The Pennsylvania State University
The Graduate School
College of Engineering

REDUNDANCY AND PARALLELISM TRADEOFFS FOR RELIABLE, HIGH-PERFORMANCE ARCHITECTURES

A Thesis in
Computer Science and Engineering
by
Angshuman Parashar

© 2007 Angshuman Parashar

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2007

The thesis of Angshuman Parashar was reviewed and approved* by the following:

Anand Sivasubramaniam
Professor of Computer Science and Engineering
Thesis Advisor, Chair of Committee

Mary Jane Irwin
Professor of Computer Science and Engineering
A. Robert Noll Chair of Engineering

Chita R. Das
Professor of Computer Science and Engineering

Vijaykrishnan Narayanan
Associate Professor of Computer Science and Engineering

Kenneth Jenkins
Professor of Electrical Engineering

Sudhanva Gurumurthi
Assistant Professor of Computer Science, University of Virginia
Special Member

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

For decades, high performance processors have provided architectural and microarchitectural abstractions that enable applications to exploit parallelism as a means to fruitfully utilize the exponentially increasing on-chip transistor counts. With shrinking device sizes, reduced supply voltages and growing integration densities leading to logic elements becoming increasingly susceptible to transient faults, research works have established techniques to leverage existing on-chip parallelism-targeted abstractions to provide redundancy for processor pipelines, thereby increasing their resilience to transient faults.

While prior works have viewed the impact of provisioning such redundancy as performance and/or implementation costs, this thesis attempts to:

- establish the view that there exists a fundamental *tradeoff* between parallelism and redundancy,
- propose efficient mechanisms that can be built to enable processor cores to operate at multiple points in this tradeoff space and argue the utility of these mechanisms, and
- demonstrate how locality-based dataflow characteristics can be exploited in novel ways to shift this tradeoff space to more efficient regions.

Table of Contents

List of Figures **vii**

List of Tables **x**

Chapter 1
Introduction **1**

Chapter 2
Background and Related Work **4**
2.1 The Soft Error Problem and Processor Core Reliability 4
2.2 Redundant Multithreading 5
2.2.1 The Sphere of Replication 6
2.2.2 Simultaneous Redundant Threading 6
2.3 Quantifying Vulnerability 7

Chapter 3
Exploiting Dynamic Instruction Reuse to provide Redundancy **9**
3.1 Introduction 9
3.2 Motivating the Need for Boosting ALU Bandwidth on a DIE 12
3.2.1 Provisioning Dual-Instruction Execution (DIE) in a Superscalar Processor 12
3.2.2 Impact of Hardware Resources 14
3.3 Microarchitectural Design of IRB-Based DIE 16
3.3.1 Dynamic Instruction Reuse 17
3.3.2 DIE-IRB Pipeline 18
3.3.3 Complexity-Effective DIE-IRB Design 21
3.3.4 Redundancy Characteristics of DIE-IRB 24

3.3.5 Minimizing Conflict Misses in the IRB . . . 26
3.4 Experimental Results . . . 26
3.4.1 Simulation Platform and Workloads . . . 26
3.4.2 Benefits of DIE-IRB . . . 28
3.4.3 Enhancing IRB Reuse Characteristics . . . 32
3.4.3.1 IRB Configurations . . . 33
3.4.3.2 Saturation Counter Management . . . 34
3.4.3.3 Proactively Inserting Entries into the IRB . . . 36
3.5 Concluding Remarks . . . 39

Chapter 4
Locality-Triggered Slice-Based Redundant Threading 40
4.1 Introduction . . . 40
4.2 Related Work . . . 42
4.3 SlicK . . . 43
4.3.1 SlicK Overview . . . 43
4.3.2 Identifying Trigger Instructions . . . 43
4.3.2.1 Store Predictor . . . 44
4.3.2.2 Branch Confidence Estimator . . . 45
4.3.3 Extracting Backward Slices . . . 45
4.3.3.1 Slice Extractor Design Goals . . . 45
4.3.3.2 Inadequacy of Traditional Mechanisms . . . 46
4.3.3.3 Our Solution – the SliceEM . . . 47
4.3.3.4 SliceEM Operations . . . 48
4.3.3.5 Hardware Implementation Concerns . . . 52
4.3.4 Integrating the Predictors and SliceEM into an SRT processor 52
4.3.4.1 Memory Ordering for the Trailing Thread . . . 54
4.4 Fault Coverage . . . 54
4.4.1 Identifying Points of Vulnerability in SlicK . . . 54
4.4.2 The Common Case – How SlicK detects soft errors . . . 56
4.4.3 Quantifying the Fault Coverage of SlicK . . . 57
4.5 Results . . . 57
4.5.1 Performance Results . . . 60
4.5.2 Fault Coverage Results . . . 63
4.5.3 Discussion . . . 64
4.6 Concluding Remarks . . . 66

Chapter 5
Bounding Vulnerabilities of Processor Structures 68
5.1 Introduction . . . 68

5.2 Background and Motivation 70
5.3 Dynamic Vulnerability Monitoring 72
5.4 Vulnerability Control via Selective Redundancy (VCSR) 74
5.4.1 Achieving Selective Redundancy 75
5.4.1.1 Maintaining a Consistent Architected State 76
5.4.2 Selecting Instructions for Redundant Execution 77
5.5 Results 79
5.6 Concluding Remarks 82

Chapter 6
Future Work **84**
6.1 Vulnerability Control 84
6.1.1 VC on Multiple Structures 84
6.1.2 VCSR Implementation Issues 85
6.2 Chip-Level Redundancy 86
6.2.1 The View of Memory 87
6.2.2 IR/OC at the Processor-L1 Cache interface 88
6.2.3 IR/OC at an On-Chip location in the Memory Hierarchy . . 89
6.2.3.1 The Redundancy-Merging Buffer 90
6.2.4 Off-chip IR/OC 95
6.2.5 Discussion 95

Bibliography **97**

List of Figures

2.1 Sphere of Replication 6

3.1 Implementing DIE in a Superscalar Processor, as proposed in [39]. The gray shaded area is the Sphere of Replication. The solid lines show the flow of the primary instruction stream and that for the secondary/dual stream is given in dashed lines. The access to the data-cache initiated by either primary or dual, though not both, is shown in dotted lines. 13
3.2 Percentage IPC Loss with respect to SIE 15
3.3 Instruction execution pipeline under different instruction-reuse scenarios. The solid lines show the flow of execution of instructions in the primary stream and the dashed lines for those in the duplicate stream. 19
3.4 Proposed Datapath Enhancements for DIE-IRB. Note that IRB lookup is done concurrently with Fetch/Decode/Dispatch and there is no dotted line from the result forwarding path to the issue window. 20
3.5 Design of issue window wakeup logic to support input-operand forwarding for DIE-IRB. Note that *IRB L/R* come into issue logic together with the instruction. 22
3.6 Value forwarding within/across instruction streams. 27
3.7 Handling faults on the result-forwarding bus using duplicate forwarding paths. 28
3.8 Percentage of IPC Gap Recovered from DIE using Instruction-Reuse. The y-axis plots the value of $\frac{IPC_{scheme} - IPC_{DIE}}{IPC_{SIE} - IPC_{DIE}}$. 30
3.9 Impact of IRB Configurations 33
3.10 Impact of various Saturation Counter Management Schemes 35
3.11 IRB Stride Predictor 37

4.1 The Slice Extraction Matrix. Column indices are indicated at the bottom of the matrix. 49

4.2 Process of inserting the instruction `add $r5 := $r2 + $r3` into the SliceEM. Notice that the instruction in column 12 ceases to be Live after this insertion. The new instruction is inserted in column 13, and a new 1 is entered into the matrix entry (`r5`, 13) to indicate this. 50
4.3 Process of inserting the ET store effective address calculation instruction `computeEA $r2 + 5` into the SliceEM. Notice the updated Execute Mask. 51
4.4 Overview of SlicK Pipeline 53
4.5 SlicK Vulnerability Example, showing the execution of a Store instruction through 101 iterations of a loop in which the data value is 0x4000 for the first 100 iterations and 0x4100 in the final iteration. This is shown for (a) the Correct Execution, (b) one possible Erroneous Execution (caught by SlicK), and (c) another possible Erroneous Execution (not caught by SlicK). 55
4.6 Percentage of instructions whose status (whether to execute or not) becomes known after x instructions in the dynamic instruction stream. 59
4.7 IPC results for SlicK executions with a simple Last Value (LV) and a more extensive Finite-Context Method (FCM) store address/value predictor, shown in comparison with SRT. All values are normalized with respect to Single Thread (non-redundant) execution. 61
4.8 Normalized IPC and maximum RUU-AVF that were run with a subset of trigger instructions randomly chosen to be dropped together with a redundant execution that drops only dynamically dead instructions. Note that the AVFs of SRT and Dead experiments are 0. 65

5.1 Dynamic AVF Variation in *177.mesa* at Cycle (top) and 100-Cycle (bottom) Granularity 71
5.2 VCSR Pipeline Overview. 76
5.3 IPC with VCSR for various AVF bounds. Also provided for each benchmark are IPCs for single-thread execution (rightmost bar) and SRT (leftmost bar). 80
5.4 Percentage of instructions redundantly executed with VCSR for different AVF bounds. 81
5.5 Summary of Results. Geometric Mean of IPC across Applications Normalized w.r.t. Single Thread IPC for different AVF bounds. Also shown is SRT performance on the y-axis (i.e. AVF=0). 82

6.1 Effect of controlling the AVF of the ROB on the AVF of the Issue Queue. 85

6.2 The Redundancy-Merging Buffer. 90
6.3 Total data transferred on-chip for Output Comparison, shown for SCB and RMB implementations. 92
6.4 Total data transferred off-chip for SCB and RMB implementations. 93
6.5 1000's of cycles executed before RMB is deadlocked, for multiple RMB sizes. 94

List of Tables

3.1 Default simulation parameters. Latencies of ALUs/caches are given in parentheses. All ALU operations are pipelined except division and square-root. 29

3.2 IPC under SIE and DIE, and their gap, for simulated SPEC2000 benchmarks. 30

3.3 Classification of IRB-Accesses 32

4.1 Simulation parameters. Latencies of ALUs/caches are given in parentheses. All ALU operations are pipelined except division and square-root. 58

4.2 IPC of single-thread (non-redundant) execution of the benchmarks and redundant execution using SRT. The loss of IPC with SRT is also given, with the average loss across the applications being 17.24%. All the data is for the execution of 100 million instructions from the first SimPoint with a weightage at least 1% for each benchmark. 59

4.3 Execution Statistics for SlicK with LV predictor. We show the percentage of flushed/eliminated instructions, the number (and as a percentage of total branches) of Branch FTs, the number (and as a percentage of total stores) of Store FTs. 60

4.4 Architectural Vulnerability Factors for important structures in a SlicK processor with LV store predictors. Vulnerability comes from unguarded instructions due to predictor Mismatches (0.58% on the average), while performance comes from flushed instructions due to predictor Hits (46% on the average). 63

5.1 Simulation parameters. Latencies of ALUs/caches are given in parenthesis. All ALU operations are pipelined except division and square-root. 79

Chapter 1

Introduction

Increasing transistor integration densities coupled with high clock frequencies and lower operating and threshold voltages make reliability an important consideration for future processor designs. Deep submicron process technologies are making circuits increasingly susceptible to several kinds of permanent and transient faults. Soft errors [6] in the form of Single Event Upsets (SEUs) caused by cosmic ray strikes [61] are one such class of errors which future processor designs will be required to address. Traditionally, only high-end mission-critical systems featured extensive fault tolerance mechanisms to handle such transient faults. In the near future, transient faults are expected to consume a significant fraction of the reliability budgets of systems targeted at a wide range of market segments.

Data stored in memory arrays are relatively straightforward to protect using informational redundancy techniques such as parity and ECC. However, technology trends indicate that random combinational and sequential logic circuits within processor pipelines will need to be protected as well [45] in order for future systems to stay within their reliability budgets [31]. Therefore, several research and development efforts in the past decade have embarked on enhancing processor core reliability against transient errors [41, 27, 40, 56, 39, 16]. Many of these approaches achieve redundancy in the core(s) by executing multiple instances of an application, with a subsequent comparison of their outputs to verify their integrity.

These techniques often make use of abstractions already available in modern

microarchitectures in order to implement the redundancy. For example, *DIE* [39] duplicates every instruction after the decode stage of a superscalar processor and exploits the processor's renaming mechanisms to maintain independence between the redundant executions. *SRT* [40] uses the two contexts of a Simultaneous Multi-threading (SMT) processor to implement redundancy, while *CRT* [30] achieves the same using multiple cores on a Chip Multiprocessor (CMP). In order to achieve full redundancy, these approaches result in throughput being effectively halved on CMP-based implementations (spatial redundancy), and performance being degraded by 20-30% on the average on superscalar and SMT-based implementations (temporal redundancy). This performance impact occurs because all of these approaches leverage upon microarchitectural constructs that were primarily designed to exploit *parallelism*, be it at the instruction level or the thread level or program level. This performance impact can be viewed as a fundamental tradeoff between parallelism and redundancy, which translates to a corresponding tradeoff between performance and reliability.

The goal of this thesis is to (a) gain insights into this relationship, (b) provide efficient mechanisms to explore this tradeoff space, and (c) explore alternative forms of redundancy that can be used to shift the entire tradeoff space to more favorable regions.

The first contribution of this thesis is a work that identifies the functional units in an out-of-order core to be the primary performance bottleneck in a redundant execution system, and exploits an alternative form of redundancy to reduce the impact of the bottleneck. The alternative form of redundancy is called Dynamic Instruction Reuse, which is based on the observation that a significant fraction of dynamic instructions are known to re-execute their previous computations as they flow through loop iterations. We show that this "history" can be used to provide the requisite redundancy for a large number of instructions. In this work, we strive to maintain full redundancy for all instructions, using instruction reuse to improve performance as much as possible.

