Efficient Fault Tolerance in Chip Multiprocessors Using Critical Value Forwarding

A Thesis
Submitted For the Degree of Master of Science (Engineering)
in the Faculty of Engineering

by
Pramod Subramanyan

Supercomputer Education and Research Center
Indian Institute of Science
BANGALORE – 560 012
June 2010
©Pramod Subramanyan
June 2010
All rights reserved
To my parents.
Acknowledgements

I owe many many thanks to my advisor, Dr. Virendra Singh. He gave me tremendous freedom in choosing a topic for research and a lot of support to try different ideas. I am very grateful to him for his advice, support and guidance over the past two years.

I would like to thank Prof. Kewal Saluja for his technical feedback and his kind words of encouragement. I would also like to thank Prof. Erik Larsson for his feedback, encouragement and hospitality during my visit to Linköping University.

I also owe thanks to Prof. Mathew Jacob for a wonderfully taught computer architecture course that laid the foundation for the work done in this thesis.

Many people at IISc have made my stay enjoyable. I would like to thank all of them for their companionship during my stay here. They are (in alphabetical order): Ashwin, BT, Janakiraman, Jaynarayan, Keshavan, Pawan, Preeti, Pushkar, Rajath, Raju, Sreepathi, Tilak, ...

Last but not least, I would like to thank my parents and my brother for their continued support. This thesis is dedicated to my parents for the sacrifices they have made over the years to help support my education.
Publications Based On This Thesis


Abstract

Relentless CMOS scaling coupled with lower design tolerances is making ICs increasingly susceptible to transient faults, wear-out related permanent faults and process variations. Decreasing CMOS reliability implies that high-availability systems which were previously restricted to the domain of mainframe computers or specially designed fault-tolerant systems may become important for the commodity market as well. In this thesis we tackle the problem of enabling efficient, low cost and configurable fault-tolerance using Chip Multiprocessors (CMPs).

Our work studies architectural fault detection methods based on redundant execution, specifically focusing on “leader-follower” architectures. In such architectures redundant execution is performed on two cores/threads of a CMP. One thread acts as the leading thread while the other acts as the trailing thread. The leading thread assists the execution of the trailing thread by forwarding the results of its execution. These forwarded results are used as predictions in the trailing thread and help improve its performance. In this thesis, we introduce a new form of execution assistance called critical value forwarding. Critical value forwarding uses heuristics to identify instructions on the critical path of execution and forwards the results of these instructions to the trailing core. The advantage of critical value forwarding is that it provides much of the speedup obtained by forwarding all values at a fraction of the bandwidth cost.

We propose two architectures to exploit the idea of critical value forwarding. The first of these operates the trailing core at lower voltage/frequency levels in order to provide energy-efficient redundant execution. In this context, we also introduce algorithms to dynamically adapt the voltage/frequency level of the trailing core based on program
behavior. Our experimental evaluation shows that this proposal consumes only 1.26 times the energy of a non-fault-tolerant baseline and has a mean performance overhead of about 1%. We compare our proposal to two previous energy-efficient fault-tolerant CMP proposals and find that our proposal delivers higher energy-efficiency and lower performance degradation than both while providing a similar level of fault coverage.

Our second proposal uses the idea of critical value forwarding to improve fault-tolerant CMP throughput. This is done by using coarse-grained multithreading to multiplex trailing threads on a single core. Our evaluation shows that this architecture delivers 9–13% higher throughput than previous proposals, including one configuration that uses simultaneous multithreading (SMT) to multiplex trailing threads. Since this proposal increases fault-tolerant CMP throughput by executing multiple threads on a single core, it comes at a modest cost in single-threaded performance, a mean slowdown between 11–14%.
Keywords

Performance, Reliability, Fault Tolerance, Transient faults, Permanent faults, Redundant Execution, Microarchitecture, Energy-efficient Architectures.
# Contents

Acknowledgements  
Publications Based On This Thesis  
Abstract  
Keywords  

1 Introduction  
1.1 Challenges of Technology Scaling  
1.1.1 Wearout Failure Modes in Future Technologies  
1.1.2 Radiation Induced Transient Faults  
1.2 Faults and Errors  
1.2.1 Types of Faults  
1.2.2 Fault Masking  
1.2.3 Types of Errors  
1.3 CMOS Reliability: An Architectural Perspective  
1.3.1 Discussion of Design Options For Fault Tolerance  
1.3.2 Requirements of Architectural Reliability Solutions  
1.4 Contributions of This Thesis  
1.5 Thesis Organisation  

2 Fault-Tolerant Microarchitectures  
2.1 Error Detection Through Redundant Execution  
2.1.1 Types of Redundant Execution  
2.1.2 Input Replication  
2.1.3 Output Comparison  
2.1.4 Fault Isolation  
2.1.5 Execution Assistance  
2.2 Survey of Proposals for Redundant Execution  
2.2.1 Fault Tolerance Through Redundant Multithreading  
2.2.2 Redundant Execution in Chip Multiprocessors  
2.2.3 Replicate-at-Dispatch Mechanisms in Superscalar Processors  
2.2.4 Performance Improvement Through Redundant Execution  
2.2.5 Discussion  

v
CONTENTS

2.3 Symptom-Based Error Detection ........................................... 40
  2.3.1 Introduction ....................................................... 40
  2.3.2 Mode of Operation ............................................... 40
  2.3.3 Discussion ....................................................... 41
2.4 Circuit Level Mechanisms ................................................ 42
  2.4.1 Discussion ....................................................... 43
2.5 Lifetime Reliability Management ....................................... 44
2.6 Concluding Remarks ...................................................... 45

3 Energy-Efficient Redundant Execution 46
  3.1 Introduction .......................................................... 46
    3.1.1 Overview ....................................................... 47
  3.2 Redundant Execution using Simple Execution Assistance ........... 50
    3.2.1 Microarchitectural Support .................................. 50
    3.2.2 Fault Detection ............................................... 52
    3.2.3 Fault Isolation ............................................... 53
    3.2.4 Voltage and Frequency Control ............................... 54
    3.2.5 Fault Coverage ............................................... 55
    3.2.6 Evaluation .................................................... 56
    3.2.7 Discussion .................................................... 60
  3.3 Redundant Execution using Critical Value Forwarding ............. 60
    3.3.1 Core Architecture ............................................ 61
    3.3.2 Operation of the Leading Core ................................ 63
    3.3.3 Operation of the Trailing Core ............................... 64
    3.3.4 Options for Input Replication ................................ 65
    3.3.5 Effectiveness of Heuristics for Critical Value Identification ..................................................... 65
    3.3.6 DVFS in the Trailing Core ................................... 67
    3.3.7 Fault Detection ............................................... 68
    3.3.8 Fault Isolation ............................................... 68
    3.3.9 Parallel Application Support .................................. 69
    3.3.10 Fault Coverage ................................................ 72
    3.3.11 Evaluation .................................................... 73
    3.3.12 Discussion .................................................... 88
  3.4 Using Cores with Faulty Functional Units ............................ 89
    3.4.1 Results ......................................................... 90
  3.5 Limitations ......................................................... 91
  3.6 Concluding Remarks .................................................. 91

4 Multiplexed Redundant Execution 94
  4.1 Introduction .......................................................... 94
  4.2 Conceptual Overview .................................................. 95
  4.3 Execution Assistance in MRE ........................................ 96
  4.4 Design of an MRE Core ............................................... 97
    4.4.1 MRE+SEA Core ................................................ 97
## CONTENTS

4.4.2 MRE+CVF Core ................................................. 98
4.4.3 Run Request Queue ........................................ 98
4.5 Fault Tolerance Mechanisms .................................. 99
4.6 Evaluation ....................................................... 99
  4.6.1 Workload Construction ................................... 100
  4.6.2 Evaluation Metrics ........................................ 101
  4.6.3 Methodology ............................................... 102
  4.6.4 Results: Weighted Speedup .............................. 103
  4.6.5 Results: Normalised Throughput Per Core ............ 104
  4.6.6 Sensitivity to Queue Sizes .............................. 105
  4.6.7 Sensitivity to Execution Chunk Size .................. 106
  4.6.8 Priority-Based Scheduling Algorithm .................. 106
4.7 Concluding Remarks ........................................... 107

5 Conclusions and Future Work .................................. 109
  5.1 Conclusions .................................................. 109
  5.2 Future Work .................................................. 110
    5.2.1 Energy-efficient Timing Speculation .................. 110
    5.2.2 Improved Mechanisms for Identifying Critical Instructions .... 111
    5.2.3 Adaptive Multiplexing Schemes ....................... 112
List of Tables

2.1 Mechanisms for Execution Assistance .................................................. 31
3.1 CMP configuration for SEA evaluation ................................................. 56
3.2 Voltage-frequency levels for per-core DVFS ....................................... 57
3.3 Critical Value Identification Heuristics ................................................. 62
3.4 CMP configuration for CVF evaluation ............................................... 73
3.5 Percentage of loads not fully re-executed in the trailing core due to PLR 78
3.6 Comparison of best and worst normalised IPC values for each benchmark 85
across different thresholds. ............................................................... 85
3.7 Comparison of best and worst normalised energy values for each bench- 87
mark across different thresholds. ...................................................... 87
3.8 Voltage-frequency Levels of Trailing Core For Evaluation of Performance 88
with Within-die Variation. Note that the leading core is operated at full 
frequency (3.0 GHz). ................................................................. 88
4.1 Classification of benchmarks by speedup ........................................... 100
4.2 Multiprogram workloads ................................................................. 100
4.3 CMP configuration for MRE evaluation ............................................. 102
# List of Figures

1.1 The bathtub curve .................................................. 3  
1.2 Interaction of alpha particles and neutrons with a MOSFET [59] .......... 6  
1.3 Fault masking mechanisms ........................................ 9  
1.4 Logical masking ....................................................... 10  
1.5 Classification of possible outcomes of a faulty bit [114] .................. 13  
1.6 Increased fault masking at higher levels of abstraction ................. 15  
   
2.1 Conceptual framework for redundant execution .......................... 21  
2.2 Execution assistance .................................................. 30  
2.3 Symptom based error detection in ReStore ................................ 41  
   
3.1 High-level block diagram of energy-efficient fault-tolerant CMP proposals 48  
3.2 Block diagram of core supporting RESEA ............................ 50  
3.3 Normalised IPC of the SPEC benchmarks for RESEA .................. 58  
3.4 Normalised Energy of the SPEC benchmarks for RESEA ............. 59  
3.5 Bandwidth requirements for RESEA .................................. 59  
3.6 Block diagram of core supporting RECVF ........................... 63  
3.7 Performance of critical value identification heuristics ............... 65  
3.8 Example demonstrating how RECVF avoids input incoherence .......... 70  
3.9 Normalised IPC for the Shared L2 Configuration ..................... 74  
3.10 Normalised energy for the Shared L2 Configuration .................. 75  
3.11 Component-wise breakdown of energy consumption ................... 76  
3.12 Normalised IPC and normalised energy for the SPLASH2 benchmarks .... 77  
3.13 Normalised IPC of the private L2 configuration .................... 79  
3.14 Normalised energy of the private L2 configuration .................. 80  
3.15 Bandwidth requirements for RECVF ................................ 81  
3.16 Impact of the higher-latency and coarse-grained DVFS ............... 81  
3.17 Impact of limited voltage scaling .................................. 82  
3.18 Impact of queue sizes .............................................. 83  
3.19 Impact of thresholds on the QSize algorithm’s performance .......... 84  
3.20 Comparison of Cycles Spent at Each Frequency Level For the Trailing Core 86  
3.21 Performance of RECVF with “slow” cores .......................... 89  
3.22 Normalised IPC and energy with faulty FP units in the trailing core .. 90  
   
4.1 Conceptual block diagram of MRE .................................. 95  
   

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Block diagram of an MRE processor core</td>
<td>97</td>
</tr>
<tr>
<td>4.3</td>
<td>Weighted speedup for MRE</td>
<td>103</td>
</tr>
<tr>
<td>4.4</td>
<td>Normalised throughput per core for MRE</td>
<td>104</td>
</tr>
<tr>
<td>4.5</td>
<td>Sensitivity of MRE+CVF to queue sizes</td>
<td>105</td>
</tr>
<tr>
<td>4.6</td>
<td>Sensitivity of MRE+CVF to execution chunk size</td>
<td>106</td>
</tr>
<tr>
<td>4.7</td>
<td>Comparison of baseline and priority-based scheduling algorithms</td>
<td>107</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Over the last three decades, continued scaling of silicon fabrication technology has permitted exponential increases in the transistor budgets of microprocessors. In the past, higher transistor counts were used to increase the performance of single processor cores. However, increasing complexity and power dissipation of these cores forced architects to turn to chip multiprocessors (CMPs), which deliver increased performance at manageable levels of power and complexity. While technology scaling is enabling the placement of billions of transistors on a single chip, it also poses unique challenges. Integrated circuits are now increasingly susceptible to soft errors [62, 86], wearout related permanent faults and process variations [10, 16, 21].

Decreasing CMOS reliability implies that reliability, along with performance and power is expected to become a first-order design constraint for future microprocessors [81]. High-availability systems, which were previously restricted to the domain of mainframe computers and specially designed fault-tolerant systems, may now become important for the commodity market as well. In fact, the high-availability server market is expected to grow faster than the general-purpose server market [38].

Current high-availability systems like the HP NonStop Advanced Architecture [13] and the IBM zSeries [27] are high-end systems that spare no expense to meet reliability targets. Although they provide excellent fault coverage, they impose a high cost of 100% duplication and more than 100% additional energy consumption. These high costs are unacceptable for fault-tolerant systems targeted at the commodity market. These
systems have different requirements and present different design challenges. Specifically, the commodity market requires low-cost and configurable fault tolerance.

In this thesis, we address the problem of enabling low-cost, efficient fault tolerance for future microprocessors. Our work targets two aspects of efficient fault tolerance. Firstly, we design architectures that deliver energy-efficient fault tolerance. Secondly, we design architectures that mitigate the throughput-loss (see §1.3.2) due to fault tolerance.

This chapter provides an introduction to challenges posed by CMOS technology scaling and discusses their effects from an architectural perspective. Section 1.1 introduces reliability problems inherent in future CMOS technologies. Section 1.2 discusses faults due to reliability issues, and examines how faults can result in errors. Section 1.3 provides an architectural perspective on reliability concerns. Section 1.4 describes the contributions of this thesis and section 1.5 concludes.

1.1 Challenges of Technology Scaling

Reliability concerns for the future technologies can arise in the form of permanent, transient and intermittent faults. Among these, permanent faults can be classified as either extrinsic or intrinsic [59]. Extrinsic faults are caused by manufacturing defects and result in infant mortality. Typically, these are eliminated by a burn-in process. In contrast, intrinsic faults arise from degradation processes that result in the wearout of silicon chips.

Figure 1.1 shows the “bathtub curve” that depicts failure modes and their variation with time. Initially, intrinsic faults due to manufacturing defects result in a high failure rate. Burn-in eliminates faulty chips during this stage. The next stage, represented by the flat portion of the curve, is the useful lifetime of a chip. During this stage, failures are mainly due to soft errors or radiation induced transient faults. Finally, near the end of a chip’s lifetime, wearout mechanisms cause an increase in the failure rate.

The dashed gray curve in the figure shows how technology scaling affects the bathtub curve. Infant mortality is expected to become more prominent, resulting in a need for

---

1These fault types are defined precisely in §1.2.1.
Figure 1.1: The bathtub curve

longer burn-in processes. Soft error rates are expected to increase and the onset of wearout is expected to occur earlier. The net result of these effects is a decrease in the useful lifetime of chips, as well as a higher failure rate during the useful lifetime.

In the rest of this section, we describe the mechanisms that result in decreased reliability for future technologies.

1.1.1 Wearout Failure Modes in Future Technologies

This section presents a brief overview of wearout mechanisms in future chips that affect lifetime reliability. These mechanisms result in a degradation of chip performance with time, and affect either wires (electromigration) or transistors (hot carrier injection and negative bias temperature instability). Although wearout mechanisms eventually result in permanent faults, they can also cause intermittent delay faults before the onset of permanent faults.

Electromigration

Electromigration [3, 21] (EM) is an undesirable consequence of driving current through wires. As electrons move through the wire, they collide with the metal ions, imparting momentum to them. This causes the metal atoms to deplete from certain regions and accumulate in other regions. Depletion and accumulation of material results in voids and hillocks which eventually lead to open and short faults respectively. Voids can result in
intermittent delay faults due to increased resistance even before opens occur. EM is the most important source of failures for wires.

Current approaches for tackling electromigration have the following undesirable consequences:

1. **Wider wires.** EM can be mitigated using wider wires because wider wires have lower current densities. However wider wires require more area and also result in reduced interconnect densities. Note that wider wires do not prevent EM, but only delay it.

2. **Impact on cycle time.** EM is tolerated in current designs by adding guardbands to the cycle time. Guardbanding allows circuits to continue to work even when the wires are somewhat degraded.

**Hot Carrier Injection**

Hot Carrier Injection (HCI) [59, 104] is a CMOS transistor wearout mechanism that results in a degradation of the maximum operating frequency of silicon chips. In HCI, electrons accelerated by the channel’s electric field collide with the gate oxide interface creating electron-hole pairs. Some of these electrons are trapped in the gate oxide layer, increasing the threshold voltage. This slows down transistors.

The impact of HCI is directly proportional to switching frequency. HCI worsens with lower temperature because the mobility of carriers increases with a decrease in temperature. Current chips tolerate HCI by frequency guardbanding.

**Negative Bias Temperature Instability**

Negative Bias Temperature Instability (NBTI) [8, 106] is a transistor aging phenomenon that increases threshold voltage and reduces carrier mobility. Unlike HCI, which affects both NMOS and PMOS devices, NBTI affects only PMOS devices.

NBTI arises when a negative voltage is applied between the gate and source terminals of PMOS device. The holes introduced by the negative-bias are absorbed by the $Si - H$ bonds, weakening them so that they are easily broken under thermal excitation. When
the bonds are broken, $H$ diffuses away leaving positive interface traps ($Si^+$), causing an increase in the threshold voltage. This results in slower transistors.

A unique feature of NBTI is the recovery phase. When the reverse bias is removed, the $H$ diffuses back and anneals the broken $Si-H$ bonds. As a result, the number of interface traps is reduced, and threshold voltage degradation due to NBTI is recovered.

NBTI degradation is exponentially dependent on temperature.

The current solution to NBTI is frequency and voltage guardbanding. To tolerate NBTI degradation, $V_{min}$ is increased by about 10%, and $F_{max}$ is reduced between 10-20% [4]. Combating NBTI are an active area of research [4, 47, 70, 72, 87, 96, 104, 106, 112, 113, 117]. In particular, architectural solutions have been proposed that either attempt to limit the reverse bias on PMOS devices [4], manage resource usage based on on-chip temperature [104], or equalise activity factors across functional units [87].

1.1.2 Radiation Induced Transient Faults

The two main sources of radiation induced transient faults are alpha particles and neutrons [59]. In this section we describe the origin of these particles, how they result in transient faults, and discuss the impact of technology scaling on the soft error rate.

**Alpha Particles**

An alpha particle is identical to a Helium nucleus and consists of two protons and two neutrons. They are emitted by radioactive nuclei such as uranium and radium. They arise from radioactive impurities in chip packaging materials.

**Neutrons**

Neutrons that induce transient faults originate from cosmic rays. Cosmic rays are believed to arise from supernova explosions, stellar flares, the sun and other cosmic activities. The cosmic rays that bombard the earth’s outer atmosphere are called primary cosmic rays. Most of these rays hit atmospheric atoms and create a shower of secondary particles, known as secondary cosmic rays. The particles that ultimately reach the earth’s
surface are called terrestrial cosmic rays. Computers used in space encounter primary and secondary cosmic rays. Computers used on earth encounter only terrestrial cosmic rays.

To the first order, the Soft Error Rate (SER) due to cosmic rays experienced by a given CMOS circuit is proportional to the neutron flux [59]. The neutron flux varies with latitude, longitude and altitude. Flux is lowest at sea level and increases with altitude until a height of approximately 15 km. This point is known as the Pfotzer point. Beyond the Pfotzer point, neutron flux decreases again because the atmosphere at these altitudes is thinner.

Interaction of Alpha Particles and Neutrons with Silicon

When an alpha particle penetrates a silicon crystal, it causes strong field perturbations resulting in the creation of electron-hole pairs [59]. If the electric field of the p-n junction is sufficiently strong, it can prevent the electron-hole pairs from recombining. These excess carriers may be swept into the diffusion regions and eventually to the device contacts, resulting in an incorrect signal. Figure 1.2 depicts this event.

