of soft errors in microprocessors. *IEEE Micro*, 25(6):30–39, 2005.

[166] Peter G. Sassone, Jeff Rupley, II, Edward Brekelbaum, Gabriel H. Loh, and Bryan Black. Matrix scheduler reloaded. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (HPCA'07)*, ISCA '07, pages 335–346, New York, NY, USA, 2007. ACM.

[167] Yiannakis Sazeides and James E. Smith. Implementations of context based value predictors. Technical Report ECE-TR-97-8, University of Wisconsin-Madison, 1997.

[168] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 248–258, Washington, DC, USA, 1997. IEEE Computer Society.

[169] Mark M. Schaffer. Residue checking apparatus for detecting errors in add, substract, multiply, divide and square root operations, 1990, Patent Number 4,926,374.

[170] Ute Schiffel. *Hardware Error Detection Using AN-Codes.* PhD thesis, Technische Universität Dresden, Dresden, 01062 Dresden, Germany, 2011.

[171] Michael A. Schuette and John Paul Shen. Processor control flow monitoring using signatured instruction streams. *IEEE Transactions on Computers*, 36(3):264–276, March 1987.

[172] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. Scalable hardware memory disambiguation for high ILP processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO-36)*, 2003.

[173] Anand Lal Shimpi. Intel's Sandy Bridge Architecture Exposed. http://www.anandtech.com/show/3922/intels-sandy-bridge-architecture-exposed, September 2010.

[174] Premkishore Shivakumar and Mich Kistler. Modeling the impact of device and pipeline scaling on the soft error rate of processor elements. Technical report, The University of Texas at Austin and IBM Austin Research Laboratory, 2002.

[175] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error

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

[176] Smitha Shyam, Kypros Constantinides, Sujay Phadke, Valeria Bertacco, and Todd Austin. Ultra low-cost defect protection for microprocessor pipelines. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 73–82, New York, NY, USA, 2006. ACM.

[177] K.Y. Sih. Checking logical operations by residues, 1972, Patent Number IP-COM000078397D.

[178] Isic Silas, Igor Frumkin, Eilon Hazan, Ehud Mor, and Genadiy Zobin. System-level validation of the Intel Pentium M processor. *Intel Technology Journal*, 7(2):37–43, May 2003.

[179] Dezsö Sima. The design space of register renaming techniques. *IEEE Micro*, 20(5):70–83, September 2000.

[180] Graham Singer. The rise and fall of amd. http://www.techspot.com/article/599-amd-rise-and-fall/page2.html, November 2012.

[181] Timothy J. Slegel, Robert M. Averill III, Mark A. Check, Bruce C. Giamei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, March 1999.

[182] Jared C. Smolens, Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Fingerprinting: Bounding soft-error detection latency and bandwidth. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XI, pages 224–234, New York, NY, USA, 2004. ACM.

[183] Jared C. Smolens, Jangwoo Kim, James C. Hoe, and Babak Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 257–268, Washington, DC, USA, 2004. IEEE Computer Society.

[184] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 194–205, New York, NY, USA, 1997. ACM.

[185] Vision Solutions. Assessing the financial impact of downtime. http://www.strategiccompanies.com/pdfs/Assessing2008.

[186] Daniel J. Sorin. *Fault Tolerant Computer Architecture*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[187] Daniel J. Sorin, Milo M.K. Martin, Mark D. Hill, and David A. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings. 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 123–134, 2002.

[188] Lisa Spainhower and Thomas A. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 43(5):863–873, 1999.

[189] Uwe Sparmann and Sudhakar M. Reddy. On the effectiveness of residue code checking for parallel two's complement multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(2):227–239, June 1996.

[190] http://www.spec.org/cpu2000/ SPEC CPU 2000, 2000.

[191] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The case for lifetime reliability-aware microprocessors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, ISCA '04, pages 276–, Washington, DC, USA, 2004. IEEE Computer Society.

[192] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The impact of technology scaling on lifetime reliability. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN)*, DSN '04, pages 177–, Washington, DC, USA, 2004. IEEE Computer Society.

[193] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Lifetime reliability: Toward an architectural solution. *IEEE Micro*, 25(3):70–80, May-June 2005.

[194] James H. Stathis. Physical and predictive models of ultrathin oxide reliability in CMOS devices and circuits. *IEEE Transactions on Device and Materials Reliability*, 1(1):43–59, 2001.

[195] Dmitri Strukov. The area and latency tradeoffs of binary bit-parallel BCH decoders for prospective nanoelectronic memories. In *Proceedings of 40th Asilomar Conference on Signals, Systems and Computers Signals, Systems and Computers*, ACSSC '06, pages 1183 –1187, 29 2006-nov. 1 2006.