At the same time, full redundancy is rarely a necessity for real systems. Systems are designed to meet the reliability requirements of their targeted market segments. These requirements are quantified in terms of a Reliability Budget, typically specified in the form of statistics such as Mean Time To Failure (MTTF)

or Mean Time Between Failures (MTBF) [31]. It is acceptable for a transient fault tolerance technique to *not* provide full coverage as long as the constraints for transient faults within the reliability budget are met. This relaxation could potentially pave the way for improved performance compared to fully redundant solutions. This inspires the second contribution of this thesis, in which we propose a mechanism that executes the redundant thread at the granularity of *slices* of instructions leading up to Stores, and uses value and control-flow locality properties to select slices for execution. In a philosophy similar to that used in the Instruction Reuse work, we use this Locality itself to provide redundancy for a large number of slices, resulting in a solution that provides improved performance with near-perfect coverage.

While the above approaches achieve perfect or near-perfect coverage with significantly improved performance compared to baseline redundant threading techniques, there is still a significant performance drop compared to single-threaded execution. Using various mechanisms, it is possible to trade off redundancy in order to gain back as much of this lost performance as required, but in doing so, the reliability budget must be brought into the equation as a first-order design constraint. The third contribution of this thesis proposes a set of techniques using which a processor can dynamically optimize performance while meeting any specified vulnerability bound by actively monitoring its vulnerability and adjusting the level of redundancy in order to remain within the specified budget.

Chapter 2

Background and Related Work

2.1 The Soft Error Problem and Processor Core Reliability

Higher transistor integration densities, increasing clock frequencies and lower operating voltages are causing circuits to become increasingly vulnerable to soft errors [6] such as those caused by cosmic ray strikes [61]. Well-established techniques such as parity and hamming codes can be used to protect data stored in memory cells, but technology trends mandate verifying the integrity of latches and combinational circuits within the processor pipeline as well [45]. Consequently, several recent research efforts have proposed techniques to efficiently protect the processor core from transient errors [41, 27, 40, 56, 39, 16].

Error Model: Transient errors can have one of three possible outcomes: (a) they can be benign errors that do not have any effect on execution, (b) they can lead to Detected Unrecoverable Errors (DUE), or (c) they can lead to Silent Data Corruption (SDC)[31]. All contributions of this thesis, as well as most of the discussed related work, focus on techniques to avoid SDC errors in a Single Event Upset (SEU) fault model. The probability of independent double-bit errors is assumed to be negligible. Although several of the proposed techniques are also capable of detecting most occurrences of independent double-bit errors, we do not provide any quantification for this capability and therefore restrict our discussions

to single errors. Further, although one of the primary motivations behind designing transient error detection mechanisms is bit-flips arising from atmospheric particle strikes, all proposed mechanisms are capable of detecting any transient errors that satisfy the SEU model, regardless of the physical phenomena that caused the errors.

Early work in developing highly resilient transient fault-tolerant systems used hardware duplication to implement lockstepping, as in the HP NonStop Himalaya [20] and IBM G5 [46]. The DIVA architecture [2, 8] takes a slightly different approach wherein a special checker processor is used at the commit stage of the main processor core's pipeline to verify the correctness of the instructions being committed. Our work is more related to the emerging area of cost-effective transient fault detection/recovery techniques using redundant execution [30, 40, 56]. These recent proposals try to leverage existing microarchitectural resources, thus making them more cost-effective for a diverse set of market segments. Further, it is not necessary to provide complete fault coverage for all deployments, and the trade-offs between performance, implementation cost/complexity and fault coverage offer a rich space of operating points for different market segments.

2.2 Redundant Multithreading

Many of the above-mentioned recent proposals are based on the idea of Redundant Multithreading (RMT), where the instruction stream is redundantly executed on multiple execution contexts, with a subsequent comparison between the outputs of the two streams (the stores to the memory system in particular) to verify their integrity [40]. Implementations of the concept have been proposed for Simultaneous Multithreading (SMT) processors as well as Chip Multiprocessors (CMP). Contention for core resources (bandwidth and storage) between the multiple redundant threads leads to significant performance degradation relative to non-redundant execution for most of these implementations. In particular, previous studies have reported 20-30% performance loss for SMT implementations. There have been studies on efficiently sharing the instruction queue and reorder buffer capacities [48] and appropriately staggering the redundant threads [37] to minimize performance loss due to redundant execution.

2.2.1 The Sphere of Replication

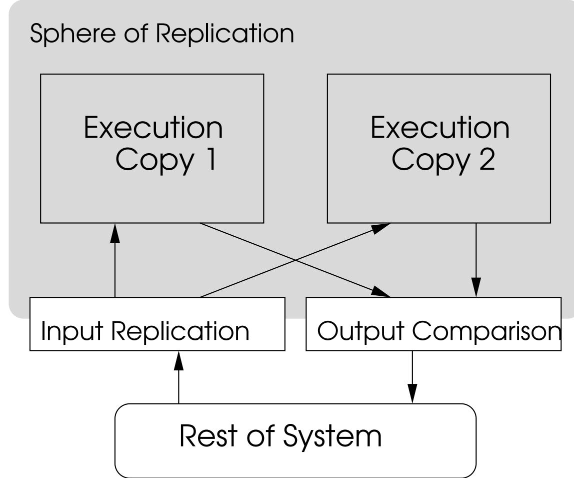


Figure 2.1. Sphere of Replication

Central to any form of Redundant Multithreading is the concept of the Sphere of Replication (SoR). All components within the SoR are protected via redundant execution, and those outside it must be protected by other means such as spatial replication or ECC. Implementing the SoR involves two operations: input replication and output comparison. Redundancy is created at the input replication point and integrity verification is performed at the output comparison point before the effects of the computation are allowed to propagate outside the SoR.

2.2.2 Simultaneous Redundant Threading

Simultaneous Redundant Threading (SRT) [40] leverages the multiple contexts provided by a Simultaneous Multithreading (SMT) processor [55] to execute redundant copies of the same program. The L1 cache interfaces are used as the input replication and output comparison points. Thus, all of the core structures including the PC, fetch and decode logic, register files, the ROB, the issue logic and the functional units lie within the SoR. The two execution streams running on the SMT processor's contexts are referred to as the *leading* (primary) and *trailing* (redundant) threads. The fetch mechanism attempts to maintain a slack between the threads. To improve performance, SRT makes use of a Branch Outcome Queue (BOQ) which contains the targets for the resolved branches in the leading thread

to prevent branch mispredictions in the trailing thread. The leading thread places load values obtained from the data cache into the Load Value Queue (LVQ) for subsequent lookup by the trailing thread. The trailing thread issues loads in program order and absorbs values from the LVQ. At the output comparison point, the address and data of every store instruction is verified by comparing the respective outputs from the two redundant executions. To achieve this, retiring stores are sent out of the store queue into a Store Checking Buffer (SCB) where they wait until the trailing thread catches up. If no discrepancy is detected, a single (architected) copy of the store is retired into the memory system in program order. The SCB should also be capable of forwarding values to the loads in the leading thread. If any one of the instructions encounters an error during the course of its residence within the SoR, the error would be detected at the output comparison point, namely, at the first store instruction that is on the forward execution slice of this instruction.

2.3 Quantifying Vulnerability

The vulnerability of a system component (such as a memory array or a processor structure) is estimated by first performing circuit-level analyses to arrive at a raw error rate, which can be expressed in terms of Mean Time Between Failures (MTBF) or Failures in Time (FIT) [31]. The raw FIT rate (FIT_{raw}) is then *derated* because microarchitectural and architectural effects reduce the probability that a transient error in the structure will actually lead to an observable error in the output. This probability can be encapsulated in terms of *Architectural Vulnerability Factors* (AVF) [31, 25]. The effective FIT rate of a structure is given by[1]:

$$FIT_{eff} = AVF \times FIT_{raw} \tag{2.1}$$

The effective FIT rate of the entire system is obtained by summing up the effective FIT rates of all its constituent components.

[1]This thesis is primarily concerned with architectural and microarchitectural issues and therefore, for clarity, we assume that propagation and timing effects (PVF, TVF) are integrated into the raw FIT rate estimates.

Computing Architectural Vulnerability Factors: The AVF of a structure captures the probability that a transient error in the structure will manifest itself in observable output. At a certain point in time, any bit in a structure can be classified as either ACE (required for architecturally correct execution) or un-ACE. Only errors in ACE bits will result in observable errors. Un-ACE-ness arises from several sources: un-occupied structural entries, mis-speculated instructions, dynamically dead instructions, logical masking, etc. (for details, please refer to [31]). The AVF of the structure is defined as the average-over-time of the ratio of ACE bits in the structure to the total number of bits in the structure.

The AVF metric is useful for architects because it de-couples process-technology and circuit-level effects from architectural and microarchitectural effects on soft error rates, thereby enabling quantitative analysis of architectural transient fault-tolerance solutions independent of underlying variables.

Chapter 3

Exploiting Dynamic Instruction Reuse to provide Redundancy

3.1 Introduction

Reliability enhancements are typically provisioned using spatial and/or temporal redundancy mechanisms, which require additional hardware, additional complexity in design, and/or incur performance penalties. Care must be taken while provisioning such redundancy so as not to defeat the purpose of the fundamental drive towards technological innovations for high performance. With this underlying philosophy in mind, we present a technique for narrowing the performance gap between an instruction-level temporally redundant out-of-order execution and an execution on a normal out-of-order core without any temporal redundancy.

Temporal redundancy can be implemented at different granularities: instruction-level [27, 39] or thread-level [41, 40, 56, 16]. In our terminology, we refer to proposals that exploit hardware supported multithreaded architectures for temporal redundancy as being thread-level, and proposals such as [27, 39] which implement mechanisms in an ordinary out-of-order core as instruction-level (with temporal separation between the two executions being more fine-grained). In the past few years, thread-level temporal redundancy has been extensively investigated with several promising proposals for enhancing its performance. The problem has been more difficult when implementing instruction-level temporal redundancy with-

out significant performance consequences. Since the resources in an out-of-order (OOO) core have been tuned with one instruction stream in mind (referred to as Single Instruction Execution, or SIE, henceforth), resource contention becomes a serious impediment to performance when these instructions are replicated temporally (we focus on Dual Instruction Execution mechanisms in this chapter, which we refer to as DIE). Previous studies [39] have reported up to 45% performance loss for SPEC CPU2000 applications for DIE compared to SIE. Any effort to alleviate this loss, without significantly complicating hardware design, can have considerable impact on future processor designs. It can be directly employed in a superscalar design to enhance reliability without significantly degrading performance. It can also be used to enhance the OOO core within a SMT or CMP architecture, to either supplement or even replace the coarser-grain thread level redundancy mechanisms. The evaluations in this section mainly focus on a superscalar design.

Contention for different resources in the OOO core, including the Reorder Buffer (ROB), the ALUs and the issue bandwidth in the DIE execution can cause longer delays thereby slowing down the execution compared to SIE While it may be possible to increase some of these resources to ease such contention in future billion-transistor designs, at the core of the problem lies the functional units/ALUs that need to be shared between the instruction streams, and the issue unit that needs to schedule those ALUs to the waiting instructions. The issue, wakeup and bypass logic is a complex piece of logic to design, often lies on the critical timing path of the processor, and is not very scalable [33, 21].

Even though it may be possible to add more ALUs, the consequence is that: (i) the issue unit needs to schedule more ALUs, and (ii) the wakeup logic needs to wake up dependent instructions, and the outputs from the ALUs have to be bypassed to the waiting instructions. The issue logic design complexity obviously grows with the number of ALUs that we provision, making it a less desirable option for reducing the gap between DIE and SIE.

Without going to the extreme of replicating many of the hardware resources, and at the same time attempting to simplify design complexity, this thesis presents a novel adaptation of a hardware technique (that has been previously examined in performance optimizations of single streams), called *instruction reuse*, to bring a DIE on an OOO core closer to the performance of SIE. The concept of instruction

reuse for optimizing SIE was first proposed in [50], and is based on the following observation: when encountering an instruction which was executed at some time in the past, we could directly use the output of the previous execution (instead of re-executing it) as long as the operands match. A simple cache (referred to as Instruction Reuse Buffer or IRB) of instructions and their operands and result-values, can be used to exploit such reuse.

An IRB can not only speed up the execution of a multi-cycle instruction, but can also enhance the overall execution bandwidth by allowing multi-ported lookups (resembling multiple ALUs). Previous studies [50, 10, 11] of IRB for single streams have pointed out that it is more useful for speeding up long latency operations, rather than the execution bandwidth, since the OOO cores have already been designed in a balanced manner for bandwidth. As a result, the work on IRBs gradually evolved into specialized mechanisms (such as value prediction) targeting long latency operations. While an IRB may seem like an effective way of amplifying ALU bandwidth without impacting the scheduling costs of the issue logic, it still does not solve the problem of having to propagate the results (from the lookup) back to waiting instructions.

Using the IRB as a starting point, this thesis makes the following main contributions:

- We illustrate the use of an IRB to ease the ALU resource contention problem of a DIE system. The IRB is used to not only speedup the long latency instructions, but to amplify the execution bandwidth, which is extremely useful to handle the doubling of load imposed in a DIE system. Whenever possible, we direct the duplicate stream through the IRB to ease the ALU requirements. At the same time, we point out that the IRB does not need to be protected from faults with any additional hardware mechanisms.

- We show how this amplification of execution bandwidth can be achieved without requiring an increase in issue width (and the scheduling costs). Consequently, it is a simple hardware extension over the resources that one would provision anyway for a balanced SIE system.

- We identify an interesting property of a DIE system, wherein we can use the output from instructions of one stream as input operands for another,

without really compromising on the desired level of temporal redundancy. This property helps us devise an IRB that does not require its outputs to feed back into the issue window, consequently not affecting the issue logic design complexity. We give details on the implementation of such an IRB, together with its integration with the rest of the datapath.

- Using several applications from the SPEC CPU2000 suite, and a detailed cycle-level model, we demonstrate how our enhancement can narrow the performance gap between DIE and SIE.