Neutrons, on the other hand, interact with silicon through inelastic collisions. Inelastic collisions cause the incoming neutrons to lose their identity and create secondary particles. Tang [102] gives the example of the following inelastic collision.

$$n + ^{28}Si \rightarrow ^4He + ^{25}Mg^*$$
$^{25}\text{Mg}^*$ is an excited compound nucleus which de-excites as:

$$^{25}\text{Mg}^* \rightarrow n + 3\, ^4\text{He} + ^{12}\text{C}$$

The collision results in the creation of four alpha particles and one residual nucleus ($^{12}\text{C}$). These particles interact with silicon and produce electron-hole pairs which may eventually result in transient faults as described earlier.

**Critical Charge**

The minimum amount of charge that has to be accumulated due to a particle strike in order to cause a bit flip is called the **critical charge**. Critical charge is a property of a specific node of a circuit. Typically, it is estimated using circuit simulation models by repeatedly injecting current pulses into the node until the circuit malfunctions.

**Impact of Technology Scaling on SER**

According to the model proposed by Hazucha and Svensson [35], circuit SER is directly proportional to area and particle flux, but decreases exponentially with critical charge. Transistor size decreases with each succeeding technology generation. This reduces the probability that a particle strike will affect a given device. However, critical charge also decreases due to a decrease in voltage. Consequently, the SER of a given circuit is expected to remain roughly constant with technology scaling [21, 59]. However, CMOS scaling doubles the number of transistors on a chip with each succeeding technology generation. This results in an exponential increase in the SER per chip due to technology scaling.

**1.2 Faults and Errors**

A **fault** is a deviation from correct behaviour of a device, circuit, architectural model or program. A fault results in an **error** only if there is a user-visible deviation from correct
behaviour. All faults may not result in errors, but an error must necessarily be the result of one or more faults.

Fault masking is the reason why a fault may not necessarily result in a user-visible error. Fault masking is discussed in detail in §1.2.2.

1.2.1 Types of Faults

One classification of faults divides them into three categories based on their temporal extent: permanent, intermittent and transient faults. Permanent faults, or hard errors, reflect irreversible physical processes and remain in the system until corrective action is taken. Intermittent faults appear and disappear by themselves and have unpredictable duration. In many cases, physical processes that eventually result in permanent faults initially cause intermittent faults. Transient faults, or soft errors, appear for a very short period of time, and disappear by themselves.

Differences between Transient and Intermittent Faults

The effects of intermittent and transient faults may seem similar, but there are three differences between the two.

1. Intermittent faults occur at the same location, while transient faults may occur anywhere in the circuit.

2. Intermittent faults tend to occur in bursts once they are activated. In contrast, a burst of transient faults is very unlikely.

3. Intermittent faults can be eliminated by replacement of the offending circuit. Transient faults cannot be eliminated in this way.

1.2.2 Fault Masking

A fault may not result in a user-visible error if its result is masked. A number of studies [15, 22, 61, 111] have shown that a majority of the faults that occur in microprocessors
Figure 1.3: Fault masking mechanisms

are masked and do not result in errors. In this section, we describe these masking mechanisms and discuss relevant studies.

Complex digital circuits like microprocessors can be considered at different levels of abstractions. At the lowest level of abstraction, the **device level**, we view the microprocessor as a collection of silicon devices that combine to form a circuit. At the next higher level of abstraction, the **circuit level**, the microprocessor is considered to be a collection of circuits that implement logic gates and storage devices. Moving further up the abstraction stack, at the **architectural level**, the microprocessor can be viewed as a collection of circuits implementing architectural and microarchitectural structures like ALUs, caches and branch predictors. The highest level of abstraction, the **application level**, we view the microprocessor as executing a program through the implementation of an instruction-set architecture.

Fault masking occurs at the *interface between any adjacent two abstraction levels*. Figure 1.3 shows many fault masking mechanisms and locations where they occur. As the figure shows, all faults originate at the device level. However, many are masked before becoming visible to the application.
Logical, Temporal and Electrical Masking

Logical masking occurs when the input to a logic gate is faulty but does not affect the output because the erroneous input is a non-controlling input. Therefore, the gate produces the correct output even in the presence of a faulty input. Figure 1.4 shows an example of logical masking. An AND gate with two inputs is shown. The fault-free input is 0, while the faulty input is changed due to the fault from 0 to 1. However, despite the fault, the output of the gate is correct as the fault has been logically masked.

Temporal masking (or latching-window masking) occurs when a fault propagates to the input of a state element such as a flip-flop or a latch, but does not arrive within the capture window of the state element [15].

Electrical masking occurs when a transient pulse is attenuated by subsequent logic gates so that the pulse does not affect the output of the circuit. [15]

A study by Blome et al. [15] found that about 6% of errors injected in registers were logically masked before propagating to microarchitectural state. They also found that more than 78% of errors injected in logic were logically masked before propagating to microarchitectural state. When temporal masking was taken into account, more than 83% of errors injected in logic were masked before the errors could propagate to the microarchitectural state.
Chapter 1. Introduction

Architectural Masking

Masking can also occur at the interface between the circuit and architectural levels of abstraction. The canonical example of this is a faulty branch predictor. A faulty branch predictor will not result in incorrect execution because of the branch misprediction recovery mechanism. We say faults in branch predictor are *architecturally masked* by the branch misprediction recovery mechanism.

Even structures that are essential for correct execution have fault masking capabilities because of the existence of dead, invalid or unused state. For instance consider a 128-entry reorder buffer (ROB) with 80 occupied entries. A fault which flips a bit in one of the unused 48 entries will not affect program correctness. Even in an entry that is being used, not all bits are required for correct execution. For example, a fault that flips a bit in the source operand field for an NOP instruction will not affect correct execution.

A study by Wang et al. [111] injected bit-flips in randomly chosen flip-flops and latches of an out-of-order superscalar microprocessor and studied how many bit-flips propagated to the architectural state. They found that more than 85% of injected faults were masked from software. A related study by Mukherjee et al. [61] found that only 14% to 47% of bits in the instruction queue were necessary for *architecturally correct execution*, indicating that this particular structure has masking rates varying between 53% and 86%.

Instruction-Level Derating

Cook and Zilles [22] define instruction-level derating as mechanisms through which a computation on incorrect values produces the correct result. Their study injected faults into the architectural state of a program and measured how many fault injections resulted in incorrect outcomes.

The architectural state of the program encompasses the register state, the memory state and the program counter. Incorrect execution was defined by Cook and Zilles as the occurrence of one of the following:
1. control flow diverges from fault-free execution
2. store address or value different from fault-free execution
3. load address results in fault
4. system call input diverges from fault-free inputs

They found significant masking effects even at the architectural level; 36% of injected faults (i.e., architecturally visible faults) did not result in incorrect output due to instruction-level derating. Their study classified instruction-level derating mechanisms into the following categories.

1. **Value comparison.** This derating mechanism is due to the fact that an injected faulty value is used only in a comparison operation, which means that significant information is discarded. As a result, even an incorrect input to the comparison operation results in the correct outcome.

2. **Subword operations.** This derating mechanism is due to operations that use only a subset of the bits in the incoming values.

3. **Logical operations.** This derating mechanism is because of operations that perform a logical AND with ‘0’ bits or logical OR with ‘1’ bits.

4. **Overflow/Precision.** This is a result of the faulty bit being unused or irrelevant to the computed result because of overflow or precision effects.

5. **Lucky loads.** This class of derating is caused by loads that read from an incorrect address but obtain the correct value. In the majority of the cases, this occurs because common values like zero are loaded.

6. **Dynamically dead values.** This mechanism accounts for values that are computed but not used. Typically, they are a result of compiler optimizations which compute a value for a path that ends up not being taken; for instance load instructions that are hoisted above a branch.
Error-Resilient Applications

Masking is also possible at the application level. Typically, this occurs in applications that operate on noisy data and use algorithms that are inherently error-resilient. Examples of these are media applications such as sound playback and picture decoding. In both these examples an error that limits itself to a few incorrect sound samples or pixels cannot be perceived humans.

Exploiting the properties of error-resilient applications for low-cost fault tolerance is an area of active research [24, 89].

1.2.3 Types of Errors

Depending on whether error detection and/or correction circuitry is incorporated in a microprocessor, a fault can have a number of different outcomes. Figure 1.5 depicts six possible outcomes due to a faulty bit.

If the faulty bit is never read, then the fault is masked and no error occurs. On the other hand, if the faulty bit is read, but has error detection and correction, then the fault is corrected and there is no user-visible error. These two cases correspond to the outcomes (1) and (2) in Figure 1.5 respectively.

If the faulty bit is read, but only error detection is incorporated in the system, then the system records a Detected Unrecoverable Error (DUE). In the case of
DUE, the system avoids generating incorrect outputs and thereby any data corruption. However, no error recovery is possible. Depending on whether the error would have affected program outcome, there are two possibilities. If the error would not have affected program outcome, then a DUE was unnecessarily signalled. This is called a false DUE event. If the error would have affected program outcome, then it is called a true DUE event. These two cases correspond to the outcomes (3) and (4) in Figure 1.5.

DUE events can also be divided into system-kill and process-kill DUE events. In the cases when the error is restricted to a set of processes, then the system can continue operation by killing the set of processes affected by the error. Such an event is called a process-kill DUE event. An error which requires a restart of the entire system is called a system-kill DUE event.

If there is no error detection or correction, again there are two possible outcomes. If the error does not affect program outcome, then the system continues to operate correctly. If the error affects program outcome, then it is called Silent Data Corruption (SDC). These two cases correspond to the outcomes (5) and (6) in Figure 1.5.

Typically system vendors specify SDC and DUE targets for their chips. For instance, the IBM Power4 processor has a 1000 year SDC MTTF (mean time to failure), 25 year system-kill DUE MTTF and 10 year process-kill DUE MTTF [18].

1.3 CMOS Reliability: An Architectural Perspective

Fault masking has important consequences in the design of reliable systems.

As discussed in §1.2.2, most faults are masked before they result in user-visible errors. Faults originate at the device level, but the majority of these faults are masked before they become visible at the circuit level [15]. Even those faults which may not be masked at the circuit level might be masked by architectural masking mechanisms [61, 111]. Furthermore, architecturally visible faults may not necessarily result in incorrect program outcome due to instruction-level derating [22].

As Figure 1.6 shows, due to fault masking effects, a reliability mechanism that detects faults at a lower level of abstraction will observe more faults than one that detects
faults at a higher level of abstraction. Many of the faults detected at a lower level of abstraction are likely to be masked by mechanisms that operate at higher levels. For instance, a circuit level fault detection mechanism will detect many more faults than an architectural fault detection mechanism. Faults at the circuit level correspond to incorrect circuit outputs. As has been discussed previously, incorrect circuit outputs need not result in an architecturally visible fault. However, a mechanism that operates at the circuit level has no information about architectural fault masking mechanisms. Hence, it must detect (and possibly correct) a large fraction of faults that may not affect program output.

This line of thought suggests that fault detection and correction must be performed at as high a level of abstraction as possible. Architectural fault tolerance mechanisms are attractive in this respect because they can maintain software compatibility and achieve a reasonable number of false positive fault detections.

1.3.1 Discussion of Design Options For Fault Tolerance

One solution for fault tolerance is to propagate all faults to the application. Each application can then decide how each fault may be handled. In fact, this approach is advocated in some recent proposals for fault-tolerant CMPs like Relax [24] and the Stochastic Processor Project [63]. Such designs can provide an application-specific level
of fault tolerance tailored to each application’s needs. Thus, such mechanisms can take ignore errors that affect noisy data or results [24], or those that cannot be perceived by humans. However, this requires that application software be modified to be reliability-aware, which necessarily implies breaking software compatibility.

Architectural-level mechanisms have the advantage of maintaining software compatibility while resulting in only a small number of false-positive error detections. For this reason, this thesis focuses on mechanisms for architectural fault tolerance. However, cost-effective architectural mechanisms cannot detect all possible errors. For instance, many architectural mechanisms share lower-levels of the memory hierarchy like the L2 caches, the northbridge and the memory controller. Errors that affect these components have to be detected using circuit-level mechanisms like radiation hardening [36, 58].

Circuit-level mechanisms for fault tolerance have the advantage of not requiring any changes to architectural and system-level design [31]. Furthermore mechanisms like the use of error correcting codes (ECC) and parity are ideally suited for protecting data stored in array structures like caches and TLBs.

In summary, fault-tolerant CMPs of the future are likely to combine complementary architectural, circuit-level and application-level fault tolerance techniques to ensure reliable execution at acceptable performance and power overheads.

1.3.2 Requirements of Architectural Reliability Solutions

Traditionally, high availability systems have been restricted to the domain of mainframe computers or specially designed fault-tolerant systems [13, 27, 44]. However, the trend towards unreliable components means that fault tolerance is now important for the commodity market as well [6]. Fault-tolerant solutions for the commodity market have different requirements and present a different set of design challenges. The commodity market requires configurable [6] and low cost fault tolerance. Low cost here means the following.

1. Low performance overhead.
2. Reduced additional energy consumption.
3. Low area overhead.
4. Transparent to software.

**Importance of Energy-efficiency**

An important aspect of fault-tolerant CMP designs is their energy-efficiency. Power and peak temperature are key performance limiters for modern processors [41]. Since the power budget for a chip is fixed, decreasing the power consumed in any core increases the power available to other cores. This allows them to operate at a higher frequency, increasing overall system performance. Furthermore, reducing power dissipation has an additional advantage of reducing operating temperatures, which can increase chip lifetimes by an order of magnitude [69]. Reducing the energy overhead of fault tolerance schemes is also important from the point of view of data center energy. Data center energy consumption is expected to reach an unprecedented level of 100 billion kilowatt hours by 2011 [56]. Unreliable chip components would imply that a significant fraction of future data centers would require fault tolerance mechanisms to cope with hardware faults. Clearly, there is a pressing need for energy-efficient fault-tolerant architectures for future microprocessors.

**The Throughput Loss Problem**

Proposals for fault-tolerant CMPs typically use two or more cores or thread contexts of a CMP to execute a single logical thread [29, 30, 32, 46, 49, 60, 76, 77, 90, 91, 101, 108]. The use of multiple cores or thread contexts to execute a single program means that the throughput of the CMP is reduced by half. Due to this throughput loss, a fault-tolerant system must have twice as many cores as an equivalent non-redundant system to achieve the same throughput. Not only does this increase the procurement cost of systems, it also increases the running cost of the system due to increased cooling costs, energy costs and maintenance costs; these costs may be greater than the cost of purchasing the system. Such high costs are undesirable for fault-tolerant general-purpose microprocessors targeted at the commodity market. Therefore, there is a need
for mechanisms that can reduce the throughput loss due to fault tolerance.

1.4 Contributions of This Thesis

This thesis introduces two architectures that tackle the problem of enabling low-cost fault tolerance for future microprocessors. The first architecture provides energy-efficient redundancy by reducing the energy overhead of fault tolerance. Our results show that this architecture has a performance overhead of 1.2% and energy consumption that is only 1.26 times that of non-redundant execution. The second architecture is called multiplexed redundant execution (MRE) and increases the performance of fault-tolerant systems by mitigating the throughput loss. We find that throughput increases of 9-13% are possible over existing proposals for fault-tolerant chip multiprocessors (CMPs).

Both these architectures utilise the idea of critical value forwarding, a new form of execution assistance [82] for leader-follower architectures [9, 77, 82]. Instructions on the critical path of execution are identified and the results of these are forwarded to the trailing core. By forwarding critical values, we obtain much of the speedup that can be obtained by forwarding all values, but use only a fraction of the bandwidth. Reducing the bandwidth cost of execution assistance mechanisms is important as interconnects are expected to become bottlenecks in future CMPs [45].

1.5 Thesis Organisation

This chapter gave an introduction to the problem of decreasing CMOS reliability. We discussed wearout mechanisms that are important for future CMOS technology generations. We described the effects of radiation induced particle strikes and discussed how technology scaling is resulting in an exponential increase in the SER of future microprocessors. We discussed the difference between faults and errors, the phenomenon of fault masking and mechanisms which cause it. We argued that architectural fault tolerance is an attractive option because of software compatibility and fault masking. Finally, we described the contributions of this thesis and the motivation for these contributions.
Chapter 1. Introduction

The rest of this thesis is organised as follows. Chapter 2 surveys related work in the area of fault-tolerant architectures and circuit designs. Chapter 3 describes our proposals for energy-efficient fault tolerance. Chapter 4 describes our proposals that tackle the throughput loss problem. Finally 5 concludes with some suggestions for future work.
Chapter 2

Fault-Tolerant Microarchitectures

This chapter presents an overview of related work in the area of fault-tolerant architectures and circuit designs.

There have been two different approaches to detect the occurrence of faults in modern chip multiprocessors (CMPs). One approach is based on redundant execution. This technique performs multiple executions of the same program and compares the outputs of each to detect errors. Traditional fault-tolerant systems like the NonStop® Advanced Architecture [13] and the IBM zSeries processors [27] are based on redundant execution. Although these architectures provide excellent fault coverage, they impose more than 100% extra area and power costs. A number of recent proposals have attempted to reduce these costs [7, 29, 30, 32, 46, 49, 60, 64, 68, 73, 75–77, 91, 101, 107, 108]. Techniques for redundant execution are described in detail in sections 2.1 and 2.2 of this chapter.

An orthogonal approach to detecting faults is symptom-based fault detection [74, 110]. Symptom-based fault detection deduces the occurrence of a fault by monitoring program behaviour and microarchitectural variables. These mechanisms have reduced costs compared to redundant execution, but cannot guarantee the detection of all faults. Hence, this thesis focuses on mechanisms based on redundant execution. An overview of symptom-based fault detection is given in section 2.3.

Circuit-level approaches to fault tolerance have been advocated in a number of proposals [11, 26, 36, 55, 58, 95]. Section 2.4 describes some of these mechanisms, while
section 2.4.1 discusses the trade-offs between architectural and circuit-level mechanisms.

Section 2.5 briefly describes the important problem of lifetime reliability management in future microprocessors. Section 2.6 concludes.

2.1 Error Detection Through Redundant Execution

Redundant execution is well-studied technique that detects errors in a system by executing the same program multiple times. The results of these multiple executions are compared to detect errors.

Figure 2.1 shows a conceptual framework for understanding error detection methods based on redundant execution. There are three important components of a system for redundant execution.

1. **Multiple streams of execution** are used to detect errors. Each stream executes the same program with the same input, and should produce the same output in the absence of errors. Occurrence of an error causes the outputs of the streams to diverge, enabling detection. In a typical hardware redundancy scheme, each stream of execution is mapped to thread contexts or cores of multicore processor.
2. **Input replication** is the process which ensures each stream of execution receives the same input. Mechanisms for input replication are explained in detail in §2.1.2.

3. **Output comparison** is the mechanism that compares the outputs of each stream of execution to detect errors. Mechanisms for output comparison in the context of CMPs are discussed in §2.1.3.

Redundant execution can only detect errors that lie within the **sphere of replication** [77]. Components inside the sphere of replication enjoy fault coverage due to redundant execution, while components outside the sphere do not. For instance, errors in the mechanisms for input replication or output comparison cannot be detected because these fall outside the sphere of replication.

### 2.1.1 Types of Redundant Execution

Redundant execution may be classified into two types based on where the multiple streams are executed.

**Space Redundancy**

Spatial redundant execution or space redundancy refers to mechanisms that execute the same program on replicated hardware [77]. Since each execution occurs on different hardware devices, this type of redundancy can detect transient as well as permanent faults.

Traditional fault-tolerant architectures like the NonStop® Advanced Architecture [13] use space redundancy; the same program on is executed on multiple processors and the outputs of these processors are compared to detect errors.

**Time Redundancy**

Temporal redundant execution or time redundancy refers mechanisms that execute a program on the same hardware multiple times. This type of redundancy can detect only...
transient faults, since permanent faults may affect all the executions, resulting in all of them producing the same erroneous outcome.

For example, one set of compiler techniques for fault tolerance emit redundant copies of instructions and compare the results of the two versions to detect errors [78, 79]. Both original and redundant copies of an instruction execute on the same hardware at different times; hence, these techniques are based on time redundancy.

2.1.2 Input Replication

Input replication is the mechanism which ensures that each stream of execution in a system for redundant execution receives the same input.

Tight Lockstepped Execution

Traditional fault-tolerant systems [12, 27] were based on the principle of tight lockstepped execution. In such systems, the two processors or pipelines use the same clock and execute the same instructions every cycle. The outputs are compared after each operation to detect errors. Input replication in a tight lockstepped system is trivial because all that needs to be done is for each primary input to a core to be wired to corresponding redundant core.