[196] Sangeetha Sudhakrishnan, Rigo Dicochea, and Jose Renau. Releasing efficient beta cores to market early. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 213–222, New York, NY, USA, 2011. ACM.

[197] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-IX, pages 257–268, New York, NY, USA, 2000. ACM.

[198] Stratus Technologies. ftServer architecture http://www.stratus.com/ products/ ftserversystems/ uptimetechnology/ ftserverarchitecture.aspx.

[199] Joel M. Tendler, J. Steve Dodson, J. S. Fields, Hung Le, and Balaram Sinharoy. POWER4 system microarchitecture. *IBM Journal Research and Development*, 46(1):5–25, January 2002.

[200] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. CACTI 5.1. *HP Technical Report HPL-2008-20*, 2008.

[201] Keshavan Tiruvallur. Beyond design... post-silicon validation challenges and opportunities. http://cache-www.intel.com/cd/00/00/51/61/516195_516195.pdf, 2011.

[202] Gaurang Upsani, Xavier Vera, and Antonio Gonzàlez. Setting an error detection infrastructure with low cost acoustic wave detectors. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA'12, 2012.

[203] Fabian L. Vargas and Michael Nicolaidis. SEU-tolerant SRAM design based on current monitoring. In *Proceedings of the 24th International Symposium on Fault Tolerant Computing*, FTCS 94, pages 106–115, 1994.

[204] Arman Vassighi and Manoj Sachdev. Thermal runaway in integrated circuits. *IEEE Transactions Device and Materials Reliability*, 6(2):300–305, 2006.

[205] Xavier Vera, Jaume Abella, Javier Carretero, and Antonio González. Selective replication: A lightweight technique for soft errors. *ACM Transactions on Computers Systems*, 27(4):8:1–8:30, January 2010.

[206] Bart Vermeulen and Sandeep K. Goel. Design for debug: Catching design errors in digital chips. *IEEE Desing and Test*, 19(3):37–45, 2002.

[207] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 87–98, Washington, DC, USA, 2002. IEEE Computer Society.

[208] Ilya Wagner and Valeria Bertacco. *Post-Silicon and Runtime Verification for Modern Processors*, volume XVII. Springer, 2011.

[209] John F. Wakerly. Principles of self-checking processor design and an example. Technical report, Stanford, CA, USA, 1975.

[210] John F. Wakerly. *Error Detecting Codes, Self-Checking Circuits and Applications.* Computer design and architecture series. North-Holland, 1978.

[211] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society.

[212] Nicholas J. Wang, Justin Quek, Todd M. Rafacz, and Sanjay J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, DSN '04, pages 61–, Washington, DC, USA, 2004. IEEE Computer Society.

[213] Tse Lin Wang. Error detection system, 1974, Patent Number 3,814,923.

[214] Yi-Min Wang, P. Y. Chung, Y. Huang, and E. N. Elnozahy. Integrating checkpointing with transaction processing. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, FTCS '97, pages 304–, Washington, DC, USA, 1997. Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS '97), IEEE Computer Society.

[215] Chris Weaver and Todd M. Austin. A fault tolerant approach to microprocessor designs. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, DSN '01, pages 411–420, Washington, DC, USA, 2001. IEEE Computer Society.

[216] Charles Webb. z6 - the next generation mainframe microprocessor. *Hot Chips*, 2007.

[217] Wikipedia. Application checkpointing. http://en.wikipedia.org/wiki/Application_checkpointing.

[218] Wikipedia. Cyrix coma bug. http://en.wikipedia.org/wiki/Cyrix_coma_bug.

[219] Kent D. Wilken and John Paul Shen. Continuous signature monitoring: Low-cost concurrent detection of processor control errors. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 9(6):629–641, 1990.

[220] Marcelo Yuffe, Ernest Knoll, Moty Mehalel, Joseph Shor, and Tsvika Kurts. A fully integrated multi-CPU, GPU and memory controller 32nm processor. In *Solid-State Circuits IEEE International Conference*, ISSCC'11, pages 264–266, 2011.

[221] Peng Zhou, Soner Önder, and Steve Carr. Fast branch misprediction recovery in out-of-order superscalar processors. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05)*, New York, NY, USA, 2005.

[222] James F. Ziegler and William A. Lanford. The effect of sea level cosmic rays on electronic devices. In *IEEE International Solid-State Circuits Conference*, volume XXIII, pages 70–71, 1980.