3.2 Motivating the Need for Boosting ALU Bandwidth on a DIE

3.2.1 Provisioning Dual-Instruction Execution (DIE) in a Superscalar Processor

In the context of superscalar processors, previous proposals have implemented the Sphere of Replication by instruction duplication [27, 39]. In [27], it is assumed that the program counter (PC), decode logic, register file, rename tables, and ROB are outside the SoR, and hence trustworthy. In [39], the authors bring the ROB into the SoR as well. They, however, leave the PC and branch-prediction structures outside the SoR since control-flow errors can be detected at the time the respective branch-instructions are resolved (i.e. though outside, these structures can still be considered safe for program execution). This modified superscalar processor, which we refer to as DIE in the rest of this thesis, is shown in Figure 3.1. At decode/dispatch, each instruction is duplicated, whereby two adjacent entries in the ROB are created. The primary instructions and their duplicates are then dispatched to the issue window and both of them complete independently based on the dataflow order of their respective streams (there is no communication between the primary and secondary streams). At the commit point, the corresponding primary-dual pairs of instructions are checked against each other to detect any inconsistencies. If no inconsistency is detected, the architected instruction is retired; otherwise an instruction-rewind is triggered (using the existing mechanism that is

Figure 3.1. Implementing DIE in a Superscalar Processor, as proposed in [39]. The gray shaded area is the Sphere of Replication. The solid lines show the flow of the primary instruction stream and that for the secondary/dual stream is given in dashed lines. The access to the data-cache initiated by either primary or dual, though not both, is shown in dotted lines.

used to recover from incorrect speculation) from the inconsistent instruction.

In the proposed DIE [39], instruction-level temporal redundancy has been provisioned without significantly additional hardware resources. The two streams (primary and dual) share the ROB capacity, the ALUs, etc. Consequently, an IPC loss (shown to be around 30% on the average for some SPEC2000 and SPEC95 applications) is to be expected compared to SIE. However, the dual execution is exploited for reducing the overhead of branch mispredictions and memory accesses. For instance, as soon as one stream detects a branch misprediction (instead of waiting for each to find out), the instructions for both streams along the mispredicted path are squashed and execution is initiated along the correct path. Similarly, since the memory system is assumed to be outside the SoR, for loads/stores, only the memory address calculation is performed for both, and the actual access is performed only once. This optimization also allows for preserving the semantics of uncached accesses and strong memory-ordering, that are typically required for memory-mapped devices.

3.2.2 Impact of Hardware Resources

To motivate the rest of this chapter, we first conduct a set of experiments to understand the impact of the extra load imposed by a DIE system, compared to SIE. There are several hardware resources in the SoR (e.g. ALUs, Issue Window, ROB) shared by the two streams, which need to handle this extra load. In general, we observed that by amplifying the number of ALUs (that actually execute the instructions), the issue width (which determines the maximum number of instructions that can be sent for execution to the ALUs every cycle), and the RUU size (which determines the number of instructions in flight in the processor), we can gain back most of the IPC loss for DIE compared to SIE. Consequently, we focus on these three parameters.

The base configuration of the SIE and DIE under consideration are presented later in section 5.5, and the important parameters to note are the number of ALUs (4 integer adders, 2 integer multipliers/dividers, 2 floating point adders, 1 floating point multiplier/divider/square-rooter), the RUU/Load-Store queue size (128/64-entries) and the issue width (8). We consider configurations that double the quantity of each of these hardware units, namely 8/4/4/2 ALUs (called *DIE-2xALU*), 256 RUU-entries and 128 Load-Store Queue entries (called *DIE-2xRUU*), and a decode/issue/commit width of 16 (called *DIE-2xWidths*). This gives us the following 7 configurations: *DIE-2xALU*, *DIE-2xRUU*, *DIE-2xWidths*, *DIE-2xALU-2xRUU*, *DIE-2xALU-2xWidths*, *DIE-2xRUU-2xWidths*, and *DIE-2xALU-2xRUU-2xWidths*, which together with the base DIE are compared with the IPC of the base SIE (i.e. the base DIE and SIE have the same capacity of resources for these units). In Figure 3.2, we plot the IPC slowdown for these configurations with respect to the base SIE for 12 applications from SPEC2000.

Performance degradation of DIE is anywhere from 1% (in *ammp*) to nearly 43% (in *art*) with around 22% on the average compared to SIE. This reiterates the earlier observation that temporal redundancy imposes a high performance penalty on the system. At the same time when we double the capacity of the three aforementioned components (*DIE-2xALU-2xRUU-2xWidths*), we are able to achieve IPCs very close to that of the SIE system (note that the rightmost bar is very short and may not even visible for many benchmarks in the figure).

While doubling the capacity of each of the three units improves IPC, we find

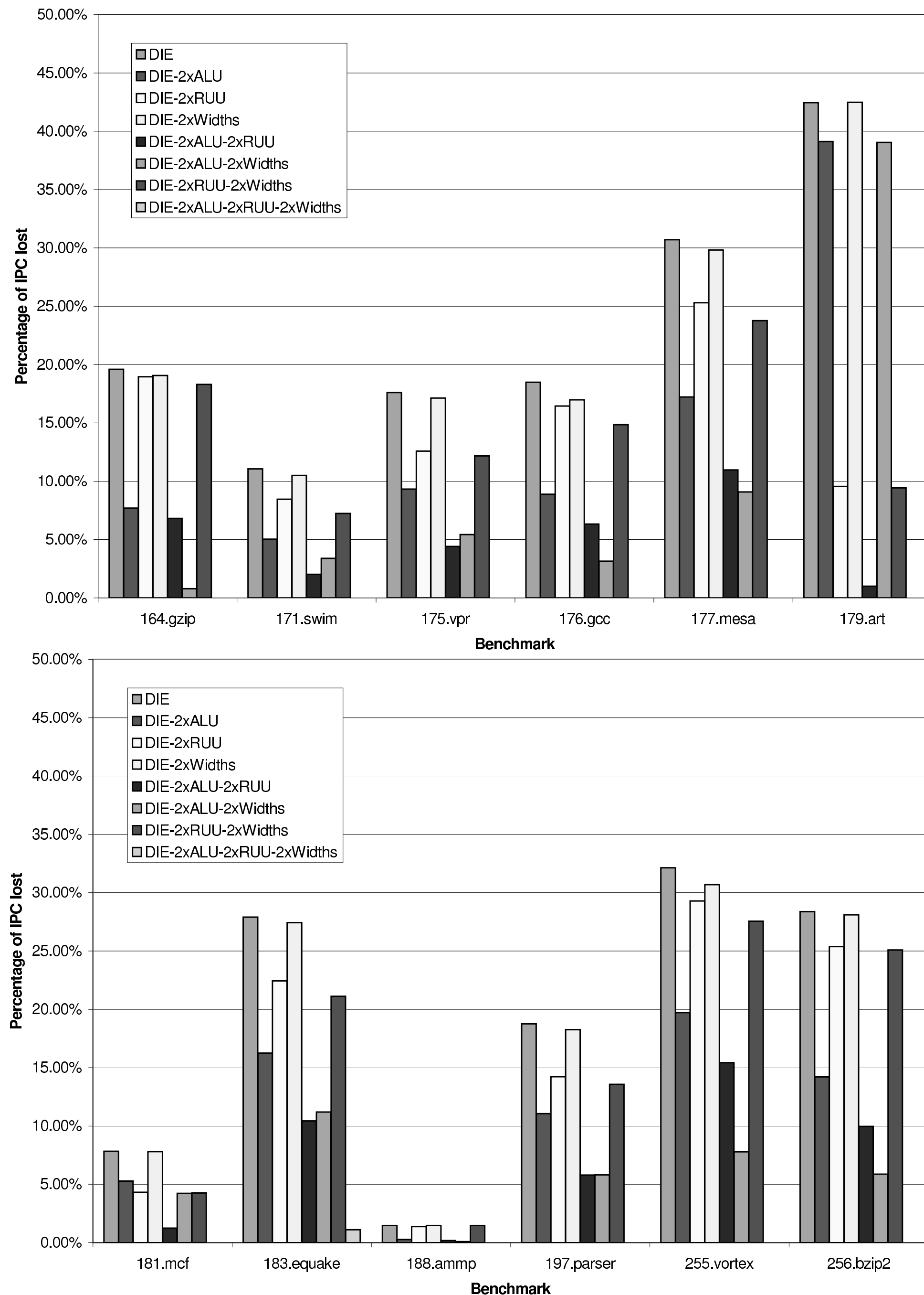


Figure 3.2. Percentage IPC Loss with respect to SIE

that, among the three, doubling the number of ALUs (the *DIE-2xALU* scheme) provides the maximum reduction in the IPC loss (13% loss in IPC on the average, compared to 16% for the RUU and 21% for the widths). An exception to this is *art* where we find that doubling the capacity of the RUU provides greater savings

possibly because this exposes more instruction and memory-level parallelism to the hardware for exploitation in this low IPC application (its IPC for SIE and DIE are 0.7316 and 0.4113 respectively). In many other cases, just doubling the ALU capacity provides better improvement than doubling all the other factors. Further, even when higher capacity in the other units can improve performance, an improvement in ALU bandwidth can further amplify their gains (note the bars for the different configurations where the ALU capacity is doubled). This motivates the need for amplifying the ALU bandwidth to enhance the performance of DIE.

3.3 Microarchitectural Design of IRB-Based DIE

In the previous section, we observed that a large fraction of the IPC loss in DIE was primarily due to the bandwidth limitation of the functional units/ALUs. A possible solution to this problem could be to increase the number of ALUs. Though intuitive (and as the previous section shows, the rewards can be substantial in an idealistic setting), the problem with this solution is the design complexity of the the wakeup, selection and bypass logic, which are in the critical path.

- *Selection Logic Complexity:* The issue logic needs to allocate a larger number of ALUs to the ready instructions, thereby increasing the scheduling complexity. If one goes for stacked arbiters for the selection logic, increasing the ALUs increases the stacking depth linearly. On the other hand, an unstacked implementation increases the size of each arbiter cell, whose consequences can be even worse than a stacked implementation [32].

- *Wakeup and Bypass Logic Complexity:* The output from each of these ALUs needs to feed back to dependent/waiting instructions in the issue window. The consequent quadratic growth in delay (due to the additional complexity in wakeup and bypass logic) tends to be on the critical timing path of the processor [33, 21, 14].

Instead, our solution for amplifying ALU bandwidth uses a novel adaptation of an approach that has previously been proposed to minimize resource conflicts on a SIE system for the functional units, namely, *Dynamic Instruction Reuse* [50].

3.3.1 Dynamic Instruction Reuse

Dynamic Instruction Reuse or, simply, Instruction Reuse (IR) is a non-speculative technique that is based on the observation that there tends to be significant reuse of the instructions in a program, and when an instruction appears again with the same input operands, it will produce the same result as before. IR attempts to exploit this property by buffering previously executed instructions in a small hardware table called an Instruction Reuse Buffer (IRB), indexed by the PC [50] (there have been variations proposed [9, 10]). Every instruction performs a lookup of the IRB to check if it has an entry, and if so, whether its input operands match those in its entry (called a reuse test). If there is a match ("IRB-hit"), then the instruction does not require a functional unit and it re-uses the value from the previous execution. If there is a miss, it has to execute on the functional units. IR was initially proposed to overcome the dataflow limit for collapsing true dependencies and to minimize resource conflicts [50, 51]. In a follow-up research, Citron et. al. [10] found that IR is effective only for long-latency operations. Since then, IR research evolved more into the study of value prediction [26, 23].

In this thesis, we re-visit IR in a new light. In a DIE-based system, we observe that the primary cause for the loss in IPC is more due to *large number of single-cycle instructions contending for a small number of functional units*, rather than the presence of long latency operations, since we have twice the load being imposed on the system.

We employ the IRB to relieve this extra load imposed by the duplicate instruction stream in the following ways to reduce the overheads that temporal redundancy imposes.

- In our DIE enhancement, referred to henceforth as DIE-IRB, the primary stream is always executed by the functional units as in SIE. Instructions in the duplicate stream, on the other hand, first look up the IRB. If there is a hit, they skip the functional unit and move onto the completion stage of their pipeline. If they miss, they contend for the functional units as before. If there is good instruction reuse, then the pressure on the functional units can be reduced.

- The port requirements of the IRB can be kept low (comparable to an IRB

designed originally for SIE) in order to achieve a fast access latency. This does not produce significant contention for the IRB ports since (i) only the duplicate stream accesses the IRB and (ii) the effective dispatch/commit width of a DIE system is half of that of a SIE system.

- As in a normal DIE, each pair (primary, duplicate) of instructions needs to be checked before retirement. The IRB does not need any special/extra protection, since temporal redundancy is provisioned for it by the primary stream executing in the functional units and vice-versa. Therefore, *the IRB lies within the Sphere of Replication* of the processor.

- Our IRB enhancements for a DIE do not incur the problems that arise when increasing the number of ALUs that were identified earlier.

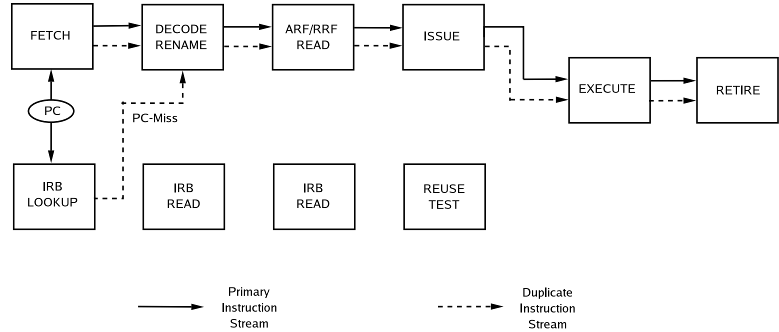
  - The number of ALUs that the selection logic needs to schedule for remains the same as in SIE.
  - In previous proposals that exploit Instruction Reuse on SIE, the IRB behaves like a functional unit and would therefore broadcast any results from a hit (from all its output ports) to its dependent instructions in the issue window. This would entail additional complexity to the issue logic, in terms of tag/result forwarding lines from the IRB to the issue window, thereby increasing the complexity of the wakeup-logic. However, using some properties unique to DIE, we show how an IRB can be incorporated in the datapath with *no extra forwarding-buses* and very little additional overhead to the issue logic.