Unfortunately, tightly lockstepped microprocessors are no longer practical for the following reasons [7, 13, 91]:

1. Low-level recovery mechanisms such as on-chip ECC protected caches may complicate lockstepped operation. For instance, ECC mechanisms can cause two lockstepped processors to fall out of sync if one the processors has to retry a register read due to an ECC-correctable fault.

2. All system-level components must be designed with lockstepped operation in mind and must synchronise with each other. For instance, memory controllers in current commodity processors operate asynchronously and do not synchronise with each other.
3. Power management techniques such as dynamic voltage frequency scaling (DVFS) cannot be used. In current processors synchronised frequency shifting that is accurate to one cycle is extremely difficult even if both processors are on the same chip.

4. Tight lockstep requires identical initialisation and determinism across cores, including in units that do not affect architecturally correct execution like branch predictors and prefetchers.

5. Pipeline level lockstepped comparisons require multiple cycles to propagate values from one pipeline to the other. This requires additional latches, which have significant cost [54]. Furthermore, instructions have to be buffered until the comparison succeeds, which can lead to a significant performance cost.

6. Increasing within-die device and circuit level variability also complicates lockstep because different cores may no longer have identical timing properties or execution resources.

**Loose Lockstepped Execution**

For the reasons outlined in the previous subsection, current high-availability systems are moving towards loose lockstepped execution [13]. In loose lockstepped execution, the two cores execute the same program at approximately the same time, but are not synchronised on a cycle-by-cycle basis.

However, loose lockstepped introduces an input replication problem. As there is a delay between the time a given instance of a dynamic load instruction is executed by the leading core and the time it is executed by the trailing core, the cores may not read the same value due to the occurrence of data races. This problem is called the **input incoherence problem** [91]. As a consequence of the input incoherence problem, any mechanism for loose lockstepped redundancy in CMPs must incorporate a mechanism that ensures that both cores receive the same input even in the presence of data races. In other words, *inputs must be precisely replicated.*
Mechanisms for Input Replication in Loose Lockstepped Processors

This section describes two proposals for precise input replication in loose lockstepped fault-tolerant processors.

**Replication of All Load Values:** The idea of replicating all load values from the leading to the trailing thread was introduced by Mukherjee and Reinhardt in the context of Simultaneously and Redundantly Threaded (SRT) processors [77]. In SRT processors, only the leading thread accesses the memory hierarchy. The trailing thread uses the value produced by the leading thread. Values are passed from the leading to the trailing thread through the use of the load value queue (LVQ) structure. A load value read by the leading thread is enqueued in the LVQ at the time of retirement of the load instruction. When the trailing thread executes the corresponding load instruction, it accesses the LVQ instead of the data cache and obtains the same value.

In an out-of-order superscalar processor, load instructions in the leading and trailing thread may not be executed in the same order. This creates the need for a mechanism that identifies which load value corresponds to which instruction in the trailing thread. Three proposals have been made that solve this problem.

1. In [77] Mukherjee and Reinhardt advocate executing load instructions in program order in the trailing thread. Since load values are enqueued at the time of instruction retirement in the leading thread, which also occurs in program order, the trailing thread can use the values in the order in which they were enqueued.

2. In [60], a different solution is proposed. A tag is associated with each load value at the time it is enqueued in the leading thread. This tag is passed on to the trailing thread’s corresponding load instruction and used to lookup the correct entry in the LVQ.

3. The third proposal, by Gomaa et al. [30] uses the observation that there is a unique mapping between instructions decoded in the trailing core and instructions retiring in the leading core. Therefore, information about the order in which instructions are decoded in the trailing core is sufficient to locate the correct LVQ.
entry corresponding to a load instruction.

**Reunion:** Smolens et al. [91] introduced Reunion which provides an elegant mechanism for dealing with input incoherence in CMPs. The intuition behind Reunion is that input incoherence is a rare event and can be handled using the same mechanisms that enable soft error recovery.

In the Reunion CMP architecture, each logical thread is executed on two cores of a CMP, a **vocal core** and a **mute core**. Coherence requests made by the two types of cores are handled differently. Requests made by the mute cores are called **phantom requests** and return a value for the block without changing the coherence state of the system. Requests from a vocal core are handled in the normal fashion.

Smolens et al. make the observation that in the common case, both cores read the same value for a given dynamic load instruction. In other words, *input incoherence is a rare event*. Therefore, in the rare cases when input incoherence does occur, Reunion uses a rollback recovery based protocol for re-execution, this time ensuring that inputs to both cores are the same.

The Reunion re-execution protocol is invoked when the architectural states of the vocal and mute cores diverge. It involves the following steps.

1. Both vocal and mute cores are initialised from the **safe state** obtained from a previously verified checkpoint.

2. The logical processor pair now executes each instruction non-speculatively, **in single-step mode**, up to and including the first load or atomic memory operation.

3. This operation is issued by both the cores using a **synchronising memory request**, which is a special coherence request that ensures that both the cores receive the same value in response.

The Reunion protocol ensures forward progress. To see why, note that rollback recovery is triggered only when the architectural states of the vocal and mute cores diverge. This can be due to a soft error or input incoherence. If it is due to a soft error,
then the error does not recur on re-execution. As a result, the architectural states will match at the next comparison, ensuring forward progress. If rollback recovery is triggered due to an instance of input incoherence, then the incoherence is eliminated by the synchronising request. Therefore, in this case too, forward progress is guaranteed.

2.1.3 Output Comparison

A system for redundant execution needs a mechanism for output comparison. As the name suggests, output comparison involves comparing the results of each stream of execution in order to detect errors. There are number of design choices for output comparison and each choice has different trade-offs.

How often outputs comparison is performed directly impacts error detection latency. Error detection latency, in turn, impacts how often checkpoints have to be taken. If the detection latency is large and the checkpointing interval is small, corrupted checkpoints will be stored. Recovery from corrupted checkpoints is useless because the error will recur on re-execution. Unfortunately, the naïve approach to frequent output comparison requires inordinate comparison bandwidth.

The three main design choices for output comparison are the following:

- Outputs are compared at the level of architectural register updates. In this case, error detection latency is minimised. However, the downside of comparing each register update is increased comparison bandwidth. Smolens et al. call this option full-state comparison [90].

- Outputs may also be compared at the data cache interface (i.e., after each store instruction), an approach first advocated in SRT processors [77]. This reduces the bandwidth required for comparison significantly but has higher error detection latency than full-state comparison.

- Outputs may be compared at the chip-external pins. This has the lowest bandwidth requirements, but also has the highest error detection latency. This is the approach
adopted by traditional fault-tolerant systems such as the NonStop® Advanced Architecture [13].

Fingerprinting

Smolens et al. introduced the technique of fingerprinting which solves the latency-bandwidth trade-off problem [90].

The idea of fingerprinting is to compress the architectural updates, which include register updates, load/store addresses and values, and branch targets, using a CRC-based hash code to generate a fingerprint. The fingerprint is computed incrementally and updated every cycle. Periodically, the fingerprints generated by the two cores are compared. If the fingerprints match, then it is likely that no error has occurred. If the fingerprints do not match, this implies the occurrence of an error.

As fingerprints involve lossy compression, there is a risk of fingerprint aliasing. Fingerprint aliasing is said to occur when two different set of architectural updates result in the same fingerprint. For a fingerprint of width \( p \) bits, the probability of aliasing is \( 2^{-p} \). Smolens et al. find that for reasonable error rates and reliability targets, the probability of aliasing is minuscule and may be ignored.

Smolens et al. find that fingerprinting results in an order of magnitude increase in the mean time to failure (MTTF) of a system when compared to equivalent systems that perform full-state comparison or comparison at the chip-external pins. Furthermore, fingerprinting requires negligible bandwidth.

2.1.4 Fault Isolation

Fault isolation requires that between the time a fault occurs and the time it is detected, the effects of the fault must not allowed to spread outside the set of processors executing the program. In particular, unverified or potentially faulty data must not propagate to I/O devices or be written back from the data caches to main memory.

Mechanisms for redundant execution use a number of different techniques to ensure fault isolation. A brief overview of these mechanisms is given below:
• SRT/CRT \cite{60, 77} processors verify each store instruction before it is allowed to retire from the store buffer. Only when the result of a store instruction is known to be correct, is it allowed to write to the data cache or an I/O device.

• Proposals such as Reunion \cite{91} and DDMR \cite{29} force fingerprints to be compared before the effects of a store instruction are visible outside the core that executes it. This ensures that only verified store instructions are made visible to other cores and I/O devices.

• A third class of proposals such as that of Rashid et al. \cite{75, 76} introduce new microarchitectural structures that buffer unverified stores before they can propagate outside the processor. The Post Commit Buffer (PCB), a structure first introduced in \cite{76}, holds unverified stores that are retired from the store buffer until they are verified. Only after verification is a store allowed to retire from the PCB to the on-chip caches.

• A final class of proposals isolate unverified data in the L1 data cache. These techniques are based on speculative versioning caches \cite{48} and track unverified lines in the data caches. Unverified lines are not allowed to be written back to lower level caches. This is approach taken by DCC \cite{49}. Our proposals also use the same approach.

2.1.5 Execution Assistance

A number of architectures for redundant execution have been organized as leader-follower configurations. In such configurations, each logical thread is executed on two cores or thread contexts. One of these cores is designated as the leading core and the other is designated as the trailing core. The leading core assists execution of the trailing core by forwarding the results of its execution. These results are used as predictions in the trailing core. Figure 2.2 illustrates this.

Execution assistance is effective because results of execution of the leading core can generate information that can accelerate trailing core execution. For example, if an
execution assistance mechanism forwards *branch outcomes* from the leading core to the trailing core, these outcomes can be used as *perfect branch predictions* in the trailing core. Similarly, the leading core may bring values into a shared data cache, preventing the corresponding misses in the trailing core.

![Diagram of execution assistance](image)

**Figure 2.2: Execution assistance**

The idea of execution assistance was introduced by Rotenberg in AR-SMT [82]. AR-SMT forwarded branch outcomes and all values as predictions to the trailing thread. Variants of this idea were also explored in DIVA [9] and SRT [77]. A large number of subsequent proposals have used some form of execution assistance in the design of several interesting architectures. Table 2.1 presents a classification of some proposed mechanisms. The rest of this subsection discusses these design options.

**Forwarding All Values**

While forwarding all values provides the highest possible speedup in the trailing thread, it also requires inordinately high bandwidth. As a result it is mainly suited for use within the components of a single processor core, like in the case of AR-SMT and DIVA. Forwarding all values to a different core is likely to require adjacent placement of cores. This reduces scheduling flexibility. Note that AR-SMT and Slipstream assume the presence of value-prediction support in the baseline processor to detect errors in the forwarded values.
Values Forwarded | Proposals
---|---
All values | AR-SMT [82], DIVA [9], Slipstream¹ [101], Madan et. al [52].
Loads and branches | SRT [77], CRT [60], SRTR [108], CRTR [30], SpecIV [46], EERE [98], MRE [99].
Branches only | PVA² [76], Paceline [32], Decoupled performance correctness [28], Circuit pruning [57].
Critical values | This thesis.

¹Slipstream’s leader core forwards all values that it executes. However, it may execute a subset of program due to ineffectual instruction elision.
²PVA examines configurations with only branch forwarding as well branch forwarding combined with L1 prefetch hint forwarding.

Table 2.1: Mechanisms for Execution Assistance

Forwarding Loads and Branches

SRT [77] introduced the idea of forwarding only load values and branch outcomes from the leading to the trailing thread. This approach has been adopted in a large number of subsequent proposals [30, 46, 60, 97, 99, 108]. Forwarding branch outcomes eliminates branch mispredictions while forwarding load values has two beneficial effects. Firstly, it eliminates data cache misses in the trailing thread. Secondly, it solves the problem of input incoherence. However, load and branch instructions form more than one-third of the instruction mix of the SPEC CPU 2000 benchmarks. As such forwarding the results of these instructions requires considerable bandwidth. Furthermore, all these proposals suffer from reduced fault coverage because they do not fully re-execute load instructions in the trailing core.

Forwarding Only Branches

Forwarding only branch outcomes has also been studied in a few proposals. This eliminates branch mispredictions in the trailing core, significantly accelerating it. Since
branch/jump instructions form about one-tenth of the instruction mix of the SPEC CPU 2000 benchmark suite, forwarding all branch instructions puts moderate bandwidth pressure on the on-chip interconnects.

**Forwarding Critical Values**

In this thesis, we introduce a new form of execution assistance called critical value forwarding. The idea of critical value forwarding is to identify instructions on the critical path of execution and forward the results of these to the trailing core. We will show in chapter 3 that critical value forwarding provides most of the speedup of forwarding all values at a fraction of the bandwidth cost.

### 2.2 Survey of Proposals for Redundant Execution

This section provides an overview of existing proposals for fault tolerance based on redundant execution. We discuss, where appropriate, the pros and cons of each method.

#### 2.2.1 Fault Tolerance Through Redundant Multithreading

Transient fault detection using simultaneous multithreading was introduced by Rotenberg in AR-SMT [82] and Reinhardt and Mukherjee [77] in Simultaneously and Redundantly Threaded (SRT) processors. An SRT processor augments SMT processors with additional architectural structures like the branch outcome queue (BOQ) and load value queue (LVQ). The BOQ is used to pass branch outcomes from the leading to the trailing thread, providing perfect branch prediction to the trailing thread. The LVQ ensures precise input replication by forwarding load values from the leading to the trailing thread. The store buffer in an SRT processor is modified to perform output comparison. An entry is not retired from the store buffer until both the leading and trailing threads have executed the store instruction and produced the same address and value.

Since an SRT processor provides an unpredictable combination of space and time
redundancy, it cannot guarantee the detection of permanent faults. SRT has a performance overhead of about 20-30% compared to a non-redundant baseline, and consumes about 1.5-1.6X the energy of non-redundant execution.

Muhkerjee et al. also introduced chip-level redundant threading (CRT) [60], which extends SRT to simultaneously multithreaded chip multiprocessors. CRT executes the leading and trailing thread on different cores of multicore processor, permitting the detection of transient and permanent faults.

Both SRT and CRT only detect errors while ignoring the issue of fault recovery. SRT was augmented with recovery by SRTR [108], while CRTR [30] augmented a CRT processor to provide fault recovery. These proposals use the state of the trailing thread for recovery. CRTR introduced dead and dependence based checking elision (DDBCE) which reduces comparison bandwidth by comparing only instructions at the end of dependence chains and eliminating the comparison of dynamically dead instructions.

Modifications to SRT have attempted to reduce its performance overhead by trading-off fault coverage. Speculative Instruction Validation (SpecIV) [46] does this by not redundantly executing instructions whose results are the same as those produced by a value predictor. The intuition behind SpecIV is that if an instruction’s result matches the value produced by the predictor, then it is unlikely to have been affected by a soft error. SpecIV’s evaluation suggests that this simple idea reduces the performance overhead of SRT by half, at a negligible cost in terms of fault coverage.

A related proposal, SlicK [68], attempts to eliminate redundant execution of instructions that occur on the backward slices of verifiable computations. Examples of verifiable computation are high-confidence branches and store instructions whose values and address can be predicted. SlicK reduces the performance overhead of SRT from 20% to 10% at a negligible vulnerability cost.

2.2.2 Redundant Execution in Chip Multiprocessors

Increasing core [17, 40] counts on future chip multiprocessors can be exploited for redundant execution. In this section, we describe a few proposals that use different cores of a
multicore processor for redundant execution.

Todd Austin introduced DIVA\cite{9} which is a novel fault detection and correction architecture which uses an in-order checker processor to detect errors in a larger out-of-order superscalar processor core. The checker processor is fabricated using larger and more reliable transistors. While the DIVA idea itself is quite robust, some implementation details are not amenable to modern deep sub-micron technologies. DIVA uses larger transistors in the checker processor to reduce susceptibility to soft errors. However it cannot guarantee the detection of all soft errors, especially those that occur in the checker processor. Another disadvantage of DIVA is that resources and functional units allocated for re-execution are unavailable for normal execution (i.e., when redundant execution is not being performed).

Smolens et al. introduced Reunion\cite{91}. As was described in §2.1.2, Reunion solves the problem of precise input replication in multicore processors by reusing the soft error recovery mechanism to avoid input incoherence.

Rashid et al.\cite{76} introduced the parallelized verification architecture (PVA). PVA is based on the observation that fault detection using redundant execution can be viewed as a combination of two tasks: execution and verification. Furthermore, the verification task can be parallelized. PVA uses two cores to execute the verification task and operates these two cores at half voltage/frequency levels. Since voltage vs. frequency is nonlinear, this significantly reduces the energy overhead of redundant execution. However, this reduction in energy consumption comes at the cost of using three cores to execute a single program.

Aggarwal et al.\cite{6} studied the fault isolation characteristics of CMPs. They find that existing CMPs have many shared resources across cores, and hence, many single points of failure. They proposed configurable isolation, which allows a CMP to be configured in two modes. One mode is a high reliability mode where no resources (L1/L2 caches, memory controller etc.) are shared between cores. The other mode is a high performance mode where resources are shared but redundancy is disabled. Enabling configurable isolation with minimal additional hardware is attractive because the same processor can be used
by users with different reliability requirements by manipulating software-configurable settings.

Aggarwal et al. [7] also studied the problem of memory replication in traditional fault tolerant architectures. They introduce the idea of a duplication cache, which holds duplicate copies of dirty pages. Clean pages are not duplicated. Duplicating only dirty pages significantly reduces the memory pressure created by redundant execution.

A number of fault-tolerant CMP proposals assume a dedicated interconnect between the two cores performing redundant execution [30, 60, 76, 90, 91, 101]. This is a serious disadvantage because the failure of any one core renders both cores unusable for redundant execution. DDMR [29] and DCC [49] relax this assumption of a dedicated interconnect. DCC uses the system bus of a shared memory CMP to perform all communication. DDMR uses a ring interconnect between the processors of CMP.

2.2.3 Replicate-at-Dispatch Mechanisms in Superscalar Processors

Modern out-of-order superscalar processors have a significant amount of idle capacity [64]. Idle capacity refers to functional units, issue queue entries, ROB entries etc. which are rarely fully utilised. This idle capacity can be used to provide transient fault tolerance by using it to execute redundant copies of a program. Typically, these mechanisms fetch or decode redundant instructions, instead choosing to replicate instructions in the dispatch stage. As a consequence of this any faults affecting the fetch and decode stages cannot be detected by such mechanisms.

The replicate-at-dispatch idea was first introduced by Sohi, Franklin and Saluja in [93] in the context of pipelined processors with multiple functional units. Subsequently, Nickel and Somani studied replicate-at-dispatch mechanisms for out-of-order superscalar processors in REESE [64]. Qureshi et al. [73] introduced a related architecture, Microarchitecture Based Introspection (MBI). MBI executes redundant copies of instructions in the shadow of a L2 cache miss. For memory intensive benchmarks of the SPEC CPU 2000 suite, this mechanism provides fault tolerance at a small performance cost. Vera et al. introduced selective replication, a partial fault coverage mechanism that on re-executes
only the most vulnerable instructions [107].

2.2.4 Performance Improvement Through Redundant Execution

A number of proposals [28, 32, 57, 101] have attempted to use some form of redundant execution in order to improve CMP performance. These techniques execute a single logical thread on two cores on a CMP, as a leading and trailing thread. The leading thread uses a speculation mechanism to improve its performance. The trailing thread detects and recovers from misspeculation. Execution assistance helps the trailing thread keep up with the faster leading thread. In this section, we describe these mechanisms for CMP performance improvement based on such techniques.

Paceline [32] is a proposal for CMP performance improvement that operates the leading core at higher than its nominal frequency. In effect, the leading core performs timing speculation, while the trailing core is used to detect errors. The leading core forwards branch outcomes to the trailing core.

Greskamp and Torellas [32] suggest that the leading core can safely be operated at up to 1.3 times the nominal frequency with no errors. They present four reasons why processors can be safely overclocked.

1. Grading artifacts. Each processor die undergoes the process of speed binning. The objective of speed binning is to assign one of several pre-determined speed grades to each part. However, since the bin frequencies are discrete, on average, a part will able to operate at a frequency higher than the bin frequency. For instance, if the bin frequencies are 3.0 GHz, 3.5 GHz and 4.0 GHz, a part capable of operating at 3.7 GHz will assigned to the 3.5 GHz bin. Such a part can be safely overclocked by about 6%.

2. Within-die variation. Humenay et al. [37] report that within-die process variations can result in considerable difference between the fastest and slowest core in a CMP. Their results show a 17% variation between the fastest and slowest core of a CMP.
3. **Safety margins.** Modern processors are designed considerable margins in order to account for device aging, operating temperature and supply voltage variations. These margins can be exploited for overclocking.

4. **Error tolerance.** Several studies have demonstrated that the onset of timing errors is gradual in an overclocked part. For instance the Razor project [26] achieves a 17% increase in frequency in exchange for an error rate of one error every $10^6$ instructions.

There are two drawbacks to leader-follower architectures that perform timing speculation. Firstly, it may not be possible to significantly overclock every manufactured part. For instance, under worst-case temperature conditions for certain chips that have undergone significant aging, there may not exist any timing margin that can be exploited to improve performance. Secondly, timing speculation is restricted to the *sphere of replication* because faults that occur outside the sphere cannot be detected. Consequently, Paceline does not perform timing speculation in the lower levels of the memory hierarchy, limiting the performance benefits of this scheme.

Recent embedded processors designed by Marvell and Broadcom [1, 2] have implemented automatic voltage scaling (AVS). AVS internally tests transistor speed and adjusts the supply voltage to minimize power for a given target frequency. Doing so helps these processors to reclaim performance lost to due to conservative design margins. This suggests that these margins can also be exploited for timing speculation as suggested by Paceline [32], Razor [26] etc.

Paceline’s evaluation shows a performance improvement of 21% for the SPECint programs and 9% for the SPECfp programs.

**Slipstream** [71, 101] is a proposal for CMP performance enhancement that is also based on a speculative leader-follower architecture with execution assistance.

The leader core in a Slipstream processor detects and elides *ineffectual instructions*. An instruction may be classified as ineffectual if it satisfies one of the following conditions:
1. The instruction computes the branch conditions of predictable branches.
2. The value produced by the instruction is not consumed (dynamically dead).
3. The value produced is the same as the previously written value (silent).

Note that the rules are applied transitively. For instance, if the only consumer of an instruction’s result is a silent instruction, then producer is regarded as dynamically dead.

Ineffectual instructions are detected in the trailing core in a Slipstream processor. Instructions detected ineffectual train an ineffectual instruction predictor in the leading core. This predictor is consulted in the leading core to bypass fetching of ineffectual instructions. Therefore, the leading core *speculatively removes ineffectual instructions* from its execution stream. This results in faster execution of the leading core.

The trailing core is accelerated by execution assistance provided by the leading core. It also detects misspeculations in the instruction removal predictor and handles recovery from misspeculation.

The **Decoupled Performance Correctness Architecture** [28] executes a “skeleton” program on the leading core which generates branch outcomes and cache prefetches for the trailing core. The trailing core executes the full program and is accelerated by the branch outcomes and cache prefetches produced by the skeleton.

A related architecture introduced by Mesa-Martinez and Renau is based on **Circuit Pruning** [57]. The leading core’s circuits are pruned by eliminating logic that is rarely used. Circuit pruning increases the frequency of operation of the leading core, but may result in circuits no longer capable of correct operation on all inputs. The trailing core, which is not pruned, is used to ensure correct execution. As in other proposals, execution assistance from the leading to the trailing core enables the slower frequency trailing core to keep up with the leading core.

### 2.2.5 Discussion

Redundant execution is a well-studied mechanism for fault tolerance. However, there are number of design choices in the application of redundant execution that can trade-off
fault coverage, performance or power among each other.

The mechanisms based on redundant multithreading (RMT) described in §2.2.1 have the design choice of executing the two threads on the same or on different cores of an SMT processor. Redundant execution on the same core can guarantee only transient fault coverage, but has reduced performance overheads.

The base assumption of RMT is a baseline SMT processor. Lee and Brooks [50] studied power-performance efficiency for CMP and SMT processors and found that efficient SMT processors require wider (e.g., 8-way) and deeper pipelines. Since many microarchitectural structures scale in a non-linear fashion with increasing issue width [65], wider pipelines have higher complexity, larger area and lead to lower clock rates. A study by Sasanka et al. [84] found that CMPs outperform SMT architectures for multimedia workloads. Another study by Li et al. [51] found that CPU bound workloads perform better on CMP architectures while SMT architectures perform better on memory bound workloads. SMT architectures also have the problem of destructive interference among threads that can result in performance degradation [53]. These studies suggest that the base assumption of an SMT processor that is made by SRT and its derivatives might not be desirable for the low-cost fault-tolerant systems that are the focus of this work.

The mechanisms described in §2.2.2 use multiple cores of a CMP to perform redundant execution. While this provides transient and permanent fault coverage, these mechanisms have a higher cost because they use an entire core for redundant execution. Furthermore, since lockstepped execution of cores is no longer feasible, CMP based redundant execution needs complex mechanisms to ensure precise input replication.

Schemes for both redundant multithreading as well as redundant execution in CMPs replicate fetch, decode, execution and retirement of each logical thread. However, the proposals mentioned in §2.2.3 eschew replication of the fetch and decode stages, instead duplicating instructions at the time of dispatch. This reduces fault coverage but simplifies implementation.

A number proposals have also attempted to provide fault tolerance as well performance improvements (see §2.2.4). With the increasing core counts in future CMPs, these
Chapter 2. Fault-Tolerant Microarchitectures

mechanisms may become important as they provide an easy way to utilise idle cores for both fault tolerance as well performance improvement.

2.3 Symptom-Based Error Detection

Unlike redundant execution, an orthogonal approach to the detection of errors is taken by mechanisms like ReStore [110] and Perturbation Based Fault Screening (PBFS) [74]. These proposals use the technique of symptom-based error detection.

2.3.1 Introduction

ReStore observes program behaviour and microarchitectural variables during the course of execution of the program. If these variables deviate from their “expected” values, ReStore takes these deviations as symptoms of a potential soft error. Examples of symptoms examined by ReStore are exceptions, high-confidence branch mispredictions, cache and TLB misses. If the symptom is indeed the outcome of a soft error, then the symptom will not recur on re-execution. Therefore, in the ReStore architecture, a symptom triggers re-execution from a checkpoint. If the error disappears on re-execution, then the soft error has been successfully dealt with. On the other hand, if the same error recurs, then it is unlikely to have been caused by a soft error.

2.3.2 Mode of Operation

Figure 2.3 (a) shows an example of successful soft error detection and correction with ReStore. A soft error results in the triggering of a symptom. When the symptom is detected, the previous checkpoint is restored and program re-execution begins from that point. In this case, the error does not recur, and hence the symptom is not triggered again. The soft error is successfully dealt with.

Figure 2.3 (b) shows an example of false positive error detection with ReStore. In this case the symptom is triggered even though a soft error has not occurred. It is likely that the symptom will be re-triggered upon re-execution. This does not affect correctness,
(a) Successful error detection and recovery in ReStore.

(b) False positive symptom or unsuccessful error recovery in ReStore.

Figure 2.3: Symptom based error detection in ReStore

but program performance is slowed due to unnecessary re-execution.

Two other cases are possible with ReStore. An error may have occurred before the
previous checkpoint was taken, but a corresponding symptom is triggered only after the
checkpoint is taken. In this case, restoring to the previous checkpoint will not eliminate
the symptom, as the checkpoint itself is corrupted.

Another possibility is that an error occurs, but fails to trigger any of the symptom
detectors. In this case also, the error is missed by ReStore.

2.3.3 Discussion

Symptom based error detection can detect many faults at a low hardware overhead.
Wang and Patel’s results [110] show that 93% of injected faults in an out-of-order su-
superscalar microprocessor were successfully detected and corrected by ReStore. This is
equivalent to 2X increase in the mean time between failures (MTBF) of the micropro-
cessor. They estimate this increase in MTBF comes at a performance overhead of only
a few percent.

Although symptom-based fault detection is attractive due to reduced overheads, it
has an important disadvantage. It is inherently incapable of detecting all faults because
faults which do not trigger symptoms will be missed by a symptom-based detector. Although fault coverage can be increased by expanding the scope of the symptoms, the
results in an increased false positive rate, which in turn leads to poor performance.

In this thesis, we study mechanisms for redundant execution rather than those for
symptom based error detection due to the higher fault coverage of techniques based on
redundant execution.

2.4 Circuit Level Mechanisms

There is a rich body of work on circuit level mechanisms for detecting errors [11, 26, 36,
55, 58, 95]. This section presents some of these proposals and discusses their trade-offs
vis-à-vis architectural mechanisms for fault tolerance.

Razor [26, 55] replicates critical pipeline registers and detect errors by comparing
the values stored in them. In effect, Razor performs timing speculation at the circuit
level. The base assumption of a Razor-based design is that augmenting only a small
number of time-critical paths of a circuit is sufficient to detect wear-out related errors.
Recent work by Sartori et al. [83] has called into question the assumption that high-
performance processor circuits contain only a small number of time-critical paths. Sartori
et al. find that even for circuits which have only a small number of timing-critical paths,
Razor may be ineffective if there are some short paths, which may cause false positive
error detections. As a result, Razor may not be directly applicable for high-performance
processor circuits.

Some recent proposals have attempted to make Razor applicable to high performance
circuit designs by reshaping the error rate vs. frequency curve. For instance, BlueShift
employs on demand selective biasing and path constraint tuning in order to optimise the
most frequently exercised critical paths of processor circuits [33]. A related proposal by
Kahng et al. [42] attempts to perform power aware slack redistribution. Timing slack
is redistributed from paths which are not frequently used to paths which are commonly
exercised in order to make the circuit amenable to timing speculation.
The SPRIT$^3$E \cite{95} framework is similar to Razor; it replicates critical pipeline registers in order to perform timing speculation. The two registers are sampled at different times to detect timing errors. SPRIT$^3$E allows better than worst case operation of circuits by dynamically tuning clock frequency at runtime based on error rate. Avirneni et al. \cite{11} introduced the SEM and STEM cells, which are replacements for critical pipeline latches specifically designed for timing speculation. These cells use three latches sampled at different times, with errors detected by comparing the values latched in each replica.

Mitra et al. \cite{58} have proposed a radiation hardening scheme that detects faults using at-speed scan logic. The C-Element acts as an inverter when both inputs to it are the same. When the inputs are different, the C-Element does not let either input propagate. Thus transient faults can be detected by connecting the redundant storage element from the scan cell to two inputs of a C-Element. The C-Element requires the scan circuit to operate at-speed. High-end VLSI designs are moving away from at-speed scan due to power consumption limitations \cite{59}.

### 2.4.1 Discussion

Razor, SPRIT$^3$E and SEM/STEM cell are primarily targeted at detecting timing errors. The ability of these mechanisms to detect soft errors is unclear. For duplicated latches to detect soft errors, the spatial distance between the replicated latches has to be large. Otherwise, the two latches may be affected in the same way by the soft error pulse. This may be quite difficult, because the spatial range of electron hole pairs generated by particle impacts may be up to 0\( .1\)\( \mu m \) \cite{118}.

Multiple-sampling techniques also assume that transients caused by a particle strike will not affect both sampling instants. This assumption has been called into question by the work of Chardonnereau et al. \cite{23} who experimentally found that the width of transient pulses can be as much as 800 \( ps \), several times the clock cycle of current processors.

Advantages of circuit level techniques when compared to architectural mechanisms are lower area and power costs. However, unlike architectural mechanisms, some circuit
level mechanisms cannot be disabled dynamically by users or applications not requiring fault tolerance.

2.5 Lifetime Reliability Management

The techniques for fault detection discussed so far attempt to detect and possibly correct faults. A different, but equally important problem is preventing faults. Faults can be prevented or their onset delayed by lifetime reliability management.

The concept of lifetime reliability management was introduced by Srinivasan et al. in [94]. Lifetime reliability management is based on the observation that wearout mechanisms like electromigration, NBTI and hot carrier injection are temperature dependent, and exacerbated by high activity factors. Furthermore, NBTI also has a recovery phase, which can be exploited to reverse threshold voltage degradation. Therefore, degradation caused by wearout mechanisms can be reduced in the following ways:

- **Reducing operating temperature.** Operating temperatures can be reduced by the well-known technique of dynamic voltage and frequency scaling (DVFS). Reducing voltage and frequency reduces temperature of operation, and hence mitigates degradation mechanisms. Unfortunately, this results in a performance loss. This approach was first studied in [94].

- **Reducing activity factors.** Lower activity factors also lead to reduced wearout. Lower activity factors can be achieved in several ways. Siddiqua and Gurumurthi use an NBTI-aware instruction scheduler which equalises the activity factors across functional units [87]. Facelift [104] proposes aging-driven application scheduling in combination with temperature management.

- **Exploiting NBTI Recovery Phases.** Penelope [4] is an NBTI-aware processor design that minimises the impact of NBTI on SRAM structures in processors. Processor SRAM structures are vulnerable to NBTI if the values stored in the SRAM structure are biased towards ‘0’, the stress-causing value for NBTI. Penelope
avoids constant stress by ensuring that ‘0’ and ‘1’ are stored for approximately equal
intervals of time in processor SRAM structures.

Lifetime reliability management is an important technique that needs to incorporated
in future processors in order to delay the onset of wearout related failures. However, the
concern of lifetime reliability is orthogonal to that of fault detection. This thesis is
primarily concerned with fault detection through redundant execution, so we do not
explore lifetime reliability management further.

2.6 Concluding Remarks

This chapter presented a brief overview of related work in fault-tolerant architectures and
circuit designs. Section 2.1 introduced a general framework for redundant execution. In
section 2.2, we presented an overview of different proposals for redundant execution. We
discussed several design options for redundant execution and trade-offs between these
options. Section 2.3 explored symptom-based error detection, an approach for error
detection that is orthogonal to redundant execution. In section 2.4, we gave a brief
overview of circuit level mechanisms for error detection and discussed some shortcomings
of these mechanisms. Finally, section 2.5 presented a brief overview of lifetime reliability
management, an important reliability related design concern for future microprocessors
that is related to, but distinct from, fault detection and recovery.
Chapter 3

Energy-Efficient Redundant Execution

3.1 Introduction

Relentless technology scaling has resulted in the onset of an era of decreased CMOS reliability. Consequently, fault tolerance is expected to be important for future commodity microprocessors [6, 38]. Traditional fault tolerant systems were restricted to the domain of mainframe computers or specially design fault-tolerant processors [13, 27]. These systems spare no expense to meet reliability targets, but come with extremely high costs. Typically, they have more than 100% area and power overheads when compared to an equivalent non-redundant system. Therefore, these systems are not directly usable as solutions to the reliability problem in commodity microprocessors. For future commodity microprocessors, there is a need for configurable, low cost and software-compatible fault tolerance mechanisms. In this context, an important problem for fault-tolerant CMP designs is of energy-efficiency. As was briefly touched upon in §1.3.2, energy-efficient fault tolerance is important for three reasons.

The performance of modern CMPs is limited by power consumption and peak temperature. The power budget for a chip is fixed, so decreasing the power consumed in any core makes more power available to other cores. This additional power can be used to increase the performance of those cores by operating them at higher voltage and frequency levels [34, 41]. For example, such mechanisms are present in the Intel Nehalem (Core i7 TM) processor [39].
Decreasing power consumption often results in a decrease in the operating temperature as well. For instance, the Sun UltraSPARC T2 limits power consumption through limited execution speculation, aggressive fine-grained clock gating and a fully static design [69]. These design decisions resulted in an operating temperature of 66°C when running typical workload applications. This is a significant reduction from typical temperatures, which are around 100°C, and translates to an order of magnitude increase in lifetime reliability. Parulkar et al. [69] report that this decrease in temperature resulted in a factor of 9 increase in GOI (gate-oxide integrity) median time-to-breakdown, while average failure rate decreased by a factor of 17. NBTI degradation improved by 29%, equivalent to an eight-fold increase in lifetime. Soft error rate also increases exponentially with temperature [25], so decreasing temperatures also helps mitigate the soft error problem.

Decreasing the energy-overhead of fault tolerance is also important from the point of view of data center energy. Data center energy is expected to increase to an unprecedented level of 100 billion kilowatt hours by 2011 [56]. Unreliable chip components would imply that a significant fraction of future data centers would require fault tolerance mechanisms to cope with hardware faults. In this context, decreasing the energy-overhead of fault tolerance helps mitigate the problem of data center energy consumption.

Clearly, there is a pressing need for energy-efficient fault-tolerant architectures. This chapter presents two proposals for fault tolerant CMPs and extensively evaluates their performance.

### 3.1.1 Overview

In this chapter we present two architectures for energy-efficient fault tolerance in chip multiprocessors. These architectures employ the technique of space redundancy, executing a single logical thread on two cores of a multicore processor. The outputs of the two cores are compared to detect errors.

Figure 3.1 shows an 8-core CMP executing 3 logical threads redundantly. Logical thread 1 is being executed on core 0 and core 1, logical thread 2 is being executed on
core 2 and core 3 and logical thread 3 is being executed on core 4 and core 7. The cores are connected by a shared bus interconnect.

Of the two cores executing a single logical thread, one is designated as the leading core, while the other is designated as the trailing core. The leading core assists execution of the trailing core by forwarding the results of its execution. Execution assistance speeds up execution in the trailing core. This speedup is exploited by operating the trailing core at a lower voltage and frequency level in order to reduce energy consumption.

The two architectures that we propose differ in the type of execution assistance that is provided by the leading core to the trailing core. The first architecture, Redundant Execution using Simple Execution Assistance (RESEA) [97, 98], forwards load values and branch outcomes from the leading core to the trailing core, similar to an idea first proposed by Reinhardt and Mukherjee [77]. We show that this form of execution assistance can be exploited to obtain significant energy savings in the trailing core. Our evaluation estimates that this proposal has a performance overhead of less than 1% and an energy consumption of about 1.34 times that of non-redundant execution. We introduce an optimisation to this proposal, which we call the early-write optimisation, which further decreases energy consumption to about 1.26 times that of non-redundant execution. Section 3.2 presents the details of this architecture.

An important drawback of RESEA is high interconnect bandwidth. RESEA is based on forwarding the results of all branch outcomes and all load values. Load and
branch/jump instructions constitute more than one-third of the instruction mix of the SPEC CPU 2000 benchmark suite. As a consequence, in the case of the aggressive superscalar processor which is the baseline for our designs, RESEA forwards one value almost every two cycles from the leading core to the trailing core. Such high interconnect bandwidth might prove to be a significant bottleneck. Hence, reducing the bandwidth cost of execution assistant is important if this architecture is to be realised in practice.

Our second architecture, called Redundant Execution using Critical Value Forwarding (RECVF) [100], is based on the idea of critical value forwarding (CVF). The goal of CVF is to identify instructions on the critical path of execution in the leading core and forward the results of these instructions to the trailing core. CVF breaks data dependence chains in the trailing core. Hence, its performance is significantly improved. Since CVF focuses on critical path instructions, it obtains significant speedup at a low bandwidth cost.

Our results show critical value forwarding enables the design of an energy-efficient fault-tolerant architecture with a performance overhead of around 1%, and energy consumption that is 1.26 times that of non-redundant execution. While the performance overhead and energy consumption of RECVF are similar to RESEA with the early-write optimisation, the key difference is in the interconnect bandwidth requirement. *Critical value forwarding requires only about one-third of the bandwidth that simple execution assistance requires but provides equivalent performance.* Reduced bandwidth requirements enable the use of RECVF on NoC or bus-based communication fabrics. Details of RECVF are presented in §3.3.

The rest of this chapter is organised as follows. Section 3.2 discusses the design of RESEA. Section 3.3 describes modifications that need to be made to RESEA to support critical value forwarding. Section 3.4 describes and evaluates an extension to RECVF that can perform correct execution on cores with faulty functional units. Finally, section 3.6 concludes.
3.2 Redundant Execution using Simple Execution Assistance

This section describes an architecture for Redundant Execution Based on Simple Execution Assistance (RESEA). In an RESEA processor, a single logical thread is executed on two cores. One of these cores is designated as the leading core while the other is designated as the trailing core. The leading core forwards the results of load instructions and resolved branch outcomes to the trailing core. This execution assistance speeds up the trailing core. This speedup is exploited by operating the trailing core at a lower voltage and frequency level.

The key design challenges for RESEA are:

- Microarchitectural structures for using load values and branch outcomes.
- Fault detection and isolation logic.
- Per-core DVFS algorithms to dynamically set the frequency of the trailing core.

The following subsections describe our solutions to these design challenges.

3.2.1 Microarchitectural Support

![Block diagram of core supporting RESEA](image)

Figure 3.2: Block diagram of core supporting RESEA
For energy-efficient redundant execution, RESEA augments a conventional out-of-order superscalar processor core with some additional structures. Figure 3.2 shows a block diagram of the modified core.

Branch outcomes from the leading core are forwarded to the trailing core’s Branch Outcome Queue (BOQ) [77]. During instruction fetch the trailing core does not use the branch predictor. Instead it accesses the BOQ to get the direction and target address of the branch. In the absence of soft errors, the BOQ provides perfect branch prediction for the trailing core.