- Finally, we explore multiple mechanisms to attempt to reduce conflict misses in the IRB and further improve performance of DIE.

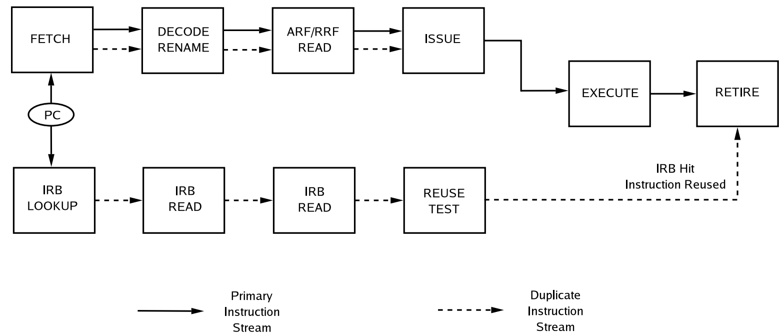
The rest of this section gives more details on these issues, together with the hardware modifications we propose to the datapath for DIE-IRB.

3.3.2 DIE-IRB Pipeline

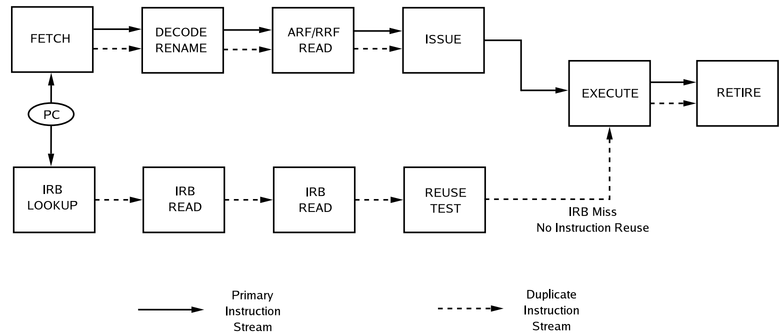
The design of the proposed hardware mandates a close scrutiny of the datapath pipeline, in order to make sure that the cost of the IRB access can be accounted



(a) PC-miss Pipeline



(b) Pipeline with successful instruction-reuse (IRB-hit)



(c) Pipeline with no instruction-reuse (IRB-miss)

Figure 3.3. Instruction execution pipeline under different instruction-reuse scenarios. The solid lines show the flow of execution of instructions in the primary stream and the dashed lines for those in the duplicate stream.

for, while still being able to ensure the benefits of an IRB. At the same time, it is important to allow more than one instruction to exploit the IRB in any one cycle, which is required for easing the ALU bandwidth requirements. In order to ensure that the IRB does not become a bottleneck in the system, we allocated the number of ports carefully. We used 4 read-ports, 2 write-ports, and 2 read-write

Figure 3.4. Proposed Datapath Enhancements for DIE-IRB. Note that IRB lookup is done concurrently with Fetch/Decode/Dispatch and there is no dotted line from the result forwarding path to the issue window.

ports. Using Cacti 3.2 [7], we found that for a 180nm process technology, a 1024 entry direct-mapped IRB (which is the eventual configuration that we suggest, and whose hit rate has been shown to be fairly good [50, 60]) can be clocked at our simulated processor clock frequency of 2 GHz using a 3-stage pipelined access as shown in Figure 3.3. The IRB is looked up in parallel with instruction-fetch using the PC as an index and there are three subsequent possibilities. In the first case, there is a PC-miss in the IRB, in which case the execution proceeds as in a normal DIE (without IRB) for the duplicate instruction. In the second and third cases, the PC hits in IRB, and it takes two more cycles to read the operands/result from the IRB entry, and one more to perform the reuse test, which, as we shall show, can be effectively overlapped with instruction-wakeup. Meanwhile, this duplicate instruction has made its way through the normal pipeline (decode/dispatch) and is at the issue stage. If the reuse test succeeds, then the duplicate instruction can bypass the execute phase, and directly move to the retire/commit stage. Otherwise, from the issue stage, the duplicate instruction goes to the functional units as in the original DIE. If the IRB needs to be updated, it is done on completion of

instruction commit/retire, and is not in the critical path.

In our implementation, we employ the IRB for integer and floating-point ALU instructions, and also for branch target-address calculation. For load/store instructions, the IRB is used only for address calculation. This simplifies our instruction reuse scheme, as we do not have to perform a potentially timing-critical memory disambiguation step to check for intervening stores for a load-instruction (which would normally require a full scan of the IRB). The modification to the DIE pipeline by the inclusion of the IRB, along with the format of an IRB-entry is shown in Figure 3.4.

3.3.3 Complexity-Effective DIE-IRB Design

In previous proposals for IR, the goal has been to overcome the dataflow limit by collapsing true dependencies. In this respect, the IRB has been treated as a functional unit. As pointed out in [10], this would be effective only if IR was targeted at long latency operations and not for single-cycle ALU operations. Moreover, if the IRB is to behave as a functional unit and provide multiple reads and updates in a cycle, it has to be a multi-ported unit. This implies that each read-port of the IRB needs to have a result-forwarding bus to broadcast the value (on a hit) to the dependent instructions in the issue window. This overhead is similar to the overhead of increasing the issue width, due to the extra match-lines and comparators in each cell of the issue window to check for input operand availability. This increase has a *quadratic* effect on the delay of the wakeup logic and the data-bypass logic [1], both of which are on the critical timing path of modern superscalar processors [33].

As noted earlier, in our design, the IRB is not treated as a functional unit/ALU as far as the issue logic is concerned. Further, we can avoid the inclusion of the extra logic and delays of data forwarding from the IRB by observing an interesting property of Dual Instruction Execution. Note that in parallel with the IRB access, the duplicate instruction has made its way through the normal pipeline into the issue window, i.e., both primary and duplicate instructions have entries in the issue window (refer to Figure 3.3). Since we anyway have the output values (register updates) from the execution of the primary stream's instructions on the

functional units forwarded to the issue window to wake up waiting primary stream instructions, why not use the same forwarding of information to wake up waiting duplicate stream instructions as well, i.e. *the results from the primary stream can be used to wake up waiting instructions from both primary and duplicate streams in the issue window.* Whenever a duplicate instruction gets its operands (from a primary instruction output), it performs the reuse test as mentioned earlier. If the reuse test passes, it picks up the result, and directly proceeds to the commit point without propagating any results to the issue window. Consequently, we do not need any data forwarding to occur from the IRB, thereby benefiting from the additional ALU bandwidth provided by the IRB, without complicating/extending the wakeup and bypass logic. It is to be noted that we can achieve this complexity-effective design without compromising on reliability, as we will show shortly.

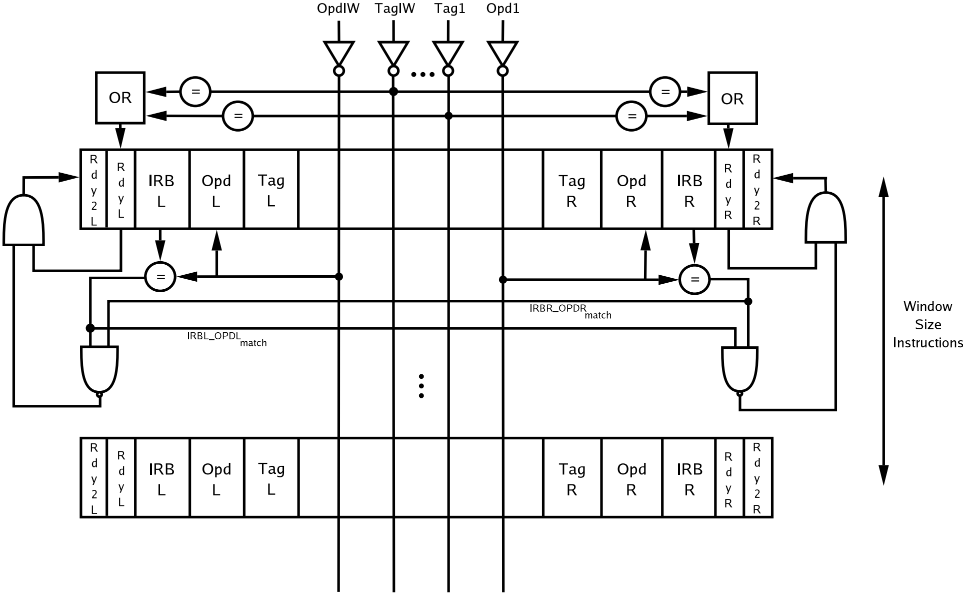


Figure 3.5. Design of issue window wakeup logic to support input-operand forwarding for DIE-IRB. Note that *IRB L/R* come into issue logic together with the instruction.

The design of the issue logic to support input operand forwarding for DIE-IRB is shown in Figure 3.5. We use a data capture instruction scheduler [43], where the tag lines (identifying the operand) and operand line (containing the propagated values) are broadcast down the issue window. Normally, each entry of the window has the operand values, and flags (*RdyL* and *RdyR*) to indicate that the operands are available (i.e. the instruction is ready to issue if both these flags are asserted).

Our enhancement to the issue window does not require any more entries beyond what is already available in the original SIE/DIE designs. Instead, each entry in the issue window is augmented with an extra field for each operand to hold the value that has been read from the IRB, designated as *IRBL* and *IRBR* for the left and right operands respectively. In addition, we use 2 additional flags, *Rdy2L* and *Rdy2R*, to indicate whether the operands are available and the instruction is ready to be scheduled. These flags are asserted when (i) the operands are available, and (ii) at least one of the operands do not match the corresponding *IRBL* or *IRBR* values retrieved from the IRB (i.e. this instruction needs to be serviced by the ALUs). Note that the latter condition is the IRB reuse test, and we need two extra comparators for each issue window slot, and the comparison can be performed in parallel with the updating of the values into the *Opd* fields. These flags are then used to determine whether this instruction can be scheduled (instead of the original *RdyR*/*RdyL* flags), and their assertion logic is given below:

$$Rdy2L = (\overline{IRBL_OPDL_{match}} \vee \overline{IRBR_OPDR_{match}}) \wedge RdyL$$

$$Rdy2R = (\overline{IRBL_OPDL_{match}} \vee \overline{IRBR_OPDR_{match}}) \wedge RdyR$$

When a primary instruction comes to the issue window, two entries are created (one for primary and another for duplicate). In the case of the duplicate, the *IRBL* and *IRBR* entries are filled with values from the IRB, whose lookup is complete by this time. In the case of the primary, one can think of these as having values which are never going to match the operand values propagated to this issue window entry (i.e. it will always execute on the ALUs). Though not explicitly shown in the figure, the result value from the IRB can directly propagate to the ROB/rename register file by the time the instruction moves to the issue window (even before the reuse test is performed). If the operands match for the duplicate, then it can directly move to the commit stage without using the ALUs. As seen from the figure, our enhancements impose little additional complexity to the issue logic.

Implementing Forwarding in Non Data-Capture Schedulers: In a scheduler that does not employ data-capture, the result tags are broadcast down the issue window and the operands are obtained from the register file. Further, the register file is read *after* the selection process. Therefore, in order to fetch the operands, the instruction scheduler would have to allocate a functional unit to instructions in the duplicate stream as well. Though this allows for a duplicate stream instruction to execute in a functional unit immediately if there is a miss in the IRB, in case of an IRB hit, the allocated functional unit is not used. Further, it cannot be re-allocated to any other ready instruction in the issue window, as any such instruction would first have to perform register file access.

A possible technique to overcome this limitation is to perform the selection *after* the register file lookup, effectively decoupling the wakeup and selection steps. Previous studies have shown that the wakeup and selection steps can be pipelined with very little impact on the overall IPC [3, 53]. In [3], the authors show that pipelining can be achieved by dividing the issue mechanism into a single-cycle wakeup-step and a multi-cycle selection-step. A similar approach can be taken in our case, where, both primary and duplicate instructions can be woken up and made to access the register file. The reuse-test is performed immediately following the register file access for the instructions in the duplicate stream, which may be performed within the same cycle as the register file access. On completion of this step, the ready instructions are assigned the ALUs. Any instruction that did not get an ALU is re-scheduled using a method similar to that described in [3].

3.3.4 Redundancy Characteristics of DIE-IRB

Having covered the design aspects of our proposal, we now discuss its temporal redundancy (reliability) properties. There are two scenarios to consider: (i) the duplicate instruction misses (either PC miss or reuse miss) in the IRB, and (ii) the duplicate instruction hits in IRB. For each of these scenarios, there are two further input operand possibilities: (a) the operands are not provided by any prior instruction (i.e. they are not waiting for any values to be propagated), and (b) they (both primary and duplicate instructions) get one or both operands from another primary instruction.

For the combination of (i) and (a), the primary and duplicate streams will execute on functional units, and our DIE-IRB system is not any different than the original DIE, thereby providing the same redundancy properties. In the case of (ii) and (a), even though the duplicate stream went through the IRB, its result is still compared with that for the primary stream which executed through the functional units. Further, note that in order to create the entry in the IRB, an earlier execution of the same instruction (both primary and duplicate) should have gone through the functional units (and their outputs would have been compared). Only the errors that struck the IRB (after the entry was inserted) to produce an output value which exactly matches the output of a primary instruction that experienced another error during the execution on a functional unit, may not be caught (the resulting probability is comparable to that for the original DIE).

Moving to possibility (b) where the operand values are being forwarded from a prior primary instruction, a pictorial depiction of the situation is given in Figure 3.6. The original DIE [39] has forwarding only between instructions of its own stream (Figure 3.6 (a)). However, in our enhancement, forwarding is done only by the primary instructions, which are sent to waiting instructions from both streams. If there is no error occurring in the forwarding itself, then the reliability characteristics are no different from the (a) possibility analyzed above. However, if an error does occur in this forwarding mechanism, it can happen in two ways that are logically shown in Figures 3.6 (b) and (c). If the error propagates to only one of the streams (either primary or dual, but not both as in Figure 3.6 (b)), note that the outputs from both instructions (i.e. instruction `add r4 <- r3,#1` in the example) are anyway checked at commit point to ensure they produce the same result.

On the other hand, the situation that we want to address is shown in Figure 3.6 (c) where the error propagates to both streams. This could happen in a data-capture instruction scheduler, since the result forwarding from the functional units to the issue window and that to the rename register file use separate paths [43]. Therefore an error on the forwarding path to the issue window could go undetected. In order to provide protection under this condition, we propose a minor enhancement to provide additional (spatial) redundancy for these paths (see Figure 3.7). For instance, by having a duplicate path (which need not traverse through the

issue window) the broadcasted tags and data can be compared with that on the original path after the last entry has received it. This adds an extra comparison at the end of the broadcast but the overhead of this operation is relatively small.

To recover from any detected errors, we do not need any special hardware or pay additional performance costs when incorporating the IRB into the previous DIE design [39, 56].

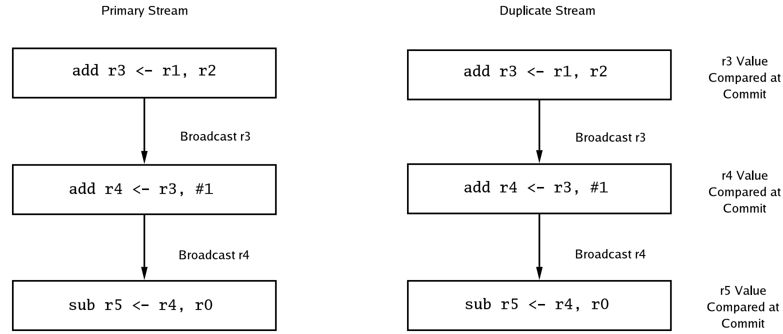
3.3.5 Minimizing Conflict Misses in the IRB

We use saturating counters, similar to those used in prediction structures like the Branch History Table [59], to reduce the conflict misses in the IRB. In our default configuration we use a 4-bit saturating counter, which is initially set at a value of 7. If any instruction accesses the IRB and finds that the entry to which it maps to is currently occupied by another instruction, then the value of the counter is decremented by one. If it, however, finds that the entry is occupied by the same PC but there is an operand mismatch, the value of the counter is decremented by 5 (this instruction is less likely to have the same operands in the future, and should be a less favorable IRB entry). On the other hand, if there is an IRB-hit, the value of the counter is incremented by 5. When the counter-value reaches zero, the entry is a candidate for replacement. In essence, the rate at which the counter degrades to 0 is an indication of the usefulness of that entry.

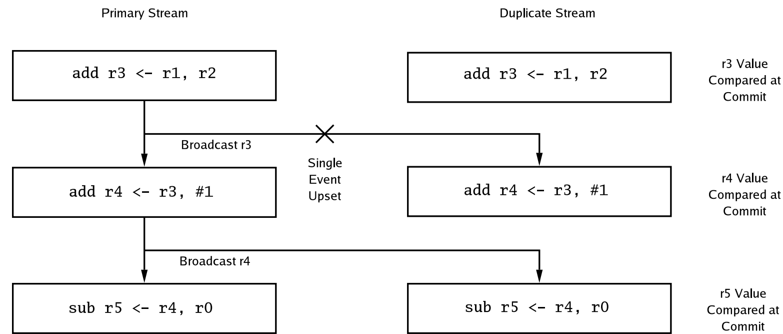
3.4 Experimental Results

3.4.1 Simulation Platform and Workloads

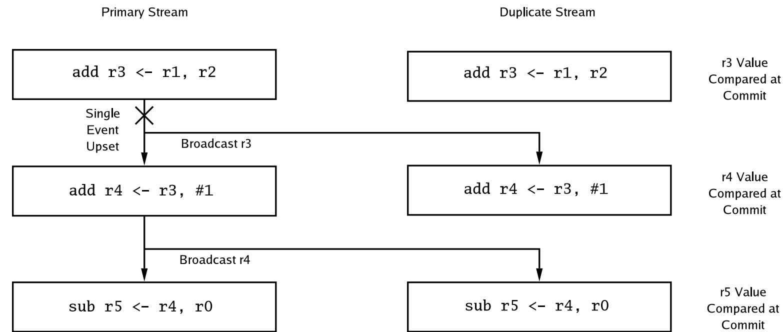
Our experiments were carried out using SIE, DIE and DIE-IRB processor models built using the SimpleScalar 3.0 framework[4]. The default simulation parameters that we use are given in Table 3.1. For the workloads, we used 12 benchmarks from the SPEC CPU2000 suite. The benchmarks were compiled for the PISA instruction set architecture with the `-O2 -funroll-loops` optimization flags. We used the reference input set for the simulations. Each benchmark was fast-forwarded by 1 billion instructions and then detailed simulation of the next 1 billion instructions was performed.



(a) Value forwarding within a stream



(b) Value forwarding only from primary with errors propagating to one

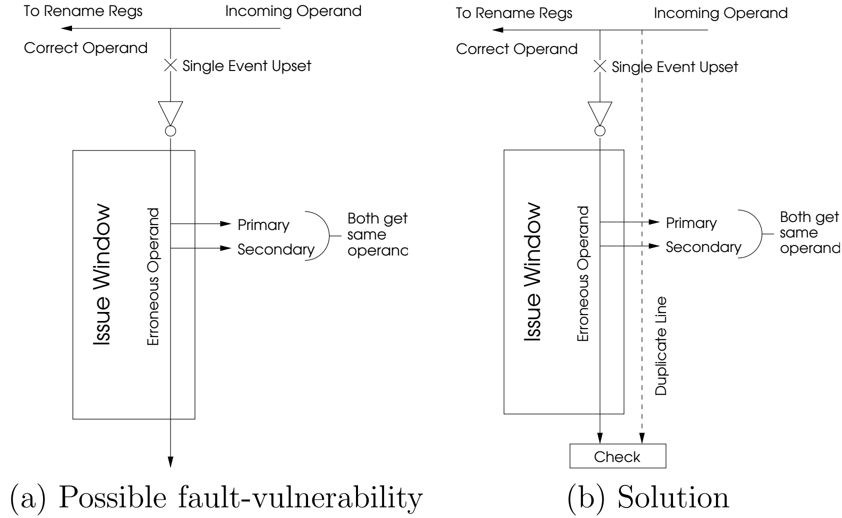


(c) Value forwarding only from primary with errors propagating to both

Figure 3.6. Value forwarding within/across instruction streams.

Our focus in the subsequent evaluations is to reduce the performance gap between SIE and DIE under error-free circumstances.

Consequently, the metric that we focus on is the percentage reduction in the gap between these two with the different enhancements. The IPCs themselves



(a) Possible fault-vulnerability (b) Solution

Figure 3.7. Handling faults on the result-forwarding bus using duplicate forwarding paths.

in the base SIE, and that for the DIE execution, together with the percentage degradation of the latter (referred to as the *gap* henceforth), are given in Table 3.2.

3.4.2 Benefits of DIE-IRB

Figure 3.8 presents the percentage of IPC gap between SIE and DIE recovered with our scheme, denoted as *DIE-IRB-1K-sat*, which uses the default 1K entry IRB parameters given in Table 3.2 with the saturating counters. In addition to our DIE-IRB scheme, we also present the benefits obtained by doubling the ALUs (the *DIE-2xALU* scheme). The third bar gives the percentage of IPC gap recovered using our scheme where we artificially set the hit-rate to be 100% in the IRB (i.e. all duplicate instructions go through it), denoted as *DIE-IRB-ideal*. The second and third bars thus give an approximate estimate of what we can hope to gain by removing the ALU bottleneck of DIE, and how best we can do with our IRB scheme, respectively. In Table 3.3, we give the characteristics of the IRB behavior in the *DIE-IRB-1K-sat* execution. The number of references by the duplicate stream to the IRB is broken down into those that do not have a PC match, those that have a PC match but fail the reuse test, and those that have a IRB hit.

We find that the DIE-IRB scheme can recover between 2% (for *art*) to 42% (for

Processor Parameters

Fetch/Decode/Issue/Commit Width	8
Fetch-Queue Size	8
Branch-Predictor Type	Combined predictor with 16K meta-table, 16K L1 and L2 tables 11-bit history-width XORed with address in L2 predictor
RAS Size	64
BTB Size	2K-entry 2-way
Branch-Misprediction Latency	7 cycles
RUU Size	128
LSQ Size	64
Integer ALUs	4 (1-cycle latency)
Integer Multipliers/Dividers	2 (3,20)
FP ALUs	2 (2)
FP Mult./Div./Sqrt.	1 (4,12,24)
L1 Cache Ports	2
L1 D-Cache	32KB, 2-way with 32B line-size (2)
L1 I-Cache	64KB, 2-way with 32B line-size (2)
L2 Unified Cache	512 KB, 4-way with 64B line-size (12)
TLB Miss-Latency	30 cycles
Memory Latency	112 cycles
Processor Clock-Frequency	2 GHz
Process Technology	180nm

IRB Parameters

Number of Entries	1024
Associativity	Direct-Mapped
Access-Latency	3-Cycles (Pipelined)
Number of Saturation-Counter Bits	4
Decrement-Value on PC-Miss	1
Decrement-Value on Reuse-Miss	5
Increment-Value on Reuse-Hit	5

Table 3.1. Default simulation parameters. Latencies of ALUs/caches are given in parentheses. All ALU operations are pipelined except division and square-root.

gzip) of the IPC that is lost due to redundant execution, with the average being around 23%. Several factors contribute to the effectiveness of the IRB in bridging the performance gap between DIE and SIE: (i) the demand for the ALU resources itself, (ii) the reuse hit rate in the IRB, and (iii) the occurrence of long latency ALU operations, and how they can be satisfied by the IRB. The relative impact

Benchmark	IPC_{SIE}	IPC_{DIE}	$\frac{IPC_{SIE}-IPC_{DIE}}{IPC_{SIE}}$ (%)
164.gzip	1.5069	1.2115	19.60
171.swim	1.9807	1.7615	11.07
175.vpr	1.2098	0.9968	17.61
176.gcc	1.2764	1.0405	18.48
177.mesa	2.2230	1.5403	30.71
179.art	0.7316	0.4210	42.45
181.mcf	0.3216	0.2964	7.84
183.equake	1.7404	1.2547	27.91
188.ammp	0.1082	0.1066	1.66
197.parser	1.2685	1.0306	18.75
255.vortex	2.3868	1.6196	32.14
256.bzip2	1.8070	1.2941	28.38

Table 3.2. IPC under SIE and DIE, and their gap, for simulated SPEC2000 benchmarks.

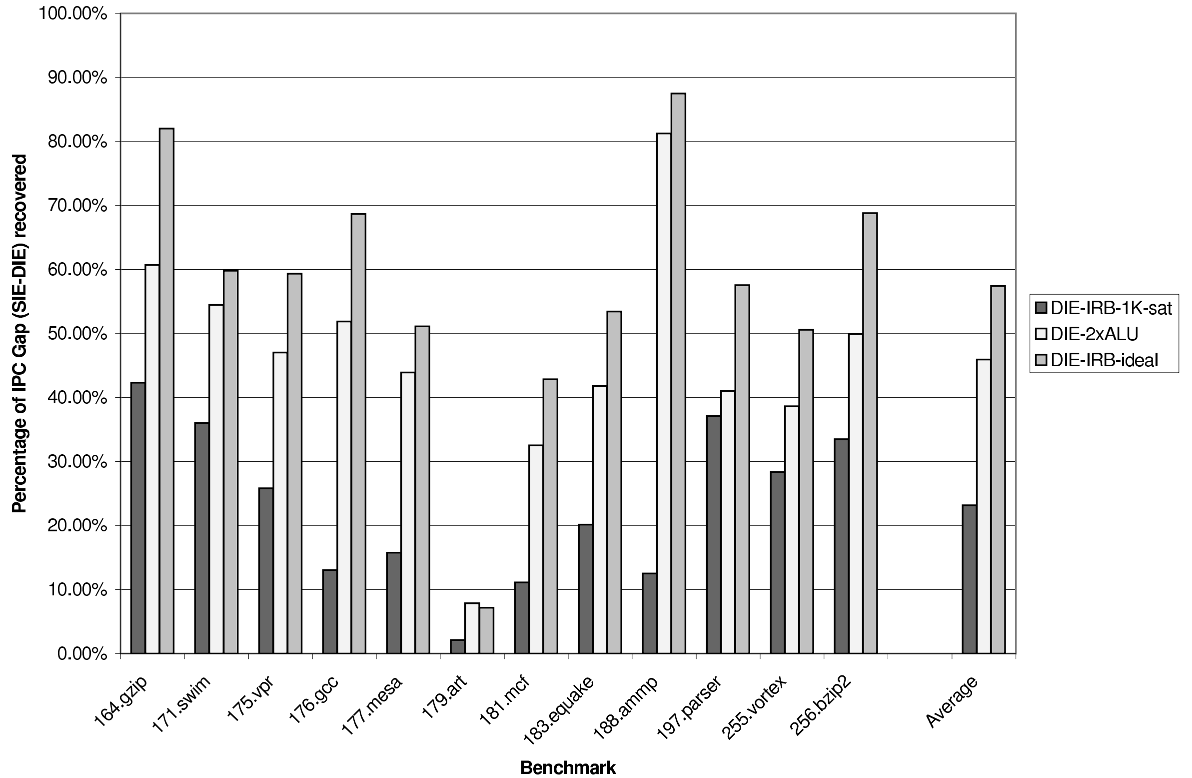


Figure 3.8. Percentage of IPC Gap Recovered from DIE using Instruction-Reuse. The y-axis plots the value of $\frac{IPC_{scheme}-IPC_{DIE}}{IPC_{SIE}-IPC_{DIE}}$.

of these factors on the applications is noted below:

- In applications such as *gzip*, *swim*, *parser*, *bzip2* and *vortex*, we are not only getting more than 25% reduction in the IPC gap, but we are getting much of

the IPC gains of doubling the number of ALUs. These are applications with high ALU demands (as evidenced by the boost in performance by doubling the ALUs - the bar for *DIE-2xALU*), which at the same time exhibit good instruction reuse, as seen by the IRB hit rate numbers given in Table 3.3 (close to 40% or higher).