The Load Value Queue (LVQ) [77] structure is used to hold load values in the trailing core. Forwarding load values from the leading to the trailing core has two benefits:

1. It solves the problem of input replication and avoids input incoherence [77, 91].
2. It speeds up the trailing core by eliminating data cache misses.

Early-Write Optimisation

Previous implementations of the LVQ accessed it at the same time as the access to the data cache, i.e., after the effective address computation. This is an unnecessary delay because the LVQ entry from which the value has to be read is known at the time of instruction decode. Therefore, we introduce the early-write optimisation. This optimisation reads the LVQ and writes the result into the destination register at the time of instruction dispatch. The effective address computation takes places later and is used only for accessing the TLB and in the fingerprint computation (see §3.2.2). As a result of this optimisation, instructions which are dependent on load instructions can begin execution immediately after the load instruction is dispatched.

The early-write optimisation breaks data-dependence chains containing load instructions in the trailing core, improving its IPC by over 30%.
3.2.2 Fault Detection

Periodically, the leading core and trailing core compute a hash value that summarizes updates that have been made to the state of the processor. This hash value is referred to as a fingerprint [90]. The two cores swap and compare fingerprints to detect errors. If an error has not occurred, then the architectural updates will be exactly the same because the sequence of retired instructions in the leading and trailing cores is identical. This guarantees that the fingerprints will also be equal. If an error occurs, the fingerprints are extremely likely to be different. A mismatch in fingerprints indicates the occurrence of an error.

Another way in which RESEA detects errors is when a branch misprediction is detected in the trailing core. Since resolved branch outcomes are forwarded from the leading to the trailing core as predictions, when the trailing core detects a misprediction, it must be due to an error.

Fingerprint Aliasing

A mismatch in the fingerprints calculated by the two cores necessarily implies the occurrence of a fault, but a fault may be such that the fingerprints computed at the two cores still match, even though the architectural updates are different. This situation is referred to as fingerprint aliasing and occurs with probability $2^{-p}$ where $p$ is the width of the fingerprint. Larger fingerprint widths can be used to reduce the probability of aliasing to an acceptably low level albeit at a slightly higher hardware cost. An appropriate fingerprint width can be calculated given the values of raw error probability and a target mean time to failure (MTTF). A target MTTF of 1000 years, assuming an error occurs every 10 days and fingerprints are compared every 50,000 cycles requires a 16-bit fingerprint. For the same target MTTF, the assumption of an error every hour requires a 24-bit fingerprint.
Checkpointing and Recovery

If the fingerprints are found to match in both cores, the leading core stores all architectural registers in the checkpoint store. It then clears the unverified bits of all cache lines (see §3.2.3).

If the fingerprints do not match in at least one core, then recovery is performed in three steps. Firstly, the register states of the two processors are restored from the checkpoint store. Secondly, all unverified lines in the L1 data cache are invalidated. Finally, the processors restart execution from the next instruction after the last checkpoint.

3.2.3 Fault Isolation

When a fault occurs in an RESEA processor, it may be detected only when the next fingerprint comparison occurs. Between the time that the fault occurs and the time it is detected, fault isolation requires that the fault must not propagate outside the processor or to other processes. There are two ways in which this can happen.

Firstly, a corrupt cache block may be replaced and written back to the lower level of the memory hierarchy, from where it can propagate to main memory or other processes. RESEA prevents this by using a modified L1 cache similar to speculative version data caches [48]. The L1 data cache stores an unverified bit along with every cache line. Any write to cache line sets the unverified bit. If the unverified bit is set, a cache line is deemed to be locked and not allowed to be written back to the lower level of the memory hierarchy. When fingerprints are compared and found to match, the unverified bits of all lines in the cache are flash-cleared. When a write to a verified line is performed, the line is written back to the L2 cache so that a verified copy of the line is available for recovery. If fingerprints do not match, then memory state is recovered by invalidating all unverified lines in the L1 data cache. Since lower levels of the memory hierarchy always contain verified data, all memory updates since the last checkpoint are “undone” by the invalidation.

For correct execution of multithreaded workloads, the verified bit must be transmitted along with the data when one cache supplies data to another cache. If a line needs to be
replaced and all the lines in its set are locked, then a fingerprint comparison is initiated. When the fingerprint comparison is complete, the lines will be unlocked and the memory access can be completed.

The second method by which a fault may propagate outside the processor is through I/O operations. RESEA forces a checkpoint to be taken and fingerprints compared before each I/O operation. This ensures that I/O is done only with verified and correct data.

3.2.4 Voltage and Frequency Control

Forwarding load values and branch outcomes to the trailing core allows it to execute faster than the leading core. This speedup can be exploited by operating it at a reduced voltage and frequency level. The challenge here is to design an algorithm that can dynamically set the voltage/frequency levels of the trailing core based on program phase behaviour.

A dynamic mechanism for setting the voltage and frequency is required because of time-varying program phase behaviour [85]. In phases of high IPC, the trailing core frequency needs to be higher, while low IPC phases can be exploited by operating the trailing core at a lower frequency.

In this context, we make the key observation that the sizes of the BOQ and the LVQ are an indication of the difference in execution speed between the two cores. To understand why, let us assume for a moment that the trailing core has infinite sized BOQ and LVQ structures. If the trailing core is operating at lower than its optimal frequency, its execution will be temporally behind the leading core, and the number of elements in the LVQ and BOQ will continuously increase. On the other hand, if the trailing core is operating at higher than its optimal frequency, the LVQ/BOQ structures will likely be empty. This suggests that an algorithm which varies the frequency of the trailing core based on the number of entries in the queues will be able to track IPC variations in leading core.
QSize-DVFS Algorithm

Our algorithm, called the QSize-DVFS algorithm, periodically samples the size of the BOQ and the LVQ at a fixed time interval of $T_s$ seconds. It uses two thresholds each for the BOQ and the LVQ: a high threshold and a low threshold. If the occupancy of both structures is greater than the high threshold, then the frequency of operation is increased. If the occupancy of both the structures is less than the low threshold, then the frequency of operation is decreased. The algorithm attempts to maintain the occupancy of the structures in between the low and high thresholds.

The thresholds can be set either statically or dynamically. Our results show that a single static threshold for all programs provides significant power savings with a small performance degradation. Hence, we only use a statically set threshold value.

3.2.5 Fault Coverage

RESEA detects faults that occur in processor logic with the exception of those that occur in certain parts of the memory access circuitry. Since only one core accesses the memory hierarchy, faults that affect the memory access circuitry (e.g., the store-to-load forwarding logic) may not be detected. Circuit level techniques [36, 58] may be used to detect these errors. A similar problem exists with cache controller logic and memory controller logic. Although ECC can protect the data and possibly the tag bits in a cache, it cannot protect against errors in the controller logic. Depending on the reliability target, this logic may have to be protected by circuit level techniques such as radiation hardening.

This loss in fault coverage is not unique to RESEA, but is common to SRT [77] and all its derivatives such as SRTR [108], CRT [60] and CRTR [30] and SpecIV [46]. Even architectures like Reunion [91], which independently access the L1 cache instead of replicating load values, share the lower levels of memory hierarchy. As a result, they provide only slightly higher fault coverage than RESEA for the memory subsystem. For example, an error occurring in the shared L2 cache controller circuitry cannot be detected by Reunion even if the L2 cache data is protected using ECC.
Since RESEA only compares fingerprints, there is some loss in fault coverage due to fingerprint aliasing. However as shown in [90], for reasonable error rates and fingerprint widths, the probability of an undetected error due to fingerprint aliasing is minuscule.

### 3.2.6 Evaluation

| # of cores | 8 |
| Nominal frequency | 3 GHz |
| ROB size | 128 instructions |
| Integer/FP window | 64/32 instructions |
| Mem/Int/FP units | 4/6/4 |
| D-cache | 64k/64B/4-way/2 cycles |
| Branch target buffer | 4k entries, 4-way set-associative |
| Branch predictor | hybrid of bimodal/gshare |
| Checkpointing interval | 50,000 instructions |
| L2 | 16 MB/64B/8-way/40 cycles |
| Technology node | 32 nm |
| Fetch/issue/retire | 4/4/4 instructions per cycle |
| Int/FP registers | 160/128 |
| Load/store queue | 32 instructions |
| I-cache | 32k/64B/4-way/2 cycles |
| Memory | 400 cycles |
| Return address stack | 32 entries |
| BOQ size | 64 entries |
| LVQ size | 512 entries |
| Checkpointing latency | 64 cycles |
| Interconnect latency | 24 cycles |

Configuration for the PVA architecture of Rashid et al. [76]

| PCB size | 1024 entries |
| PCB access hash table | 257 entries |
| Hash table access latency | 1 cycle |
| PCB sections | 8 |
| PCB access latency | 8 cycles |

Table 3.1: CMP configuration for SEA evaluation

Our evaluation uses an appropriately modified version of the SESC execution-driven simulator [80]. The simulator models an out-of-order superscalar processor in a detailed manner and fully simulates “wrong-path” instructions. Details of the CMP model are given in Table 3.1.

Our simulations model fault-free execution. The rationale for this is that faults occur only rarely so the performance and power of the system are determined by the characteristics of the fault-free case. We assume that the fault detection capabilities of the system are unchanged by DVFS. However, we note that soft-errors increase exponentially with a decrease in voltage, causing an increase in the number of detected faults due to our DVFS proposals for energy-efficient redundancy [25]. However, lower power consumption leads to lower temperatures, so our techniques have a positive impact on permanent fault rates.
In order to put our results in context, we compare our architecture against two previous proposals: (1) the Parallelized Verification Architecture (PVA) from [76] and (2) Chip-level Redundantly Threaded (CRT) processors from [60].

To estimate energy consumption, we modified SESC’s power model, which is based on Wattch [19]. We included power models for the LVQ, BOQ and the Post Commit Buffer (PCB) used in PVA. CACTI 5.3 [103] was used to model the energy of the shared L2 cache. Leakage power is modeled only for the array structures in the processor like caches, branch predictors and TLBs using Wattch and CACTI’s leakage power model. While leakage power is modeled as average energy per clock cycle, dynamic energy is modeled using a constant average energy per access model. The CACTI and Wattch models internally use the technology node (32nm) to appropriately scale dynamic and leakage power values.

The voltage-frequency levels for per-core DVFS used in our study are shown in Table 3.2. We assume fine-grained, low-latency per-core DVFS similar to the proposal in [43].

| Voltage (V) | 1.0 | 0.9 | 0.8 | 0.7 | 0.6 | 0.6 |
| Frequency (GHz) | 3.0 | 2.7 | 2.4 | 2.1 | 1.8 | 1.5 |
| DVFS level change latency | 100ns |
| DVFS update interval | 1µs |

Table 3.2: Voltage-frequency levels for per-core DVFS

Workload

We simulated ten integer and ten floating point benchmarks from the SPEC CPU 2000 benchmark suite. For each benchmark, we executed a single SimPoint [85] of length one billion instructions.

IPC Results

Figure 3.3 shows the normalised IPC of the benchmarks from SPEC CPU 2000 suite. The IPC is normalised by the IPC of non-redundant execution in order to measure the
Figure 3.3: Normalised IPC of the SPEC benchmarks for RESEA

performance overhead of fault tolerance. The bars labelled PVA and CRT refer to the proposals of Rashid et al. [76] and Mukherjee et al. [60] respectively. RESEA refers to our baseline proposal and ‘RESEA+EarlyWrite’ refers to our proposal enhanced with the early-write optimisation.

The geometric mean normalised IPC of PVA is about 95.35%, while that of CRT is about 95.25%. In contrast both RESEA configurations have slowdown of less than 1%, i.e. mean normalised IPC greater than 99%.

**Energy Results**

Figure 3.3 shows the normalised energy of the benchmarks from SPEC CPU 2000 suite normalised by the energy of non-redundant execution. Geometric mean normalised energy consumption of CRT is 1.52 times that of non-redundant execution. Mean normalised energy consumption of PVA is 1.32 times that of non-redundant execution. RESEA without early-write has a mean energy consumption of 1.34 times that of non-redundant execution. With the early-write optimisation, RESEA has energy consumption of only 1.26 times that of non-redundant execution.
RESEA’s energy consumption is higher for benchmarks that have low data cache miss rates and higher branch prediction accuracy. This is the reason for the high energy consumption of bzip2, vortex, ammp, mesa and sixtrack.

**Bandwidth Results**

![Figure 3.5: Bandwidth requirements for RESEA](image)

Figure 3.5 shows the bandwidth requirements for each of the configurations that we
examine. CRT has the highest bandwidth requirements, while PVA has the lowest. The bandwidth requirements of RESEA are significantly higher than PVA.

3.2.7 Discussion

Our results show that RESEA without the early-write optimisation has a lower performance overhead but slightly higher energy consumption than PVA. RESEA with the early-write optimisation has a lower performance overhead and lower energy consumption than PVA. Both PVA and RESEA are faster and consume lesser energy than CRT. An important difference between PVA and RESEA is that PVA uses 3 cores to execute a single program, while RESEA uses only 2 cores. Thus PVA’s throughput is significantly lower than RESEA.

Although RESEA’s results are promising, it has two disadvantages. The first is reduced fault coverage because the trailing core does not re-execute load instructions. The second is the high bandwidth requirement imposed on the interconnect. The ideas of partial load replication and critical value forwarding introduced in the next section describe one solution to these problems.

3.3 Redundant Execution using Critical Value Forwarding

This section proposes Redundant Execution using Critical Value Forwarding (RECVF), which is an architecture for energy-efficient fault-tolerant CMPs. RECVF executes one logical thread on two cores of a CMP. One of these cores is designated as the leading core, while the other is designated as the trailing core.

We introduce the idea of critical value forwarding (CVF). In an RECVF processor, the leading core assists execution of the trailing core by forwarding the results of instructions on the critical path of execution. CVF breaks data dependence chains in the trailing core because the results of instructions on the critical path are made available to the trailing core even before they complete execution. This in turn allows instructions dependent on these instructions to execute earlier, creating a cascade effect that
improves the performance of the trailing core.

RECVF solves the following key challenges in design of such an architecture:

- Identifying instructions on the critical path. The challenge here is to identify a few critical instructions that have the most impact on performance.

- Designing mechanisms for transferring the results of these instructions from the leading to the trailing core.

- Validating the forwarded values in the trailing core to ensure correct operation even in the presence of an error in the forwarded values.

We show how critical value forwarding can be combined with per-core Dynamic Voltage-Frequency Scaling (DVFS) [41, 43]. Per-core DVFS allows the trailing core to execute at a much lower frequency than the leading core, significantly reducing the energy overhead of redundant execution. In this context, we propose two algorithms for per-core DVFS and examine the energy savings due to these.

3.3.1 Core Architecture

Figure 3.6 shows the block diagram of an RECVF processor core. The processor pipeline is augmented with three additional structures, the branch outcome queue (BOQ), the instruction result queue (IRQ) and circuitry implementing a critical value identification heuristic. The BOQ and the IRQ are used only in the trailing core, while critical value identification is performed only in the leading core.

Identifying Critical Values

Our approach to identify critical path instructions is similar to [105]. The basic idea is to mark an instruction as critical if it satisfies certain marking criteria during its execution. We evaluated a number of critical value identification heuristics based on this principle. A list of these is shown in Table 3.3.
### Chapter 3. Energy-Efficient Redundant Execution

#### Table 3.3: Critical Value Identification Heuristics

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Marking Criteria</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 roStall</td>
<td>Instruction at the head of ROB prevents retirement because it is not yet executed.</td>
<td>Instructions that are unable to execute until they reach the head of the ROB are likely to be on the critical path.</td>
</tr>
<tr>
<td>2 instQHead</td>
<td>Instruction reaches head of instruction queue before being selected for execution.</td>
<td>Instructions unable to execute until they reach the head of the instruction queue are likely to be on the critical path.</td>
</tr>
<tr>
<td>3 instQHFree</td>
<td>Instruction produces a value that frees an instruction at the head of its queue.</td>
<td>Forwarding this value may help the dependent instructions execute earlier in the trailing core.</td>
</tr>
<tr>
<td>4 freedN</td>
<td>Instruction frees at least N instructions for execution when it completes.</td>
<td>An instruction that frees a large number of other instructions for execution is more likely to be on the critical path.</td>
</tr>
<tr>
<td>5 fanoutN</td>
<td>Instruction produces a value that is consumed by at least N other in-flight instructions.</td>
<td>An instruction that produces a value that a large number of other instructions consume is likely to be on the critical path.</td>
</tr>
<tr>
<td>6 everyN</td>
<td>Every .Nth instruction is marked as critical.</td>
<td>A simple heuristic that serves as a benchmark for comparison against more sophisticated heuristics.</td>
</tr>
<tr>
<td>7 allBJ</td>
<td>All branch/jump instruction outcomes are forwarded.</td>
<td>This policy estimates the speedup obtained by forwarding just branch instructions.</td>
</tr>
<tr>
<td>8 mispredBJ</td>
<td>Only mispredicted branch/jump instruction outcomes are forwarded.</td>
<td>This policy compares the loss in speedup due to forwarding mispredicted branch outcomes in comparison to forwarding all branch outcomes.</td>
</tr>
<tr>
<td>9 loadsOnly</td>
<td>Only mispredicted branches and load values are forwarded.</td>
<td>This is the baseline for full load replication (FLR). (See §3.3.4.)</td>
</tr>
<tr>
<td>10 all</td>
<td>All possible values are forwarded.</td>
<td>Corresponds to an oracle heuristic given infinite storage space and infinite bandwidth.</td>
</tr>
</tbody>
</table>
Handling Branch/Jump Instructions

Branch/Jump instructions are handled differently for the purposes of critical value identification. RECVF marks mispredicted branch instructions as critical. The target addresses of mispredicted branches are forwarded from the leading to the trailing core. In the trailing core, these addresses are used as predictions. As will be seen in §3.3.5, this mechanism provides almost the same speedup as forwarding the results of all branch instructions, but requires very little bandwidth.

For the purpose of identifying critical instructions, branch instructions whose direction has been predicted correctly, but target address was incorrectly predicted are marked critical.

3.3.2 Operation of the Leading Core

With the exception of critical value marking and forwarding, the leading core operates like conventional superscalar processor cores. Critical value forwarding is done after instruction retirement.

Both leading and trailing core execute instructions in chunks. When the leading core
finishes the execution of a chunk it requests the trailing core to execute that chunk. An instruction index within the current chunk is forwarded along with the value by the leading core. This index is used by the trailing core to map forwarded instruction results to instructions.

After the execution of a certain number of chunks, the leading and trailing cores synchronize and exchange fingerprints to detect errors. A fingerprint \([90]\) captures updates to the architectural state of the processor by hashing register updates, load and store values and addresses and branch targets using a CRC code. Errors are detected when the fingerprints do not match. (See section 3.3.7).

### 3.3.3 Operation of the Trailing Core

#### Operation of the BOQ

The trailing core stores the branch outcomes it receives in the branch outcome queue (BOQ). Unlike previous implementations of the BOQ, our implementation does not store the targets of all branch instructions. A branch outcome is mapped to a branch instruction using the index transmitted by the leading core. When a branch instruction is fetched, the BOQ is examined to see if its outcome is available. If so, the outcome overrides the branch predictor.

#### Operation of the IRQ

The trailing core stores the results of instructions other than branch instructions in the Instruction Result Queue (IRQ). Like the BOQ, theIRQ also stores an index along with the value to map instruction results to instructions. At the time of dispatch, the IRQ is examined to see if the result of this particular instruction is available. If so, the IRQ is read and its value is written into the register file. This allows dependent instructions of this particular instruction to begin execution immediately.
3.3.4 Options for Input Replication

The default implementation of RECVF, which we refer to as **partial load replication** (PLR), fully re-executes most load instructions in the trailing core. The only load instructions that are not re-executed are those that read from cache lines obtained from cache-to-cache transfers (see §3.3.8). Experiments with the SPLASH2 benchmarks revealed that PLR re-executes 92% of all load instructions. Note that for a single-threaded program, all load instructions are fully re-executed. Hence, **PLR has most of the fault-coverage of a mechanism that fully re-executes loads without the corresponding complexity and performance/energy costs**.

We also study the option of **full load replication** (FLR). FLR works like SRT/CRT [60, 77] and replicates the results of all load instructions in the leading core and transfers them to the trailing core. This option is expected to perform better at the cost of lower fault coverage and higher bandwidth.

3.3.5 Effectiveness of Heuristics for Critical Value Identification

Figure 3.7 shows the performance of the critical value identification heuristics. The graph shows the mean speedup and bandwidth for each of the heuristics.