- On the other hand, in *gcc* where even though the ALUs are in demand, the IRB hit rate is less than 25%, thereby leaving a considerable gap between *DIE-2xALU* and *DIE-IRB-1K-sat*.

- There are a few applications where the IRB hit rates in Table 3.3 seem reasonable, but they still do not provide as much IPC improvements since the ALU bandwidth may not be a significant bottleneck. For instance, in *art* (and perhaps in *mcf* to a certain extent), ALUs are not that much of a problem (note that doubling the ALUs only provides 8% IPC improvement in art). As noted earlier in section 3.2, this is due to the low ILP in this application, and there are not enough ready instructions to exploit the available bandwidth, thereby limiting the benefits of an IRB. In the case of *ammp*, the gap between SIE and DIE is itself not very significant (only 1.6%), and the absolute difference between an IRB based approach and doubling the ALUs is quite small.

- In a few other applications, despite the ALU limitations and the reasonably good IRB hit rates, there is still a gap between what doubling the ALUs can achieve, and what is provided by the IRB. This is a consequence of how the long latency ALU operations are executed. Let us consider the case of *mesa*, where despite the 44% IRB hit rate, the benefit for *DIE-2xALU* is more than twice that of *DIE-IRB-1K-sat*. This is because, of the long latency ALU operations, only 5.6% go through the IRB (compare this with *swim* in which we found around 18% long latency operations serviced by the IRB, even though its overall IRB hit rate is even lower than *mesa*). Collapsing of long-latency operations is also one of the reasons why *DIE-IRB-ideal* does even better than *DIE-2xALU* (another reason is the fact that *DIE-IRB-ideal* does not contend for issue ports, unlike *DIE-2xALU*). A consequence of this could be the earlier resolution of branches.

Overall, we find that the IRB does a fairly good job of bridging the gap between SIE and DIE, while not incurring the hardware complexity of design involved in doubling the number of ALUs. On the average, we obtain 23% reduction in IPC gap between SIE and DIE, which is over 50% of that provided by doubling the number of ALUs. At the same time, the performance of *DIE-IRB-ideal* indicates that there is more room to grow for this scheme, if we can somehow enhance its hit rate (as is investigated in the next subsection).

Benchmark	**PC miss**	**IRB miss**	**IRB hit**
164.gzip	8.46	49.44	42.09
171.swim	3.29	56.71	39.98
175.vpr	38.70	23.75	37.54
176.gcc	28.70	46.33	24.96
177.mesa	39.62	16.18	44.19
179.art	0.01	59.19	40.79
181.mcf	0.02	64.37	35.60
183.equake	5.34	60.69	33.96
188.ammp	8.57	54.03	37.39
197.parser	9.95	36.41	53.63
255.vortex	37.55	23.59	38.84
256.bzip2	0.09	56.42	43.48

Table 3.3. Classification of IRB-Accesses

3.4.3 Enhancing IRB Reuse Characteristics

Techniques for enhancing the hit behavior of the IRB could prove to be useful in alleviating ALU bandwidth even further (note that *DIE-IRB-ideal* shows we have much to gain by improving IRB hit behavior). There can be three different ways of achieving this: (i) We can enhance the IRB configuration to hold more (possibly useful) entries to reduce capacity/conflict misses (i.e. PC-misses), (ii) Even if we have to be confined to a small IRB (to meet timing requirements) we can try to make sure only useful entries are maintained in the limited space of the IRB so that non-useful entries do not evict useful ones (reducing both PC-misses and reuse misses), and (iii) We can also proactively insert entries into the IRB to anticipate what would be needed (reducing reuse misses). We explore these three issues in the next three sets of experiments. In the interest of clarity, we focus on 6 benchmarks - *swim*, *vpr*, *gcc*, *equake*, *vortex*, and *bzip2* - for the rest of the experimental results.

3.4.3.1 IRB Configurations

An intuitive way of reducing PC-misses would be to increase the IRB capacity and/or associativity. However such enhancements can affect access latencies, thereby not being able to meet our timing constraints for the required number of ports. In order to examine whether we really need large IRB sizes/associativities, we conduct experiments with different IRB sizes (256, 1K and 1M entries) and associativities (direct-mapped, 2-way and 4-way) and present the resulting performance consequences in Figure 3.9. In these experiments, we still assume (perhaps very optimistically for a 1M IRB) the access to be performed within a 3-stage pipeline as discussed earlier, since we are examining how well our 1K IRB used until now, compares with one that has much higher capacity (1M). Note that there are no saturation counters being used in this experiment.

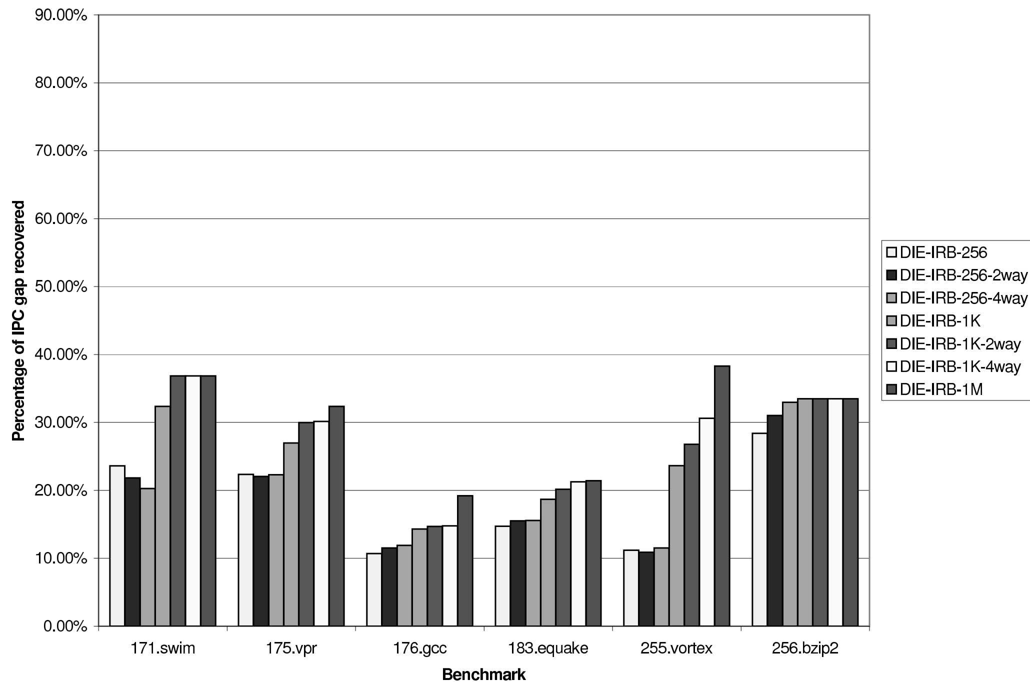


Figure 3.9. Impact of IRB Configurations

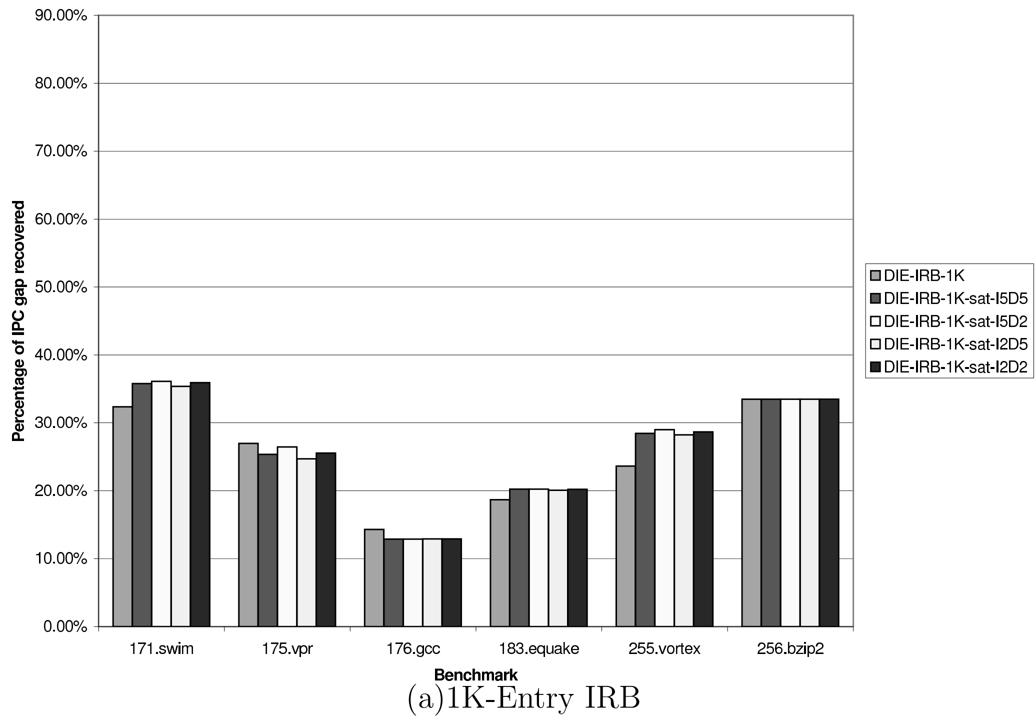
When we go from a 256 to 1K entry IRB, we get good performance improvements (especially for *swim* and *vortex*) due to a larger number of reused instructions being cached within the IRB. This results in lower PC-misses to the IRB. However, going from an IRB size of 1K to 1M provides lesser gains in the recovered IPC.

It is possibly not worthwhile extending the IRB access times with such a large configuration simply to bridge this small gap. Our choice of a 1K-entry IRB is a reasonably good configuration considering these trade-offs, and we have found this to provide the desired number of ports within the required timing constraints. An exception to this is *vortex*, where, there is an improvement of nearly 50% in bridging the gap between SIE and DIE when a 1M-entry IRB is used (compared to 1K). This motivates the need for a possibly better management of the given space (1K) for the IRB, as is conducted in the next subsection, rather than looking to expand its capacity. Incidentally, earlier proposals [50, 60] for the IRB in the context of a SIE, have also suggested a size of 1K entries.

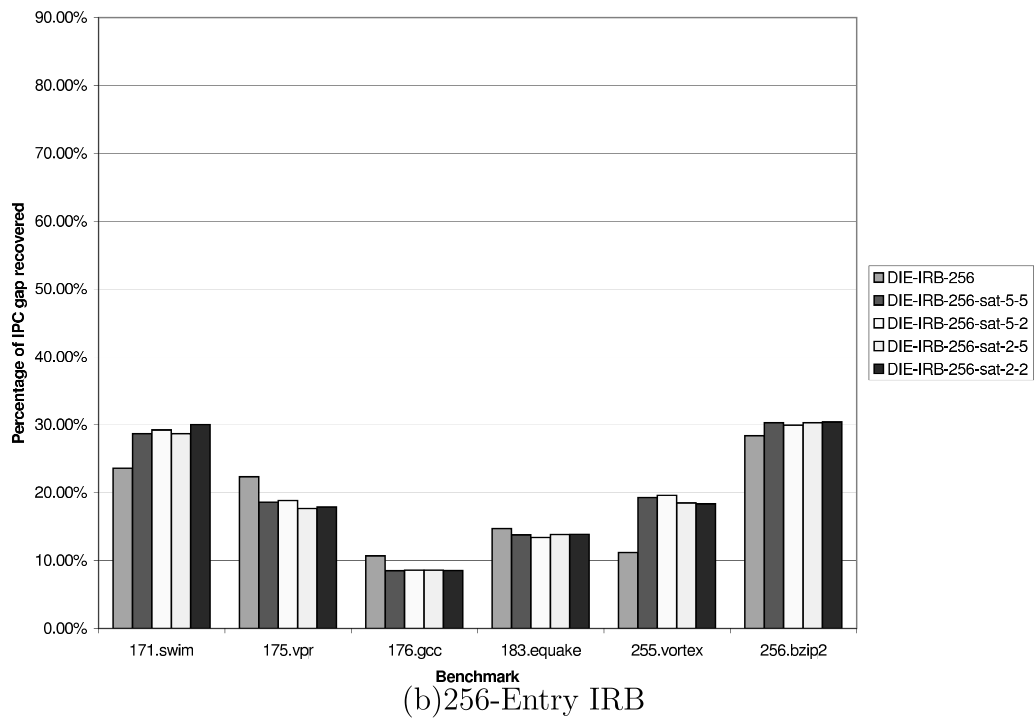
The impact of associativity is not as significant as that for capacity, especially at smaller sizes (for 256 entries, capacity is more the problem, while for 1K entries the importance of conflicts may creep in). Again, except for *vortex*, the impact of associativity at the 1K size is not very significant (and can again increase access times), and consequently we simply suggest using a direct-mapped 1K entry IRB, and manage it better as described next.

3.4.3.2 Saturation Counter Management

Within the given IRB space, we would like to ensure that only useful entries are kept, and non-useful entries do not evict the useful ones. As mentioned in Section 3.3, we use a 4-bit saturation-counter, similar to how branch history tables handle this problem. A PC-miss decrements the counter by 1 whereas a reuse-miss or hit decrements and increments the counter by 5 respectively. When the counter-value of an entry reaches zero, it becomes a candidate for replacement. We considered three other schemes, namely, *DIE-IRB-1K-sat-I5D2*, *DIE-IRB-1K-sat-I2D5*, and *DIE-IRB-1K-sat-I2D2*. The *DIE-IRB-1K-sat-I5D2* scheme attempts to keep an entry that has been reused within the IRB longer by incrementing the value of the counter by 5 on a reuse-hit and decrements it only by 2 on a reuse-miss (PC-misses decrement the counter by 1 as before). On the other hand, the *DIE-IRB-1K-sat-I2D5* scheme tends to make entries that have failed the reuse-test candidates for eviction sooner by decrementing the counter by 5 on a reuse-miss and incrementing it by 2 on a hit. The *DIE-IRB-1K-sat-I2D2* scheme uses a finer-grained increment/decrement value of 2.