Speedup is the ratio of the IPC of the trailing core to that of leading core. Both cores are operated at the nominal frequency. The IPC of the trailing core is computed only over its active period, i.e., excluding the regions of time between the completion of
chunk \(i\) and the start of execution of chunk \(i + 1\). This is a conservative estimate of the speedup. Section 3.3.11 has further details on our methodology.

CVF has a large impact on the performance of the trailing core. The trailing core experiences speedup of 1.6X and 2.2X over the leading core for PLR and FLR respectively. This means that the trailing core can be operated at approximately 0.6 times the frequency of the trailing core for the PLR configuration, while it can be operated at less than half the frequency of the leading core for the FLR configuration.

**Discussion of Best Performing Heuristics**

The critical value forwarding heuristics with the highest speedups are \(\text{fanout}_2\) and \(\text{freed}_2\). Of these, \(\text{fanout}_2\) has slightly higher speedup, but this speedup comes at the cost of higher bandwidth. \(\text{freed}_2\) marks those instructions as critical which, upon completion, cause at least two other instructions to be marked ready for execution. An instruction on the critical path of execution is likely to free at least one other instruction when it completes. Therefore, an instruction which frees two other instructions is much more likely to be on the critical path. This is the reason why \(\text{freed}_2\) is the best heuristic for identifying critical values if we consider the speedup-per-unit bandwidth metric.

However, the disadvantage of \(\text{freed}_2\) is that it requires monitoring the instruction scheduling and wake up logic. These are likely to be on the timing critical paths of out-of-order superscalar processors [66]. Therefore, it is desirable to find an approximation to \(\text{freed}_2\) that provides similar speedup at comparable bandwidth cost.

For this reason, we developed the \(\text{Fanout}_2\) metric. \(\text{Fanout}_2\) marks instructions which have two in-flight consumers as critical path instructions. Note that instructions marked as critical by \(\text{fanout}_2\) are a strict superset of those marked critical by \(\text{freed}_2\).

\(\text{Fanout}_2\) can be implemented using a set of 2-bit saturating counters associated with each physical register. The counter associated with a register is incremented at the time of instruction decode when it is determined that this register is used as source operand to an instruction. The counter increments may be delayed so that this circuitry does not affect the processor critical paths. For the processor modeled in our study, the hardware
overhead of fanout2 is only 576 bits (72 bytes). Furthermore, these changes affect only the in-order pipe stages of the microprocessor, easing design and verification complexity. Fanout2 also has the highest speedup among all the critical value identification heuristics that we consider. Therefore, in the rest of this thesis, we report results only for this heuristic.

3.3.6 DVFS in the Trailing Core

Critical value forwarding creates slack in the trailing core which can be exploited by operating the core at a lower voltage-frequency level. However, the slack is not constant for all programs and varies with program phase. When the leading core is executing in a phase of high IPC, there is less slack to be exploited in the trailing core, and vice versa. Therefore, the challenge is to dynamically set the voltage-frequency level of the trailing core based on the program phase behavior. In this section we describe two algorithms for this.

QSize-DVFS algorithm

The QSize-DVFS algorithm is the same as the algorithm presented in §3.2.4. This algorithm uses the occupancies of the BOQ and the IRQ to set the frequency of the trailing core.

IPC-DVFS algorithm

The intuition behind the IPC-DVFS algorithm is that the ratio of the IPCs of the two cores can be used to set the frequency of the trailing core. For example, if for a certain period of execution, the IPC of the leading core is 1.0, while that of the trailing core is 2.0, then the trailing core ought to be operated at half the frequency of the leading core.

The IPC-DVFS algorithm generalizes this idea in the following way. The two cores keep track of their respective IPC over the DVFS update interval. At the end of the interval, the ratio of the leading core IPC to the trailing core IPC is taken, and a scaled version of this value is used to set the frequency of the trailing core for the next interval.
3.3.7 Fault Detection

To detect faults, RECVF uses the same fingerprint based output comparison scheme presented in §3.2.2. The set of cores executing a program periodically synchronize and exchange fingerprints [90]. Faults are detected when fingerprints are exchanged and at least one of the cores detects a mismatch in the fingerprints. If the fingerprints match, then the current register state is stored in a checkpoint store and all lines in the cache are marked verified (See §3.3.8). In case there is a fingerprint mismatch, then the register state is restored from the checkpoint store and all unverified lines in the data cache are invalidated (see §3.3.8) before restarting execution from the successor of the last verified instruction.

Verification of Forwarded Values

A value forwarded from the leading to the trailing core may be corrupted due to the occurrence of an error. At first glance, it appears as if we need an additional mechanism to verify the correctness of each value that is forwarded from the leading core. However, it can be shown that fingerprinting can detect errors in the forwarded values.

To see why this is the case, assume that an instruction $i_x$ in the leading core forwards an erroneous value corresponding to the instruction $i'_x$ in the trailing core. Assume without loss of generality that $i_x$ is the earliest instruction that forwards an erroneous value to the trailing core. When $i'_x$ executes in the trailing core, its input operands will have the correct (i.e. error-free) values and will compute the correct result. Consequently, $i_x$ and $i'_x$ generate different results, one correct and one erroneous. Therefore, fingerprints computed in the two cores will be different, detecting the error.

3.3.8 Fault Isolation

The fault isolation mechanisms used in RECVF is similar to the mechanism presented in §3.2.3. Each line in the cache is augmented with an unverified bit that tracks lines which have been modified but not verified. The unverified bits in the cache are flash-cleared
after a successful fingerprint comparison.

3.3.9 Parallel Application Support

To support correct execution of parallel workloads, RECVF builds on a baseline MOESI

Parallel Application Support for FLR

In the case of Full Load Replication (FLR), only the leading core accesses the memory
hierarchy. The result of each load instruction is forwarded to the trailing core by the
leading core. The trailing core does not independently access the memory hierarchy, and
instead uses the value forwarded by the leading core. Store instructions do not write to
the data cache.

This scheme is essentially the same as the mechanism used for input replication
in SRT, CRT, CRTR and SpecIV processors. In effect, the instruction result queue
(IRQ) also functions as a load value queue (LVQ) [77]. Cache coherence is performed
as if the private data caches of trailing cores are not present in the system. To ensure
fault isolation, cache coherence transactions must propagate the unverified bit when
performing cache-to-cache transfers.

Parallel Application Support for PLR

FLR has two disadvantages. Firstly, it has reduced error coverage because only the
leading core accesses the memory hierarchy. Although ECC can protect against errors
that affect the data stored in the caches, FLR is still vulnerable to logic and controller
errors. Furthermore, FLR also requires high bandwidth to transfer the result of every
load instruction from the leading core to the corresponding trailing core.

PLR attempts to increase error coverage of FLR using while maintaining a simple
implementation. It is based on the following observations.

The first observation is that a large fraction of the cache lines accessed by a particular
thread are not shared with other processors. This means that loads and stores to these
cache lines can be fully and safely re-executed in both the leading and trailing threads. Secondly, even for cache lines that are shared among processors, input incoherence cannot occur unless one or more of the processors write to the cache line. The implication of these facts is that input incoherence is a problem only for cache lines that are shared and modified with other processors.

RECVF leverages these observations by tracking unverified shared cache lines in the memory hierarchy. Loads that read from unverified lines obtained from cache-to-cache transfers are not re-executed in the trailing core. Instead the leading core forwards the value that it reads to the trailing core. The trailing core uses these forwarded values without verification. Reads from other lines are fully re-executed in the trailing core.

Stores are fully re-executed and write the to private data caches. However, only the leading core writes back lines to the lower levels. Cache lines replaced by the trailing cores’ private caches are discarded. Writebacks are performed only when writing to verified cache lines, or when an owned and verified cache line is being replaced.

The trailing cores do not participate in any coherence bus transactions initiated by the leading cores. Read requests from the trailing cores’ caches are answered by either the lower level (shared L2 or main memory) or caches containing verified copies of the line. An operation executed by a trailing core does not change the state of any cache line in any of the leading cores. As mentioned previously, cache-to-cache transfers have to propagate the unverified bit.

To understand how RECVF ensures correct operation, consider the example timeline shown in Figure 3.8. At \( t_1 \), leading core 1 (\( LC_1 \)) reads from a memory location, bringing it into its cache. Assume that this cache line is not present in any other processor. At
$t_2$, $LC_2$ writes to this cache line. There are two effects of this: (1) the cache line is invalidated in $LC_1$, and (2) the cache line is marked unverified. Subsequently, at $t_3$, $TC_1$ (trailing core 1) reads from this cache line. This read fetches the verified copy of the cache line from L2/main memory. This is because leading cores do not respond to requests from trailing cores’ caches. Next at $t_4$, when $LC_3$ reads the same cache line, $LC_2$ responds by providing the cache line to it. Now, the cache line is marked as unverified in both $LC_2$ and $LC_3$. Therefore when $TC_3$ will execute the corresponding load instruction at $t_5$, it will not access the data cache and instead reads the value forwarded by $LC_3$.

At $t_7$, when $TC_2$ writes to the cache line, it invalidates the copy of the line in $TC_1$, but not in any of the leading cores. Even though, $TC_3$’s read occurs before $TC_2$ has written to the cache line, it is able to execute correctly because it uses the value forwarded from $LC_3$.

Correctness of PLR

To understand why PLR ensures correct execution of multithreaded workloads, let us analyse it as two separate cases.

Consider the operation of the leading cores. In this case, PLR’s coherence mechanism works similar to a conventional MOESI cache coherence protocol with one main difference. Unlike the conventional protocol, PLR does not writeback unverified lines to a lower level. Since all lines are eventually verified, this change does not affect correctness.

For the trailing cores, two conditions are necessary for correctness.

1. In the absence of soft errors, a dynamic load instruction in the trailing core should read the same value as the corresponding dynamic load instruction in the leading core.

If the dynamic load instruction in the leading core reads from an unverified line obtained from a cache-to-cache transfer, then its value would have been replicated to the trailing core for use without verification. In this case, the corresponding trailing core dynamic load instruction reads the same value trivially.
If the dynamic load instruction in the leading core reads from an unverified line not obtained from a cache-to-cache transfer, this implies that the write operation which made the line unverified originated from the same leading core. The trailing core will replicate this write during its execution, and hence obtain the correct value for the read operation.

If the dynamic load instruction in the leading core reads from a verified line, then the trailing core obtains the correct value from either its own cache, one of its peer caches or the lower level. A verified copy of the line is guaranteed to be present in at least one of these locations.

2. Trailing core writes should not interfere with leading core operation. This condition holds true because leading cores ignore invalidates generated by the trailing core caches.

Summary

To summarize, PLR partially replicates load instructions from the leading to the trailing core. Our results in §3.3.11 show that PLR fully re-executes more than 92% of all loads, indicating that the maximum loss in fault coverage of load instructions due to partial replication is bounded by 8%.

3.3.10 Fault Coverage

Since, RECVF is based on spatial redundancy [77], it can detect both transient and permanents faults. Specifically, this includes all soft errors and hard errors that result in diverging architectural updates across the cores. RECVF provides a high degree of coverage for processor control and execution logic. However, RECVF may not be able to cover all faults that occur in the cache coherence related circuitry because it does not redundantly access the memory hierarchy for unverified cache lines obtained from cache-to-cache transfers. Our results in §3.3.11 will demonstrate that PLR provides much higher fault coverage than RESEA for faults that occur in the memory access circuitry.
as it redundantly accesses the data cache for most load instructions in the trailing core.

### 3.3.11 Evaluation

**Methodology**

<table>
<thead>
<tr>
<th></th>
<th>Technology node</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong># of cores</strong></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td><strong>Nominal frequency</strong></td>
<td>3 GHz</td>
<td>Fetch/issue/retire</td>
</tr>
<tr>
<td><strong>ROB size</strong></td>
<td>128 instructions</td>
<td>Int/FP registers</td>
</tr>
<tr>
<td><strong>Integer/FP window</strong></td>
<td>64/32 instructions</td>
<td>Load/store queue</td>
</tr>
<tr>
<td><strong>Mem/Int/FP units</strong></td>
<td>4/6/4</td>
<td>I-cache</td>
</tr>
<tr>
<td><strong>D-cache</strong></td>
<td>64k/64B/4-way/2 cycles</td>
<td>Memory</td>
</tr>
<tr>
<td><strong>Branch target buffer</strong></td>
<td>4k entries, 4-way set-assoc.</td>
<td>Return address stack</td>
</tr>
<tr>
<td><strong>Branch predictor</strong></td>
<td>hybrid of bimodal/gshare</td>
<td>BOQ size</td>
</tr>
<tr>
<td><strong>Checkpointing interval</strong></td>
<td>50,000 instructions</td>
<td>Checkpointing latency</td>
</tr>
</tbody>
</table>

**Shared L2 configuration**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L2</strong></td>
<td>16 MB/64B/8-way/40 cycles</td>
<td>L2</td>
</tr>
<tr>
<td><strong>Interconnect latency</strong></td>
<td>24 cycles</td>
<td>Interconnect latency</td>
</tr>
<tr>
<td><strong>L2</strong></td>
<td>2 MB × 8/64B/8-way/24 cycles</td>
<td></td>
</tr>
</tbody>
</table>

**Private L2 configuration**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PCB size</strong></td>
<td>1024 entries</td>
<td></td>
</tr>
<tr>
<td><strong>PCB access hash table</strong></td>
<td>257 entries</td>
<td></td>
</tr>
<tr>
<td><strong>Hash table access latency</strong></td>
<td>1 cycle</td>
<td></td>
</tr>
<tr>
<td><strong>PCB sections</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PCB access latency</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration for CRT [60]</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LVQ size</strong></td>
<td>512 entries</td>
<td>BOQ size</td>
</tr>
<tr>
<td><strong>BOQ size</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>IRQ size</strong></td>
<td>512 entries</td>
<td></td>
</tr>
<tr>
<td><strong>Checkpointing latency</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.4: CMP configuration for CVF evaluation**

Our methodology is similar to that used for the RESEA configurations, details of which are given in §3.2.6.

We use a modified version version of the SESC execution-driven simulator. Power models are based on Wattch [19] and CACTI 4.1, while the shared L2 cache power is modeled used CACTI 5.3 [103]. As in §3.2.6 we compare our proposal’s performance with that of PVA [76] and CRT [60]. Power models are added for the BOQ, LVQ, IRQ and PCB. Details of the CMP model are shown in Table 3.4. The DVFS configuration is shown in Table 3.2.

We show results for two CMP configurations. One is a conventional CMP architecture
Chapter 3. Energy-Efficient Redundant Execution

with a 16 MB shared L2 cache. The second configuration has a 2 MB private L2 cache associated with each core. In the private-L2 configuration prefetch hints are forwarded from the leading core L2 cache to the corresponding trailing core L2 cache.

We only show results for the best performing critical value identification heuristic, \textit{fanout2}. We show results for the PLR and FLR configurations as well as the QSize and IPC DVFS algorithms.

**Workload**

Our workload comprises of ten integer and ten floating point benchmarks from the SPEC CPU 2000 suite. For each benchmark, a single simulation point of length one billion instructions is executed [85]. For parallel applications we used the SPLASH2 benchmarks [115] with the inputs suggested in [14].

**Shared L2: IPC Results**

![Figure 3.9: Normalised IPC for the Shared L2 Configuration](image)

Figure 3.9 shows the IPC of each of the benchmarks normalised by the IPC of the baseline processor. The mean IPC degradation of PVA is 4.65%. PVA loses performance
mainly due to high PCB occupancy. A fully occupied PCB can stall retirement in the leading core. CRT’s mean IPC degradation is 4.75%. In CRT, a store cannot retire from the leading core’s store buffer until it is verified by the trailing core. This creates additional pressure on the store buffer. As a result, CRT has a performance problem with *mesa* and *vortex*, both of which have a high fraction of store instructions.

The CVF configurations exhibit mean IPC degradation varying between 0.5% and 1.4%. The worst performing benchmark for CVF is *apsi*. *Apsi* is slowed down because of interconnect congestion during certain phases of the program execution.

**Shared L2: Energy Results**

![Figure 3.10: Normalised energy for the Shared L2 Configuration](image)

Figure 3.10 shows energy consumption normalised by the energy consumption of the baseline processor. PVA consumes 1.32 times the energy of the baseline, while CRT consumes 1.52 times the energy of the baseline. The CVF configurations using PLR consume 1.26 times the energy of the baseline. FLR is somewhat successful in trading-off lower fault coverage for reduced energy consumption with a mean energy consumption of 1.20 times the baseline.
To understand these results better, we present a breakup of the energy consumption in Figure 3.11.

![Figure 3.11: Component-wise breakdown of energy consumption](image)

We make the following observations from the figure. Firstly, as one would expect, the leading cores of each configuration dissipate roughly the same amount of energy. Secondly, the trailing core in CRT consumes more energy than the trailing cores in CVF and PVA, because the latter are running at lower voltage-frequency levels. A third interesting observation is that PVA consumes significantly higher energy in the L2 cache. This is because PVA stores verified lines in the L2 cache, which is effectively results in a write-through policy for the L1 cache.

**Shared L2: Multithreaded Workload Results**

Figure 3.12 shows Snavely and Tullsen’s weighted speedup metric [92] and normalised energy for the SPLASH2 benchmarks. In this section we only compare against CRT, because [76] does do not explain how to support parallel applications with PVA.

CRT has a performance degradation of about 10% for these benchmarks. The two configurations based on PLR have a performance degradation of about 3%, while the configurations based on FLR have a performance degradation of about 10%.

PLR has a higher performance overhead for the SPLASH2 benchmarks when compared to the SPEC benchmarks because all the cores of that are executing the application
Chapter 3. Energy-Efficient Redundant Execution

need to synchronize at the checkpoints. This increases the frequency of checkpointing, increasing the performance overhead.

The average energy consumption of CRT for these benchmarks is 1.7 times the baseline. The average energy consumption of PLR with the QSize-DVFS algorithm is 1.4 times the baseline. For PLR with the IPC-DVFS algorithm, the energy consumption is 1.5 times the baseline. FLR also consumes similar amounts of energy. The reduced dynamic power consumption is offset by increased energy losses due to leakage.

**Quantifying Coverage Loss due to PLR**: Table 3.5 shows the percentage of loads in the trailing core which obtain their values from the leading core instead of redundantly accessing the data cache. This is a measure of the loss in coverage due to partial load replication. On average, PLR redundantly accesses memory for more than 92% of the load instructions. These results show that PLR is able to achieve comparable fault coverage to input replication mechanisms like Reunion [91] and DCC [49] without the same level of complexity.
Table 3.5: Percentage of loads not fully re-executed in the trailing core due to PLR

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>QSize-DVFS</th>
<th>IPC-DVFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>cholesky</td>
<td>0.14%</td>
<td>0.14%</td>
</tr>
<tr>
<td>fft</td>
<td>0.20%</td>
<td>0.59%</td>
</tr>
<tr>
<td>fmm</td>
<td>7.93%</td>
<td>7.74%</td>
</tr>
<tr>
<td>ocean</td>
<td>2.02%</td>
<td>2.00%</td>
</tr>
<tr>
<td>radiosity</td>
<td>24.78%</td>
<td>25.52%</td>
</tr>
<tr>
<td>radix</td>
<td>0.92%</td>
<td>0.92%</td>
</tr>
<tr>
<td>raytrace</td>
<td>14.75%</td>
<td>14.01%</td>
</tr>
<tr>
<td>water-nsquared</td>
<td>5.35%</td>
<td>5.43%</td>
</tr>
<tr>
<td>water-spatial</td>
<td>10.15%</td>
<td>10.15%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>7.36%</strong></td>
<td><strong>7.39%</strong></td>
</tr>
</tbody>
</table>

Private L2: IPC Results

Figure 3.13 shows the normalised IPC for the private-L2 configuration. The mean IPC degradation for PVA with a single-ported PCB is 10.4%. **Vortex** and **sixtrack** are the worst-affected benchmarks, with IPC degradations of 22% and 19% respectively. This is due to increased PCB occupancy caused by higher interconnect latencies. **CRT** also performs poorly for this configuration exhibiting mean IPC degradation of 9.3%. This is because of increased occupancy of the store buffer. On an average, store buffer occupancy for CRT is 2.2 times the store buffer occupancy for the baseline processor. Compared to the results for the shared L2 configuration in §3.3.11, the problem here is exacerbated by higher interconnect latencies. The architectures based on CVF have a much lower mean IPC degradation for this configuration, varying between 2.2% and 3.9%.

An interesting pathological behavior is displayed by **wupwise**. For this particular benchmark, PVA has almost no IPC degradation while CRT and the configurations based on critical-value-forward exhibit an IPC loss varying between 6.0% and 10%. **Wupwise** has a very high L2 miss rate of about 90%. In other words, the vast majority of the
misses in the L1 data cache also turn out to be misses in the L2 cache. Consequently, the latency of PCB lookups in the trailing cores is fully hidden by the large L1 miss latency. As a result of this, the IPC degradation due to long-latency PCB lookups, seen in the other benchmarks for PVA, does not occur for \textit{wupwise}. This why PVA is the configuration with the least IPC degradation for \textit{wupwise}.

**Private L2: Energy Results**

Figure 3.14 shows the energy dissipation for the private L2 configuration. It is apparent that PVA and CRT dissipate much more energy than CVF. PVA consumes 1.62 times the energy of the baseline processor. CRT dissipates 1.92 times the energy of the baseline. CVF consumes 1.45 times the energy of baseline processor for the PLR configuration. FLR configurations consume 1.39 times the energy of the baseline.

**Bandwidth Requirements**

Figure 3.15 shows the average core-to-core bandwidth required by each scheme in units of values per cycle. For PVA this includes the bandwidth consumed by verifying stores,
PCB lookups and invalidation messages, but does not include the bandwidth required for the additional writes performed to the L2 cache.

The PLR based design for CVF has the lowest bandwidth requirement, while CRT and FLR have the two highest requirements. PVA has a bandwidth requirement that is slightly higher than that of CVF.

Comparing Figure 3.15 with Figure 3.5, it is evident that CVF requires only one-third the bandwidth of RESEA+EarlyWrite. This decrease in bandwidth comes at no performance or energy cost.

**Sensitivity to DVFS Latencies**

Our baseline architecture uses fast fine-grained DVFS similar to the proposal in [43] re-evaluating trailing core voltage-frequency level every 1 $\mu$s. In this section we evaluate the impact of more conservative per-core DVFS implementations.

Figure 3.16 shows the normalised IPC and energy values averaged across all the benchmarks for a number of DVFS configurations. These results are for the shared L2 configuration. The x-axis shows the DVFS update latency and update interval, while
Figure 3.15: Bandwidth requirements for RECVF

Figure 3.16: Impact of the higher-latency and coarse-grained DVFS.
Chapter 3. Energy-Efficient Redundant Execution

the two y-axes show normalised IPC and energy. For example, “0.1 ms/1 ms” means switching between voltage-frequency levels takes 0.1 ms, and voltage-frequency levels of the trailing core are re-evaluated every 1 ms.

IPC decreases with increasing DVFS update intervals. This is because longer update intervals find it harder to track fine-grained changes in program phase. However, energy consumption remains roughly constant across all the DVFS configurations.

The IPC-DVFS algorithm suffers a performance loss of 9% at a 1 ms update interval. In contrast, for the QSize-DVFS algorithm even with a very conservative DVFS architecture that updates the voltage-frequency level every 1 ms, the mean IPC degradation is only 4% and the increase in energy dissipation is restricted to a few percent. This is an important result, showing that CVF is applicable even for much more conservative per-core DVFS implementations.

Sensitivity to Reduced Voltage Scaling

![Figure 3.17: Impact of limited voltage scaling](image)

Both our proposal and PVA rely on voltage and frequency scaling in order to reduce energy dissipation. Aggressive voltage scaling might prove to be difficult in future technology nodes. Figure 3.17 shows the impact of reduced voltage scaling for the private L2 configuration.
PVA’s energy dissipation increases by 9.1%. CVF’s PLR+QSize configuration’s energy dissipation increases by only 6.8%. None of the CVF configurations show an increase of more than 8.25%.

The reason CVF scales better than PVA is due to increased leakage power. PVA dissipates more leakage power than CVF due to the fact that it uses three cores for execution. Dynamic power scales quadratically with voltage. We use the Butts and Sohi’s first-order model of leakage power [20, 116], which suggests that leakage decreases linearly with a decrease in supply voltage. Therefore, since dynamic power decreases more than static power, the contribution of leakage power as a fraction of total power increases.

Sensitivity to Queue Sizes

Figure 3.18 shows the impact of BOQ and the IRQ sizes on CVF’s performance. There are five bars shown in the figure. Each bar is labelled as “BOQ size/IRQ size”.

![Figure 3.18: Impact of queue sizes](image)

The results show that CVF performs well even with extremely small queue sizes, a 32-entry BOQ and 64-entry IRQ. The results in this thesis use 64-entry BOQ and
512-IRQ; this size minimizes energy-delay squared ($ED^2$) metric.

Sensitivity of QSize-DVFS to Threshold Selection

Figure 3.19 shows the impact of the thresholds used in the QSize-DVFS algorithm on the performance of the CVF. Each bar is labelled as ‘BOQLow:BOQHigh/IRQLow:IRQHigh’. The values are the low and high threshold values for the BOQ and the IRQ structures respectively.

![Figure 3.19: Impact of thresholds on the QSize algorithm’s performance](image)

Although the variation of IPC and energy with threshold values is somewhat complex, two important features of this variation are evident. Firstly, the selection of threshold has only a small impact on the performance of the algorithm. The difference in performance overhead between the best and the worst threshold values is less than 2%; the difference in energy consumption is less than 1%. Secondly, in general, larger threshold values have lower energy consumption at the cost of a higher performance overhead. The reason for this is that a lower threshold value reacts faster to potential changes of program phase which require the trailing core to switch from low to high frequencies. This leads to
better performance for smaller threshold values. However, some of these switches might be unnecessary, leading to higher energy consumption.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Highest Normalized IPC</th>
<th>Corresponding Threshold</th>
<th>Lowest Normalized IPC</th>
<th>Corresponding Threshold</th>
<th>Percentage Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>sixtrack</td>
<td>0.92</td>
<td>4:8/32:64</td>
<td>0.81</td>
<td>4:32/32:256</td>
<td>13.57%</td>
</tr>
<tr>
<td>apsi</td>
<td>0.85</td>
<td>4:8/16:32</td>
<td>0.80</td>
<td>8:32/64:256</td>
<td>6.43%</td>
</tr>
<tr>
<td>gcc</td>
<td>0.97</td>
<td>4:8/16:32</td>
<td>0.94</td>
<td>8:32/64:256</td>
<td>3.23%</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.97</td>
<td>4:8/16:32</td>
<td>0.94</td>
<td>8:32/64:256</td>
<td>2.88%</td>
</tr>
<tr>
<td>ammp</td>
<td>0.98</td>
<td>4:8/16:32</td>
<td>0.95</td>
<td>8:32/64:256</td>
<td>2.80%</td>
</tr>
<tr>
<td>vortex</td>
<td>0.92</td>
<td>4:8/16:32</td>
<td>0.90</td>
<td>4:32/32:256</td>
<td>2.77%</td>
</tr>
<tr>
<td>mesa</td>
<td>0.98</td>
<td>4:8/16:32</td>
<td>0.96</td>
<td>4:32/32:256</td>
<td>2.38%</td>
</tr>
<tr>
<td>crafty</td>
<td>0.99</td>
<td>4:8/16:32</td>
<td>0.96</td>
<td>8:32/64:256</td>
<td>2.31%</td>
</tr>
</tbody>
</table>

Table 3.6: Comparison of best and worst normalised IPC values for each benchmark across different thresholds.

Table 3.6 compares the best and worst performing thresholds for a subset of the SPEC CPU 2000 benchmarks. The benchmarks not shown in the table did not show a variation of more than 2% between the best and worst performing thresholds. Sixtrack is most sensitive to threshold selection, showing 13.5% variation in performance between the best and worst cases while Apsi shows 6.4% variation. The remaining eighteen benchmarks show less than 3.5% variation between the best and worst cases, indicating that the algorithm is relatively insensitive to threshold selection.

Some more insight about this behaviour can be gleaned by observing the number of cycles spent in each frequency level by the trailing cores for the best and worst-performing threshold configurations. This is shown in Figure 3.3.11. Bzip2 is the application for which critical value forwarding is the least effective. As a result, it spends a significant amount of time in the higher frequency levels. The fastest threshold configuration for bzip2 spends more time in the higher frequency levels than does the slowest threshold configuration. Similar behaviour is seen in gcc and sixtrack.

In contrast, swim, twolf and vpr spend most of their time in the lowest frequency
Figure 3.20: Comparison of Cycles Spent at Each Frequency Level For the Trailing Core.
levels. Therefore, the choice of thresholds has little effect on the performance of these applications.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Highest Normalized Energy</th>
<th>Corresponding Threshold</th>
<th>Lowest Normalized Energy</th>
<th>Corresponding Threshold</th>
<th>Percentage Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>apsi</td>
<td>1.53</td>
<td>4:8/16:32</td>
<td>1.45</td>
<td>8:32/64:256</td>
<td>5.83</td>
</tr>
<tr>
<td>vortex</td>
<td>1.53</td>
<td>4:8/16:32</td>
<td>1.48</td>
<td>8:32/64:256</td>
<td>3.44</td>
</tr>
<tr>
<td>ammp</td>
<td>1.49</td>
<td>4:8/16:32</td>
<td>1.46</td>
<td>8:32/64:256</td>
<td>2.19</td>
</tr>
<tr>
<td>gcc</td>
<td>1.46</td>
<td>4:8/16:32</td>
<td>1.43</td>
<td>8:32/64:256</td>
<td>2.05</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.66</td>
<td>4:8/16:32</td>
<td>1.63</td>
<td>8:32/64:256</td>
<td>1.61</td>
</tr>
<tr>
<td>crafty</td>
<td>1.48</td>
<td>4:8/16:32</td>
<td>1.46</td>
<td>8:32/64:256</td>
<td>1.36</td>
</tr>
<tr>
<td>sixtrack</td>
<td>1.44</td>
<td>4:8/16:32</td>
<td>1.42</td>
<td>8:32/64:256</td>
<td>1.33</td>
</tr>
</tbody>
</table>

Table 3.7: Comparison of best and worst normalised energy values for each benchmark across different thresholds.

Table 3.7 shows the variation in energy dissipation for the benchmarks across the threshold values. Only benchmarks which show more than 2% variation are shown in this figure. The maximum variation of 5.8% is shown by apsi. The remaining 19 benchmarks show less than 3.5% variation.

**Intra-die Variation: Performance With “Slow” Cores**

Our proposals can take advantage of within-die process variations in CMPs. For instance, Humenay et al. [37] find a 17% difference between the fastest and slowest cores in a CMP. The architectures we develop in this chapter can utilise the slower cores of a CMP more efficiently by designating them as trailing cores for redundant execution. Since execution assistance mechanisms can accelerate the trailing cores, their slower operation will not result in a significant slowdown of the application itself.

Figure 3.3.11 shows the performance of RECVF when the trailing core frequency is limited to a maximum of 2.7 GHz and 2.4 GHz. Details of the voltage and frequency levels for these experiments are shown in Table 3.8.
Table 3.8: Voltage-frequency Levels of Trailing Core For Evaluation of Performance with Within-die Variation. Note that the leading core is operated at full frequency (3.0 GHz).

<table>
<thead>
<tr>
<th>Frequency (GHz)</th>
<th>Voltage (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0 GHz max</td>
<td>1.0 0.9 0.8 0.7 0.6 0.6</td>
</tr>
<tr>
<td>2.7 GHz max</td>
<td>1.0 0.9 0.8 0.7 0.6</td>
</tr>
<tr>
<td>2.4 GHz max</td>
<td>1.0 0.9 0.8 0.7</td>
</tr>
</tbody>
</table>

The results show that there is minimal performance degradation, but a significant increase in energy consumption. As most of time the trailing cores operate at lower than the max frequency, limiting the max frequency does not impact performance much. However, the “slow” cores operate at a higher voltage level for a given frequency, so they consume more energy.

### 3.3.12 Discussion

Our evaluation shows that CVF has a performance overhead of less than 1.2% for a shared-L2 CMP and consumed only 1.26 times the energy of the baseline processor. For a future CMP with higher latency interconnects and private L2 caches, CVF has a performance overhead of less than 4.0% and consumed 1.45 times the energy of the baseline processor. Our comparison of CVF to two previous proposals for fault-tolerant CMPs finds that CVF delivers higher energy-efficiency and lower performance degradation than either of the proposals.

CVF also improves significantly upon the RESEA configurations proposed in §3.2. For the PLR configurations, CVF is able to deliver very slightly better energy-efficiency and similar performance to RESEA at 1/3rd the bandwidth cost. For the FLR configurations, CVF is able to deliver higher energy-efficiency and similar performance for similar bandwidth costs as RESEA. These results illustrate that CVF utilises on-chip bandwidth more efficiently than previous proposals for execution assistance.
3.4 Using Cores with Faulty Functional Units

CVF incorporates mechanisms to transfer the results of instructions from the leading core to the trailing core. These mechanisms can be reused to permit the use of cores with faulty functional units similar to [67].

Assume that one of the cores of a DMR pair has faulty functional units. CVF designates the faulty core as the trailing core and the fully functional core as the leading core. Instructions which cannot be executed on the faulty core are executed on the leading core, and their results are forwarded to the trailing core. For transient fault detection, a redundant copy of the instruction is executed in the leading core and the results of the two copies are compared. This scheme is especially effective for functional units which consume a large amount of area, but are infrequently used, like the floating point units.

The details of the mechanism are as follows. At the time of instruction decoding in the leading core, the core identifies instructions which cannot be executed in the trailing core. To provide transient fault coverage, the leading core inserts a redundant
copy of the instruction and a comparison instruction into a structure know as the re-
execution queue. The comparison instruction ensures that the redundant instruction
and the original instruction produce the same result, protecting against transient faults.
The re-execution queue is a small structure with only four entries. It is used to issue
the redundant instruction and the comparison instruction immediately after the original
instruction is issued. At the time of instruction retirement, the original instruction’s
result is forwarded to the trailing core. Like with other forwarded values, the trailing
core writes this result into the register at the time of instruction dispatch. However,
the trailing core does not execute the instruction redundantly, and instead retires the
instruction using the value forwarded by the leading core.

3.4.1 Results

![Normalized IPC and Energy](image)

Figure 3.22: Normalised IPC and energy with faulty FP units in the trailing core.

Figure 3.22 shows normalised IPC and energy of the SPEC floating point benchmarks
when executing in the presence of faulty floating point (FP) units. We consider two cases:
(1) a faulty FP divider and (2) where the FP multiplier as well as divider are faulty.

The divider unit is used rarely as it executes FP division and square root instructions.
In the case when it is faulty, the performance overhead over non-redundant execution is
only 7%. In comparison, when redundant execution is performed with fault-free cores,
the performance overhead is about 1%. The energy consumption with a faulty divider is 1.29 times the baseline non-redundant system. This is only 2% more than the energy consumption of a redundant system with fault-free functional units. When both the multiplier and divider are faulty, the performance overhead is 29%. The energy consumption is 1.49 times the baseline.

These results are promising and show that that CVF can provide energy-efficient redundant execution even in the presence of permanent faults in one of the cores with only modest performance and energy costs.

3.5 Limitations

Our proposal for the use of faulty cores makes the assumption that faulty functional units can be identified at the time of manufacturing test [88]. Secondly, it is restricted to failures occur in faulty functional units. For instance, the present form of the scheme cannot be used to handle faults that occur in the control logic of a microprocessor.

In terms of implementation complexity, our instruction-mutation scheme is similar to microcode patches which are used in current microprocessors to work around design bugs [109]. However, our mechanism imposes an additional task on the OS to track the functionality of each core, and appropriate schedule and configure cores for the execution of fault-tolerant applications.

3.6 Concluding Remarks

This chapter presented two proposals for energy-efficient fault tolerance in future multi-core processors.

The first architecture, called redundant execution using simple execution assistance (RESEA), is based on forwarding branch outcomes and load values from the leading to the trailing core and combined this with per-core DVFS for energy-efficient redundant execution. We also introduced the early-write optimisation, a technique that significantly improved the effectiveness of simple execution assistance. Our results showed
that RESEA had a performance overhead of about 1% and consumed about 1.26 times the energy of non-redundant execution.

Our second architecture is called redundant execution using critical value forwarding (RECVF). RECVF improved upon RESEA by providing the same performance at a fraction of the inter-core communication bandwidth cost. This is an important result, because interconnects are expected to be bottlenecks in future processors [45]. Lower bandwidth requirements imply that RECVF can be used in the context of NoC-based interconnection fabrics. RECVF introduced a new form of execution assistance, called critical value forwarding, which identifies instructions on the critical path of execution and forwards the results of these to the trailing core. By focusing on critical instructions, CVF are able to obtain most of the speedup associated with forwarding all instructions at a fraction of the bandwidth cost.

RECVF combined the idea of critical value forwarding with that of per-core DVFS to operate the trailing core at a lower voltage/frequency level. We introduced and evaluated two design options for input replication in RECVF, partial load replication (PLR) and full load replication (FLR). We also studied RECVF’s performance for two new per-core DVFS algorithms. Our results showed that RECVF with PLR has a performance overhead of about 1% and energy consumption of about 1.26 times that of non-redundant execution. These results are achieved at a fraction of the bandwidth cost of RESEA. If interconnect bandwidth is not a bottleneck, then RECVF with FLR had a performance overhead of about 1% and energy consumption that about 1.20 times that of non-redundant execution. Thus, RECVF+FLR has better energy-efficiency than RESEA at approximately the same bandwidth cost.

We compared both RESEA and RECVF to existing proposals for fault-tolerant CMPs, the parallelized verification architecture (PVA) of Rashid et al. [76] and Chip-level Redundant Threading introduced by Mukherjee et al. [60]. We found that RESEA and RECVF had a lower performance degradation and higher energy-efficiency than both of the earlier proposals.

We also introduced an extension to RECVF that allows it operate using cores with
fault functional units. We studied a simple case where the trailing cores’ floating point units are faulty. Our evaluation showed that RECVF can operate correctly with modest performance degradations even for FP-intensive benchmarks.
Chapter 4

Multiplexed Redundant Execution

4.1 Introduction

The era of decreasing CMOS reliability has resulted in a number of proposals [6, 29, 29, 30, 32, 49, 52, 60, 75, 76, 90, 91, 97, 98, 100, 101] that take advantage of inherently replicated hardware resources in CMPs to provide fault tolerance. Typically, these schemes use some form of space redundancy where two cores or thread contexts of a CMP are used to execute a single logical thread. Inputs to the two cores are replicated and the outputs generated by the two cores are compared to detect the occurrence of errors.

The use of two cores or thread contexts to execute a single program means that the throughput of the CMP is reduced by half. This results in a throughput loss because a fault-tolerant system must have twice as many cores to achieve the same throughput as an equivalent non-redundant system. Besides increasing the procurement cost of systems, throughput loss also results in increased cooling costs, increased energy consumption and a higher maintenance cost. Therefore, the cost of operation of a fault-tolerant system is significantly higher than an equivalent non-redundant system. Such high costs are undesirable for fault-tolerant general purpose microprocessors for the commodity market. Therefore, there is a need for fault-tolerant architectures that can minimize this throughput loss.

In this chapter, we present Multiplexed Redundant Execution (MRE), a technique that reduces the throughput loss due to fault tolerance. MRE employs the well
known technique of space redundancy [77], where two copies of a program are executed on different cores of a CMP with replicated inputs. The outputs of these two streams of execution are compared to detect faults. MRE is based on the observation that execution assistance can accelerates the execution of the trailing thread in a fault-tolerant system. This frees execution bandwidth that can be used for executing other threads or applications.

Based on this observation, MRE uses coarse-grained multithreading to schedule several trailing threads on a single processor core. Therefore, unlike previous work, where each leading/trailing thread combination requires two cores for execution, we are able to multiplex trailing threads on a single trailing core with only a small performance penalty as compared to non-redundant execution.

Our evaluation shows that MRE increases the throughput of a fault-tolerant CMP by 9 – 13% as compared to previous proposals for fault-tolerant CMPs. This increase in throughput comes at a modest cost in single-threaded performance, a mean slowdown that varies between 11–14%.

### 4.2 Conceptual Overview

![Figure 4.1: Conceptual block diagram of MRE](image)

As is shown in Figure 4.1, MRE partitions the processors of a CMP into different pools of cores. One pool is the set of leading cores. These cores execute the leading threads
of applications that require fault tolerance. A second pool consists of the set of trailing cores. These cores execute trailing threads of applications which require fault tolerance. A third pool of processors executes non-redundant applications. The classification of processor cores into leading, trailing and non-redundant cores is only a logical distinction. Physically, all cores are identical and unused structures are appropriately disabled based on the mode of operation.

This partitioning is similar to traditional fault tolerance schemes for CMPs except that the pool of trailing cores is allowed to be smaller than the pool of leading cores. In this case, a single trailing core executes multiple trailing threads. These trailing threads are executed concurrently using coarse-grained multithreading.