(a)1K-Entry IRB



(b)256-Entry IRB

Figure 3.10. Impact of various Saturation Counter Management Schemes

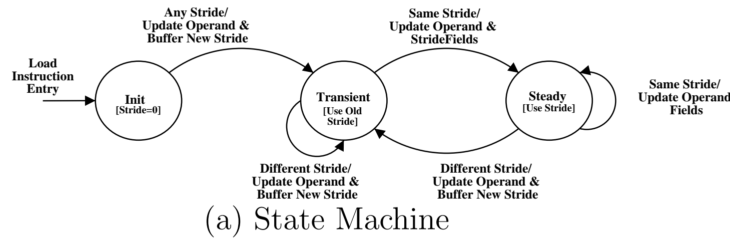
The percentage of IPC recovered by using such IRBs for 1K entries and 256 entries are given in Figure 3.10. The bar for *DIE-IRB-1K* denotes an IRB execution without any saturating counters. We expect saturation counters to be useful when the conflicts really become a performance bottleneck, and perhaps not very useful in the other cases (or maybe even an overhead to a certain extent for those). We find that this is the case for many of the applications in these results, and the benefits of the saturating counter are really felt in an application like *vortex* where the 1K IRB was not doing as well as a 1M IRB. We find that the saturation counter based IRB reducing the SIE to DIE gap by 29% compared to the 24% without the saturating counters for *vortex*.

3.4.3.3 Proactively Inserting Entries into the IRB

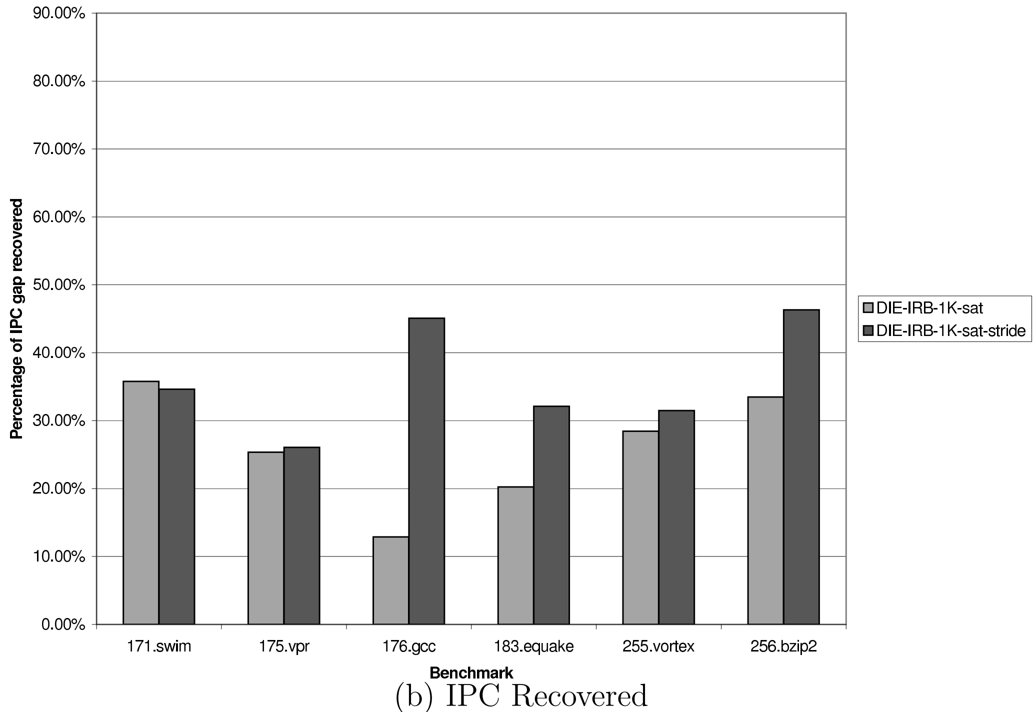
In the instruction reuse technique used until now, we have focused on identical instances of the dynamic instructions in the program. Whenever operands change, even if there is some regularity in the change, such instructions cannot benefit from the IRB. If one can anticipate what operands an instruction would use, then an appropriate entry could be inserted in the IRB, ahead of encountering this instruction. Please note that our scope in the following experiment is to only investigate some of the benefits of this approach, and we make an idealistic assumption about the costs (both hardware and time overheads) as explained toward the end.

Previous studies [57, 62] have shown the presence of *strides* in the register data flow of programs. We attempt to enhance the IRB hit rate by exploiting strides in the operand values using the following hardware. In addition to holding the value for each operand (encountered the previous time) as before, each IRB entry also holds a stride field for each operand. When an instruction is encountered for the first time, the stride fields are initialized to zero. When the next dynamic instance of the instruction maps to the IRB, the stride of the operands (which is the difference between the values in the operand fields of the IRB and those in the current instance) are calculated. The next time the same instruction is encountered, its strides are again calculated. If they do not match, then the IRB functions as previously described (i.e. only the strides are updated). On the other hand, if they do match, after servicing this instruction we replace its operand values by adding the strides to them (i.e. we anticipate the operand values for

the next time), and we also perform the ALU operations on these operands to get the result, and store this in the result field of the IRB entry. The next time this instruction arrives with the anticipated strided behavior, it would encounter a reuse hit, and can pick up the output directly from IRB. The state machine for this mechanism is shown in Figure 3.11(a), which is similar to the one given for the stride-based value-predictor given in [57]. Note that the earlier IRB without strides is a degenerate case of this mechanism (strides of value 0).



(a) State Machine



(b) IPC Recovered

Figure 3.11. IRB Stride Predictor

We conduct a simple experiment (to gauge its potential), by assuming there is separate hardware to perform the necessary stride/result computations in the IRB, and those results are obtained before the next dynamic instance of that instruction. Figure 3.11(b) gives the IPC benefits that could be obtained using this mechanism.

While this is not universally useful (not all applications exhibit such strided register value behavior), we do find substantial benefits in some others. For instance, the IPC improvement in *gcc* is nearly 3 times better than what we were getting for the original IRB. *equake* and *bzip2* also show potential benefits for such anticipated IRB entry creation. This mechanism can reduce the IRB misses (and increase the hits) due to dynamic instances of an instruction with its input operands following a strided pattern. The number of PC-misses is relatively insensitive to using this predictor, as their behavior is dictated more by the policies pertaining to the saturation counter management. For *gcc*, using this mechanism, we are able to reduce the IRB-misses to 18.5% and get a hit rate of over 52% (compared to 46.33% and 24.96% shown earlier in Table 3.3. On the other hand, for *vpr*, IRB-misses are reduced to around 22% (from 23.75%) and hits increase to 40% from 37.5%, whereby the overall benefits obtained is small. For *swim*, there is a slight reduction in the performance. This is especially interesting since the predictor mechanism actually reduces the IRB misses to 50% (from around 57%) and gives a corresponding increase in the number of hits. Upon a detailed investigation of the execution statistics, we found that it is possible for an instruction with good stride behavior to evict another from the IRB which happens to be in a critical path. We noticed that the hit rate of floating point compare operations went down with the stride predictor, which in turn increased the latency for branch resolution (impacting mis-speculation costs in *swim*). However, on the average we find more positive effects due to such proactive entry insertion.

In this experiment, our point was merely to note that there could be steps undertaken to enhance the instruction reuse coverage provided by the IRB, to further mitigate the IPC loss of DIE (and we used an idealized simulation framework in the evaluation). While this does require additional ALUs and comparators (to compute and compare strides, and compute results), such hardware resources need not be coupled with the ALUs in the execution engine of the main datapath and thus may not be on the critical path of the processor. We require only as many of these units as the number of write and read-write ports for the IRB (which in our case is 4). Finally, this mechanism can be undertaken on a best-effort basis, i.e. if the strides can be anticipated and the corresponding outputs available before the next dynamic instance of that instruction, then it would be useful (else, one would

go to the normal ALUs as before).

3.5 Concluding Remarks

This chapter has proposed a novel solution to alleviate the performance loss incurred when implementing instruction-level temporal redundancy in an out-of-order superscalar processor. As in earlier studies, we observed between 1% to 43% loss in IPC, in a temporally redundant (dual) execution (called DIE) over a single instruction execution (SIE) for a set of SPEC CPU2000 benchmarks. An important contributor to this performance gap is the insufficient ALU bandwidth, since the same number of ALUs provisioned for an SIE core needs to handle twice the number of instructions imposed on a DIE. Increasing the number of ALUs may not be an easy solution to this problem since it increases the complexity of the issue and bypass logic.

We propose a novel adaptation of an existing idea, Instruction Reuse (previously examined to improve single stream execution), to alleviate the ALU bandwidth problem on a DIE. The primary instructions always use the ALUs. However, the duplicate instructions are directed to an instruction reuse buffer (IRB) from where they can directly pick up the result (instead of execution on ALUs) as long as that instruction has been previously encountered with the same set of operands. Else, the duplicate instructions are directed to the ALUs as in a normal DIE. This mechanism is able to provide the required temporal redundancy semantics without requiring any extra protection from faults for the IRB. The temporal redundancy is being provided by instruction reuse rather than re-execution.

In our proposal, the issue logic does not manage the IRB (its lookup is done ahead in the pipeline) and thus does not increase the scheduling complexity. At the same time, we exploit a unique feature of DIE, wherein the primary stream can forward data values to waiting instructions for both primary and secondary streams, thereby not requiring us to complicate the wakeup and bypass logic.

Through detailed simulations, we show that our proposal can provide between 2% to 42% reduction in the IPC gap between SIE and DIE, with an average of around 23% over 12 SPEC2000 benchmarks.

Chapter 4

Locality-Triggered Slice-Based Redundant Threading

4.1 Introduction

In the previous chapter, we showed how Instruction Reuse, a form of value locality, can be used to provide partial redundancy in an attempt to reduce the performance overhead associated with full redundancy. One of the problems we faced during the implementation of DIE-IRB was to maintain correct dataflow in the redundant thread/stream when only a subset of instructions were being executed. We resolved this issue by making the primary thread forward outputs to both the primary and redundant threads, but the tightly-coupled execution nature of DIE made this relatively simple to implement. In a more loosely-coupled SMT-based redundant execution system such as SRT, this is harder to achieve without causing redundancy violations. Secondly, the per-instruction nature of dynamic instruction reuse results in the IRB needing to be a high-bandwidth array structures with several read/write ports. We also note that most real systems typically do not require full redundancy, but must instead meet a certain reliability budget. Performance-efficient transient fault tolerance techniques with near-perfect (but not necessarily 100%) reliability characteristics are, therefore, attractive points in the performance-reliability tradeoff space.

We present in this chapter a *partial redundant threading* paradigm that is based

on two key ideas, (i) Slice-based redundant execution, and (ii) Value and Control-flow Locality exploitation.

A redundant threading system can be viewed as a primary thread generating outputs – Stores – that emanate from the processor, and a redundant thread verifying the integrity of these outputs. The redundant thread can be further envisioned not as a sequence of dynamic instructions, but as intertwined dependency chains or slices of instructions that ultimately lead up to these Stores. If a partial set of instructions needs to be chosen for redundant execution, then it makes sense to do this at the granularity of such *backward slices* of store instructions. This avoids expensive redundancy-violating techniques to copy state across threads. We propose the design of a simple and efficient backward slice extractor that is able to identify the set of instructions that lie on the backward slices of a selective set of stores, thereby enabling partial threading at the slice granularity.

With this mechanism in place, we next propose a policy for selecting slices for redundant execution. We use results from the extensively researched area of value locality for this purpose. Of particular interest to us is the fact that store addresses and data have been shown to exhibit good predictability [24]. We use predictors to verify store addresses and data, and redundantly execute only those slices where predictions fail. Thus, predictability/locality is itself being exploited as a redundancy mechanism for the non-selected slices. In addition to stores, we also track backward-slices of branches and use branch predictability to avoid redundant execution of certain slices. We show that even with very simple predictor configurations, a substantial number of slices can be eliminated while maintaining an extremely low error vulnerability.

We call our design that incorporates these two ideas **SlicK** [35] (short for Slice-Kill), and present the detailed implementation and evaluation of SlicK on a Simultaneously and Redundantly Threaded (SRT) processor baseline. We evaluate the performance of SlicK on a cycle-accurate simulator using all applications from the SPEC CPU2000 suite. We evaluate the error coverage of our design using the Architectural Vulnerability Factor (AVF) [31] approach.

SlicK's locality-triggered slice-based execution paradigm enables it to:

- maintain an AVF of 0%-2% for critical processor structures while performing 10.6% better than SRT, buying back over 50% of the performance loss

incurred due to redundant threading;

- achieve this with an efficient set of structures that are simple to implement and off the processor's critical paths;
- provide a useful baseline for further exploration (via *policies*) of the performance-reliability design space by providing the basic *mechanisms* for selective elimination of slices.

4.2 Related Work

Other works that attempt to adapt locality principles to reliability include [19], which observes that soft errors often manifest themselves as mispredictions in warmed-up predictors, and uses the misprediction recovery interval to perform more extensive error detection checks. In [58], the authors exploit the fact that soft errors could result in a range of observable symptoms such as ISA exceptions and branch mispredictions, and use these symptoms to trigger a rollback and re-execution from a previously stored checkpoint.

Slipstream [54, 22] is a performance-enhancing technique that runs a speculative prefetch A-thread ahead of the main R-thread. Instructions that are detected to be *ineffectual* (i.e., not required for correct forward progress) are removed from the A-thread. Slipstream achieves a certain amount of fault-tolerance due to the redundant computation being performed by the two threads. The "ineffectuality" of instructions in the A-thread is in fact a result of the locality that exists in the instruction stream.