Execution of the application is carried out in chunks. The leading core executes a chunk of instructions and sends a message to the trailing core requesting the execution of the corresponding chunk. Upon the receipt of a request to execute a chunk, the trailing core pushes this request into a run request queue (RRQ). When the current chunk executing in the trailing core completes execution, the trailing core (if necessary) switches contexts and executes the next request from the head of the RRQ.

### 4.3 Execution Assistance in MRE

MRE relies on the technique of execution assistance to accelerate execution in the trailing core. For this, we explore the two mechanisms for execution assistance that were introduced in chapter 3.

1. **Simple Execution Assistance** forwards the results of load values and branch outcomes from the leading core to the trailing core. We explore both baseline SEA as well as SEA with the early-write optimisation (see §3.2.1).

2. **Critical Value Forwarding** identifies instructions on the critical path of execution in the leading core and forwards the results of these to the trailing core (see §3.3). Depending on the design option used for input replication, this mechanism
Chapter 4. Multiplexed Redundant Execution

either provides similar speedup at a much lower bandwidth cost (PLR) or higher speedup at a similar bandwidth cost (FLR).

4.4 Design of an MRE Core

This section describes the operation of MRE-enabled processor cores.

4.4.1 MRE+SEA Core

Figure 4.2(a) shows an MRE core that uses simple execution assistance. The operation of this core is similar to the operation of the RESEA core presented in §3.2. The main differences are:

- Per-thread BOQ and per-thread LVQ structures.
- A new microarchitectural structure called the Run Request Queue.

The per-thread BOQ is accessed at the time of instruction fetch and contains branch outcomes forwarded from the leading core. The BOQ is used instead of the branch predictor in the trailing core.

1See §3.3.4 for a discussion of these options.
The per-thread LVQ contains load values forwarded from the leading core. In the baseline MRE+SEA system, the LVQ is accessed after the load instruction’s effective addresses is computed. In the case of an MRE+SEA system with the early-write optimisation, the LVQ is accessed at the time of instruction issue.

4.4.2 MRE+CVF Core

Figure 4.2(b) shows an MRE core that uses critical value forwarding. This core is analogous to the RECVF core from §3.3 in the same way the MRE+SEA is analogous to the RESEA core. Again, there are two main differences from the RECVF core.

- Per-thread BOQ and per-thread IRQ structures.
- A new microarchitectural structure called the Run Request Queue.

The per-thread BOQ is accessed at the time of instruction fetch and contains branch outcomes forwarded from the leading core. If a branch outcome is present in the BOQ, then it overrides the output of the branch predictor.

The per-thread IRQ is accessed at the time of instruction issue. If an instruction’s result is present in the IRQ, then this result is written into the instructions designation physical register immediately. This allows dependent instructions to begin execution earlier, accelerating trailing core execution by breaking data dependence chains.

4.4.3 Run Request Queue

The trailing core uses coarse-grained multithreading to execute multiple trailing threads. Execution of an application is carried out in chunks. Each time the leading core executes a certain number of instructions, it sends a request to the trailing core to execute the corresponding chunk. This request is enqueued in the run request queue (RRQ) in the trailing core. When the trailing core finishes the execution of a chunk, it executes the next chunk from the head of the RRQ. If necessary, a context switch is performed. Unless a fingerprint comparison is to be made (described in §4.5), the leading core continues execution after signalling the end of a chunk.

Requests are enqueued in the RRQ in the order in which they are received. The
default scheduling mechanism in MRE switches between threads in the order in which requests are enqueued in the RRQ. This ensures fairness and forward progress for all threads.

We also investigate a mechanism which gives priority to stalled threads, i.e., leading threads which are waiting for the completion of a fingerprint exchange (see §4.5). A quantitative comparison of these mechanisms is shown in §4.6.8.

### 4.5 Fault Tolerance Mechanisms

Any fault-tolerant system needs to address three important issues: fault detection, fault isolation and fault recovery.

The occurrence of faults is detected by MRE through the mechanism of fingerprint comparison [90]. A fingerprint creates a CRC-based hash of the architectural updates of each core, every cycle. The hash values of the two cores are exchanged and compared to detect errors. Note that the trailing core will have to maintain separate fingerprint registers for each thread. If no error occurs, the sequence of retired instructions and architectural register file updates will be identical between each leading/trailing thread pair, guaranteeing that the fingerprints will be equal.

MRE’s fault isolation mechanism is similar to the mechanisms used by RESEA and RECVF. The L1 data cache tracks unverified lines in order to prevent them from escaping the sphere of replication. The details of these mechanisms may be found in §3.2.3 and §3.3.8.

MRE’s fault recovery mechanism involves (1) restoring the register state from a check-point store and (2) restoring memory state by invalidating unverified lines in the L1 data cache. The details are the same as the mechanisms used by RESEA and RECVF.

### 4.6 Evaluation

This section details the workloads, metrics and methodology that we used to evaluate the effectiveness of MRE.
Chapter 4. Multiplexed Redundant Execution

<table>
<thead>
<tr>
<th>Category</th>
<th>Speedup</th>
<th>Benchmarks</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>low speedup</td>
<td>&lt; 1.6</td>
<td>bzip2, vortex, mesa</td>
<td>3</td>
</tr>
<tr>
<td>medium speedup</td>
<td>1.6 – 2.0</td>
<td>art, sixtrack, apsi, crafty, gcc, ammp</td>
<td>6</td>
</tr>
<tr>
<td>high speedup</td>
<td>&gt; 2.0</td>
<td>parser, twolf, gap, vpr, mgrid, gzip, applu, mcf, wupwise, swim, equake</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 4.1: Classification of benchmarks by speedup

<table>
<thead>
<tr>
<th>Category</th>
<th>Workloads</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>low-low</td>
<td>bzip2_vortex</td>
<td>1</td>
</tr>
<tr>
<td>low-med</td>
<td>mesa_crafty</td>
<td>1</td>
</tr>
<tr>
<td>low-high</td>
<td>bzip2_applu, vortex_mgrid</td>
<td>2</td>
</tr>
<tr>
<td>med-med</td>
<td>ammp_art, crafty_sixtrack</td>
<td>2</td>
</tr>
<tr>
<td>med-high</td>
<td>apsi_twolf, ammp_vpr, gap_crafty</td>
<td>3</td>
</tr>
<tr>
<td>high-high</td>
<td>swim_equake, gzip_mcf, twolf_parser, vpr_swim</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.2: Multiprogram workloads

4.6.1 Workload Construction

We constructed a set of 2-program workloads out of the SPEC CPU 2000 benchmark suite. The workloads were constructed by examining the results shown in §3.3.11 and dividing the benchmarks into three classes based on the speedup due to critical value forwarding. These classes are shown in Table 4.1.

Using the information in Table 4.1, we constructed the thirteen workloads shown in Table 4.2. The number of benchmarks in each category in Table 4.2 is approximately proportional to the product of the number of benchmarks in the corresponding categories in Table 4.1. Note that the counts are biased towards the categories with low and medium speedup. Hence, our results may be pessimistic estimates of the benefits of the multiplexing.
4.6.2 Evaluation Metrics

We use two metrics to evaluate the performance of multiplexing. The first is the weighted speedup metric introduced by Snavely and Tullsen [92].

The weighted speedup of a multithreaded or multiprogrammed workload measures the average of the speedup (or slowdown) experienced by each component of the workload. Weighted speedup is a fair metric to measure the performance of multiprogrammed or multithreaded workloads unlike, for example, the sum of the IPCs metric. The sum of IPCs metric can be optimised in an unfair mechanism by avoiding allocation of execution resources to the challenging parts of the workload (i.e., the low IPC threads).

\[
WtSpeedup = \text{Average}\left\{ \frac{IPC_{\text{redundant}}(i)}{IPC_{\text{non-redundant}}(i)} \right\}
\]

To quantify the reduction in throughput loss we introduce the metric normalised throughput per core (NTPC). The normalised throughput of a single thread is defined as the IPC of that thread when running in redundant mode divided by the IPC of that thread when running in non-redundant mode. The normalised throughput per core of a workload is defined as the sum of the normalised throughputs of the threads comprising the workload divided by the total number of cores the workload is running on.

\[
NTPC = \frac{1}{N_{\text{cores}}} \sum_{i=1}^{N_{\text{threads}}} \frac{IPC_{\text{redundant}}(i)}{IPC_{\text{non-redundant}}(i)}
\]

For an ideal dual modular redundant system which suffers no performance overhead, the normalised throughput would be 0.5. A real DMR system will always have some overheads due to communication and comparison, reducing the NTPC to less than 0.5.

NTPC is the same as the weighted speedup metric scaled by the number of cores being used.
4.6.3 Methodology

We used a modified version of the SESC execution-driven simulator [80]. We used the workloads listed in Table 4.2. We skipped the first three billion instructions of each component benchmark and then executed a total of one billion instructions. Details of the CMP model are shown in Table 4.3.

The total context switch penalty in the trailing core is 18 cycles. It is modeled as three components. A two cycle penalty is modeled for redirecting the fetch mechanism. Renaming is stalled by eight cycles to restore the register state of the new thread. Retirement is stalled by eight cycles to save the register state of the old thread.

| # of cores | 8 |
| Nominal frequency | 3 GHz |
| ROB size | 128 instructions |
| Integer/FP window | 64/32 instructions |
| Mem/Int/FP units | 4/6/4 |
| D-cache | 64k/64B/4-way/2 cycles |
| Branch target buffer | 4k entries, 4-way set-associative |
| Branch predictor | hybrid of bimodal/gshare |
| 16k entries in each predictor | per-thread BOQ size |
| Checkpointing interval | 50,000 instructions |
| L2 | 16 MB/64B/8-way/40 cycles |
| Technology node | 32 nm |
| Fetch/issue/retire | 4/4/4 instructions per cycle |
| Int/FP registers | 160/128 |
| Load/store queue | 32 instructions |
| I-cache | 32k/64B/4-way/2 cycles |
| Memory | 400 cycles |
| Return address stack | 32 entries |
| per-thread BOQ size | 512 entries |
| per-IRQ size (PLR/FLR) | 512/1024 entries |
| Checkpointing latency | 64 cycles |
| Interconnect latency | 24 cycles |

Table 4.3: CMP configuration for MRE evaluation

We show results for six configurations.

1. **CRT-4**: Four cores are used to execute two programs redundantly on a CRT processor [60].

2. **CRT-3**: Three cores are used to execute two programs redundantly on a CRT processor. The third core in this asymmetric configuration uses simultaneous multithreading (SMT) to multiplex the two trailing threads.

3. **MRE-3-SEA**: Three cores are used to execute two programs redundantly using
MRE. Simple execution assistance is used and multiplexing is done through coarse-grained multithreading.

4. **MRE-3-SEA+EarlyWrite**: This is the same as MRE-3-SEA except that the early-write optimisation is enabled for this configuration.

5. **MRE-3-Fanout2-PLR**: Three cores are used to execute two programs redundantly. Critical value forwarding using the fanout2 heuristic is the execution assistance mechanism. The configuration uses partial load replication (PLR) and coarse-grained multithreading.

6. **MRE-3-Fanout2-FLR**: This is the same as MRE-3-Fanout2-PLR except that full load replication (FLR) is used.

### 4.6.4 Results: Weighted Speedup

![Graph showing weighted speedup for MRE configurations](image)

**Figure 4.3**: Weighted speedup for MRE
Figure 4.3 shows the weighted speedup for each of the configurations examined. CRT-4 has the least mean slowdown of about 10%. CRT-3 has a slowdown of about 21%, while MUX-3-SEA has a slowdown of about 24%. Even though MRE uses only 3 cores for execution, while CRT-4 uses 4 cores, the mean slowdown of MRE-3-Fanout2-FLR (11%), MRE-3-Fanout2-PLR (14%) and MRE-3-SEA+EarlyWrite (15%) are all comparable to CRT-4’s slowdown of 10%. This is an important result and shows that multiplexing is effective in increasing CMP throughput with only a modest single-thread performance penalty.

### 4.6.5 Results: Normalised Throughput Per Core

Figure 4.4: Normalised throughput per core for MRE

Figure 4.4 shows the normalised throughput per core for each of the configurations. CRT-4 has the least throughput of 0.452, while MRE-3-Fanout2-FLR has the highest throughput of about 0.593. MRE-3-Fanout2-PLR and MRE-3-SEA+EarlyWrite have
comparable throughputs of 0.569 and 0.574 respectively. For any multiplexing mechanism that executes two programs redundantly on three cores, the maximum NTPC is 0.67. MRE-3-Fanout2-FLR is able to reach within 11% of this limit.

CRT-3 has a throughput of 0.526, which is higher than CRT-4, indicating that although CRT-3’s single-thread performance suffers, it provides higher throughput than CRT-4. The three multiplexing configurations, MRE-3-SEA+EarlyWrite, MRE-3-Fanout2-FLR and MRE-3-Fanout2-PLR, have significantly higher throughput than CRT-3. CRT-3 uses SMT for multiplexing, while MRE uses coarse grained multithreading. This result substantiates our claim that MRE provides high-throughput redundancy at a low cost because the implementation cost and complexity of SMT is much higher than that of coarse-grained multithreading.

4.6.6 Sensitivity to Queue Sizes

![Figure 4.5: Sensitivity of MRE+CVF to queue sizes](image)

Figure 4.5 shows the variation of weighted speed of MRE with BOQ and IRQ sizes. The bars in the figure are labelled as ‘per-thread BOQ Size/per-thread IRQ Size’.

The results show that PLR’s performance does not change at all for the three queue size configurations that we studied. However, FLR’s performance decreases by more
than 20% for the 512 entry IRQ when compared to the 1024 entry IRQ.

4.6.7 Sensitivity to Execution Chunk Size

MRE’s trailing thread is executed in chunks. The size of the execution chunk can have an impact on the performance of MRE. Larger chunk sizes perform fewer context switches. On the other hand, smaller chunk sizes have the advantage of reducing the slack between the leading and the trailing threads.

Figure 4.6 shows the impact of different chunk sizes on each of the MRE+CVF configurations. These simulations use infinite-sized BOQ andIRQ. The results show that for PLR, the best execution chunk size is either 512 instructions or 1024 instructions. However, for FLR the 2048-instructions execution chunk size provides the best performance.

4.6.8 Priority-Based Scheduling Algorithm

Figure 4.7 compares the performance of the baseline and the priority based scheduling algorithms. The priority based algorithms attempts to favour stalled leading threads
Figure 4.7: Comparison of baseline and priority-based scheduling algorithms

(i.e., threads waiting for fingerprint comparison). Surprisingly, this algorithm performs slightly worse than the baseline algorithm.

4.7 Concluding Remarks

This chapter introduced the technique of multiplexed redundant execution (MRE). Multiplexing is a mechanism that reduces the cost of fault tolerance by increasing the throughput of fault-tolerant CMPs. MRE exploits the technique of execution assistance to accelerate the execution of trailing threads and executes multiple trailing threads on a single processor core. Trailing threads are multiplexed using the low-cost technique of coarse-grained multithreading.

Our results have shown that throughput increases of up to a maximum of 30%, with mean increases varying between 14–19%, are possible due to MRE when compared
to a perfect DMR system. MRE also provides higher throughput and lower single-thread performance degradation when compared to a CRT [60] configuration that uses simultaneous multithreading to multiplex trailing threads. Since MRE uses only coarse-grained multithreading, this substantiates our claim that MRE provides higher fault-tolerant CMP throughput than existing mechanisms at a lower cost. MRE resulted in a mean performance degradation of 11–14% compared to non-redundant execution.
Chapter 5

Conclusions and Future Work

5.1 Conclusions

Relentless scaling of CMOS fabrication technology in making future processors increasingly susceptible to transient faults, wearout related permanent faults and process variations. As a result, fault tolerance, which was previously restricted to the domain of mainframe computers and specially designed fault-tolerant systems may now become important for the commodity market as well. The commodity market has different requirements than traditional fault-tolerant solutions; it needs configurable and low-cost fault-tolerant solutions.

This thesis addressed two problems relating to low-cost fault tolerance. The first of these is energy-efficient fault tolerance, while the second is mitigation of the throughput loss due to fault tolerance.

Our work is performed in the context of leader/follower architectures, where a single program is executed on two cores of CMP. One of these cores is designated as the leading core and the other core is designated the trailing core. Typically, the leading core assists the execution of the trailing core by forwarding the results of its execution. In this thesis we introduce a new form of execution assistance called critical value forwarding. Critical value forwarding identifies instructions on the critical path of execution and forwards the results of these to the trailing core. By focusing on critical instructions, critical value forwarding achieves most of the speedup of forwarding all values at a fraction of the
bandwidth cost.

We proposed an architecture that uses execution assistance mechanisms like critical value forwarding to design energy-efficient fault-tolerant CMPs. This architecture exploits the speedup of the trailing core due to execution assistance by operating the trailing core at a lower voltage/frequency level. In this context, we introduced two new per-core DVFS algorithms for dynamically adjusting the frequency of the trailing core. We also proposed new mechanisms for input replication in fault-tolerant CMPs.

Our results showed that this proposal for energy-efficient fault-tolerant CMPs had a performance degradation of less than 1% and energy consumption that is only 1.26 times that of non-redundant execution; this is achieved at a modest interconnect bandwidth cost. Our evaluation also showed that this proposal compares favourably to existing proposals for energy-efficient redundancy.

We also proposed an architecture that increases the throughput of fault-tolerant CMPs by executing multiple trailing threads on a single processor core. This technique is called multiplexed redundant execution, and uses coarse-grained multithreading to execute multiple trailing threads on a single core. Our evaluation showed that this proposal delivered 9–13% higher throughput than previous proposals, including one configuration that uses simultaneous multithreading (SMT) to multiplex trailing threads. This increase in throughput comes at a modest cost in single-thread performance, mean slowdown varying between 11%–14%.

5.2 Future Work

Several interesting directions for future work are possible based on this thesis. A few suggestions are listed in the following subsections.

5.2.1 Energy-efficient Timing Speculation

Our results in §3.3.5 indicate that critical value forwarding provides a mean speedup of about 1.6X of the trailing core at modest bandwidth cost. This speedup can be exploited
for energy-efficient timing speculation in a manner similar to Paceline [32].

The leading core can be operated at higher than the safe frequency, while the trailing core can be operated at a lower voltage-frequency level than the nominal frequency. The leading core improves performance, while the trailing core detects and recovers from occasional timing errors.

Paceline uses branch forwarding to speedup the trailing core. In comparison, critical value forwarding provides higher speedup of the trailing core at approximately the same bandwidth cost. Higher speedup enables energy reduction because the trailing core can be operated at lower than the baseline voltage-frequency levels, unlike in Paceline.

5.2.2 Improved Mechanisms for Identifying Critical Instructions

Our current mechanism for identifying critical instructions are optimised for simplicity of implementation. Higher speedup of the trailing core may be possible if more complex mechanisms are adopted.

For example, the fanout2 heuristic simply counts the number of in-flight consumers of an instruction’s value without accounting for wrong-path instructions. Ideally, wrong-path instructions should not contribute to the fanout computation. A more accurate counting mechanism that delays forwarding values until mispredictions are known might reduce bandwidth pressure without reducing speedup.

Furthermore, the mechanisms we propose for critical value identification are bandwidth and speedup oblivious. In some benchmarks like mcf, the indirect execution assistance provided by a shared L2 cache itself significantly speeds up the trailing core. In such cases, bandwidth requirements can be reduced by not forwarding the results of some instructions if it can be detected that sufficient speedup of the trailing core is already being achieved. A related case is that of apsi. Apsi is significantly slowed down by critical value forwarding due to interconnect bandwidth limitations. The performance of apsi may be improved by not forwarding the results of some instructions in order to reduce interconnect bandwidth pressure.
5.2.3 Adaptive Multiplexing Schemes

The results presented in this thesis provide an initial evaluation of the benefits of multiplexed execution using a static multiplexing scheme. A number of optimizations to MRE are possible. Two examples are: (1) adaptively migrating trailing threads across cores to reduce the performance losses due to multiplexing, (2) dynamically turning off multiplexing for program phases where the trailing thread does not benefit from execution assistance, and (3) redundant execution in the shadow of an L2 cache miss like in MBI [73].

Adaptive migration might be attractive for future multicore processors where a large number of programs are being redundantly executed on a large number of cores. The idea here is to adaptively and dynamically migrate trailing threads of programs to cores which have execution slack to accommodate them.
Bibliography


[39] Intel Corp. Intel Turbo Boost technology in Intel Core microarchitecture (Nehalem)
based processors, November 2008.