SlicK differs from all of these approaches in its explicit use of Predictors to provide redundancy for entities that exit the Sphere of Replication: Stores. Our slice-based microarchitecture designed around this principle achieves significant performance benefits with extremely low vulnerabilities and provides an attractive baseline for further exploration of the performance-reliability design space.

Forward and backward-slice extraction and associated optimizations have been explored extensively in the past for a variety of applications [12, 13, 29, 42, 54, 63], both in hardware and software.

4.3 SlicK

4.3.1 SlicK Overview

SlicK executes the leading thread in its entirety, exactly as in SRT. For the trailing thread, it uses a set of predictors to attempt to verify the outputs of the leading thread without re-execution. The only trailing thread instructions that are executed are those that belong to the backward slices of outputs that the predictors could not verify. This results in a partial redundant threading solution where all outputs from the SoR have been verified in some manner, but the actual number of instructions that are redundantly executed is significantly lowered, thereby reducing resource contention and enhancing performance.

In the following subsections, we progressively walk through the steps required to build a SlicK system. First, we define the notion of trigger instructions based on which slices are identified. Next, we describe the design details of a slice-extraction microarchitecture that identifies instructions belonging to slices of these triggers. Finally, we show how to integrate these structures into an SRT microarchitecture to complete the SlicK design.

4.3.2 Identifying Trigger Instructions

The first step in designing a slice-based redundant execution system is to identify *trigger* instructions based on which slices are to be marked and extracted. We define a trigger instruction to be one of the following: (i) a store instruction; or (ii) a branch instruction. A store trigger has a direct consequence on what leaves the SoR. A branch is also considered a trigger point for verification because the control flow path that leads to these stores needs to be verified as well. For instance, it is possible for both taken and not-taken paths of a branch to lead to the same store instruction, with each path generating a different address or data for the store. If there is an error in the control flow, both threads would execute this store erroneously.

If the integrity of a trigger can be "verified" (we will elaborate on this shortly), then the instruction itself and its backward-slice need not be redundantly executed and can be dropped or flushed from the trailing thread. We refer to such trigger

instructions as *Flush Triggers (FT)*. On the other hand, when the integrity of a trigger instruction cannot be verified, then that instruction and its backward-slice need to be extracted and redundantly executed. We refer to such instructions as *Execute Triggers (ET)*. It is possible for an instruction to fall on the backward-slices of more than one trigger. If even one of these triggers is an ET, then the instruction would need to be redundantly executed. Note that there are two backward-slices to track for a store instruction, one for the address computation and the other for the store data. Both have independent backward slices that may or may not need to be executed based on the verifiability of the address and the data.

Our verification mechanism for triggers is based on the well-researched topics of branch [59] and store data and address [24] prediction. SlicK uses predictors to attempt to verify the outputs of trigger instructions and provide the required redundancy. The output (store data/address, branch target) of the trigger instruction of the leading thread is compared with that of the corresponding predictors. If the comparison is successful (a "Hit"), then the trigger is categorized as FT. If the comparison is either a "Mismatch", or the predictor is not able to predict due to insufficient confidence ("No-Prediction"), then the trigger is categorized as ET, requiring redundant execution of its backward-slice. Since predictors are essentially storage arrays, it is possible for errors to accumulate in them if entries remain untouched for a long time. Therefore, we recommend parity protecting these structures.

4.3.2.1 Store Predictor

We use independent predictors for Store Data and Store Addresses. Both these predictors are accessed by the PC of the Store Instruction. We primarily employ a simple last-value predictor with 4-bit saturation counters, but also evaluate the benefits offered by a more extensive Finite Context Method predictor [5]. Our results show that a simple 1024 entry direct-mapped last value predictor gives reasonable performance.

4.3.2.2 Branch Confidence Estimator

Traditional Branch Predictors do not give a "No-Prediction" decision. Although modern branch predictors have high hit rates, we wish to convert as many Mismatches to No-Predictions as possible (this enhances fault coverage and is explained further in Section 4.4). We use a simple pattern-based filter for confidence estimation on branch predictions [18]. The filter is constructed from an array of resetting saturation counters and is indexed with the current branch prediction appended to the global branch history.

4.3.3 Extracting Backward Slices

Given a dynamic sequence of instructions comprised of ETs, FTs and non-trigger instructions, we now need an online technique to identify the set of instructions that lie on the backward slices of ETs and FTs and mark them as requiring execution or being flushable. We assume that (i) instructions arrive at the slice extractor in non-speculative (correct path) program order, (ii) decoded architected register identifiers are available for these instructions, and (iii) memory dependencies are handled independently and do not need to be taken into account by the slice extraction mechanism.

4.3.3.1 Slice Extractor Design Goals

We identified the following requirements for a backward-slice extractor for SlicK:

1. **Adequate Bandwidth.** The leading thread in SlicK executes in the traditional SRT-like manner. In order to not adversely affect performance, it is critical for the slice extractor to be able to support a bandwidth equal to the average commit bandwidth of the leading thread, thereby enabling a smooth flow of instructions through the pipeline without interruptions.

2. **Low Latency.** Inordinate delays in trailing thread slice extraction would lead to increased pressure on leading thread buffers. By definition, backward slice instructions are older than their corresponding triggers in program order, and therefore any slice extractor would have no way of immediately identifying an instruction the moment it arrives. It has to buffer it for a

certain amount of time before its "status" can be determined. This inherent latency is an unavoidable property of the instruction sequence itself.

We do however wish to minimize the *processing latency* of the slice extractor. When a trigger instruction arrives at the extractor, all buffered instructions whose status can be determined by this trigger must be identified by the extractor in as little time as possible.

3. **Support for Early Deletion.** Certain situations may require the oldest buffered instructions in the extractor to be forcibly deleted (popped), regardless of the fact that their status was unknown. This could happen if, for example, the leading thread has stalled due to resource shortage and the trailing thread is required to proceed forcibly in order to clear the stall. In such situations, the slice extractor needs to be able to pop the oldest instruction(s) with minimal latency and update its internal state to maintain correct dependencies.

4.3.3.2 Inadequacy of Traditional Mechanisms

A straightforward technique for backward-slice extraction is to buffer the sequence of instructions until a trigger arrives, and then perform a reverse program-order traversal of all buffered instructions in order to determine which of them lie on the backward slice of the trigger. Many traditional hardware and software slice extraction techniques have used variations of this approach [12, 29].

Unfortunately, this approach works well only if triggers are infrequent, slice extraction is required on rare occasions, and a certain amount of processing latency is tolerable. In our case, triggers are quite frequent (potentially every store and branch), slices need to be identified constantly without interrupting the instruction flow, and processing latencies could severely affect performance.

It is possible to maintain a smooth instruction flow by using additional storage to buffer incoming instructions while another set is undergoing traversal [29], but our attempts to build a slice extractor for SlicK with this technique resulted in a complex design that also turned out to be sub-optimal. Despite the additional buffering, instructions can only be processed in non-overlapping batches (or "windows") at a time, and dependencies across these windows are lost. To solve this,

instructions could be "spilled" from one window into the next, but this makes the design even more complex to implement. Further, processing latencies remain high with this approach.

4.3.3.3 Our Solution – the SliceEM

We make the following key observations in order to come up with a simple and efficient backward-slice extraction mechanism for SlicK.

- In the presence of multiple triggers, we do not need to associate each instruction individually with a trigger. Instead, we only need to know if an instruction lies on the backward slice of one or more ETs or FTs, or both.
- Complete slices need not be identified. It is acceptable for an old instruction to exit the slice extractor's buffers with an "unknown" status, as long as dependencies are correctly maintained.
- The number of distinct dependency identifiers (i.e., architected register identifiers) in the decoded instruction sequence is finite and typically small (in the range of 8-32 for most ISAs).

Our solution, the Slice Extraction Matrix (SliceEM), borrows from several previous works on backward slicing [29], forward slicing [42], and matrix based instruction scheduling logic [3, 28]. The SliceEM constantly updates and keeps track of all dependency chains that exist between the instructions currently residing within its buffer, and it employs a simple set of structures to achieve this. This makes it possible to *instantly* identify all the instructions (among those that are currently buffered) lying on the backward slice of a trigger that has just arrived. No reverse program order traversal is necessary, and all our design goals are satisfied.

We call the window of buffered instructions that the SliceEM is operating on at any point in time as the Slice Analysis Window (SAW). A larger window results in a higher probability that the oldest entries in the window have their status determined. As our later experiments will show, a window size of 256 is able to determine the status of a majority of the instructions. The SAW does not need to store the instructions themselves. It is simply a mask that indicates which

instructions need to be executed/dropped and which are still with an unknown status.

The SliceEM is composed of three main structures (Figure 4.1) that are all initialized to zero:

- A *Bit Matrix* with columns corresponding to each instruction in the SAW and rows corresponding to each architecturally visible register. The Bit Matrix at any given instant indicates which registers are "live" (if there is at least one non-zero column for a given row), and which instructions in the window are on the backward-slice of that register (the corresponding columns of that row which are non-zero). For example, in Figure 4.1, instructions in columns 3, 5 and 9 lie on the backward dependency chain of register r3, implying that any future instruction using r3 as an input will require the services of instructions 3, 5 and 9 (and no other instruction) in order to execute correctly.

- A *Live Mask* with one bit corresponding to each entry in the SAW. At any instant this mask indicates that the corresponding instruction has written into at least one "live" register.

- An *Execute Mask* with one bit corresponding to each entry in the SAW. This marks the instructions in the SAW that need to be redundantly executed because they could not be verified.

4.3.3.4 SliceEM Operations

We now describe how the SliceEM handles all Insertion and Deletion operations. Instructions are always inserted and deleted (popped) in FIFO order, and a circular queue algorithm is used.

Inserting a non-trigger instruction: When a **Non-Trigger** instruction, e.g., an `add $r5 := $r2 + $r3` register-arithmetic instruction is inserted into the window, the SliceEM does the following (starting from the SliceEM status in Figure 4.1, the process of inserting this instruction is shown in Figure 4.2):

1. Allocate a new column index k ($k = 13$ in this example) for this instruction in the SAW.

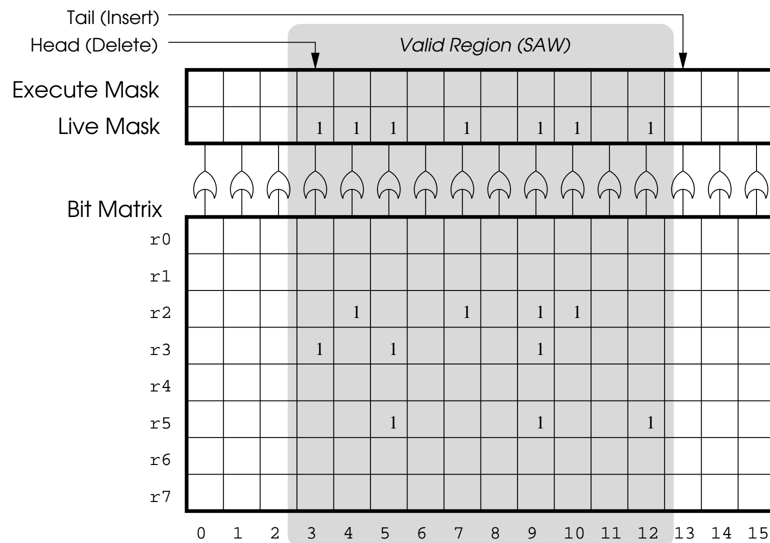


Figure 4.1. The Slice Extraction Matrix. Column indices are indicated at the bottom of the matrix.

2. Read the rows corresponding to the inputs of this instruction (rows for r2 and r3 in this example), and compute the bitwise-union of these rows.

3. Set the k-th bit of this union to 1. It is guaranteed that this bit was previously a 0, otherwise this column could not have been allocated for this new instruction.

4. Write the modified union vector into the row in the Matrix corresponding to the instruction's output register (r5 in this example), completely overwriting any values that may be already present.

5. Update the Live Mask vector by taking a wired-OR *of each of the columns.* The Live Mask of the new instruction would get set to 1. It is possible that some of the previous 1s in this vector could now get reset to 0 if this new output register row (r5 in the example) does not have a 1 in the corresponding position, and no other row has a 1 in that column either.

Inserting an Execute Trigger: When an ET instruction (for instance an effective address calculation instruction for a store `computeEA $r2 + 5`) enters the window, we do the following (starting from the Matrix in Figure 4.2, the process is shown in Figure 4.3):

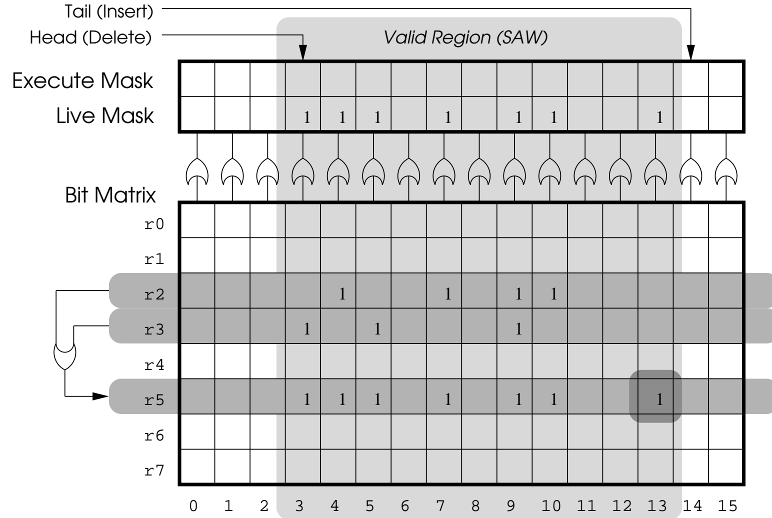


Figure 4.2. Process of inserting the instruction `add $r5 := $r2 + $r3` into the SliceEM. Notice that the instruction in column 12 ceases to be Live after this insertion. The new instruction is inserted in column 13, and a new 1 is entered into the matrix entry (`r5`, 13) to indicate this.

1. Allocate a new column index k ($k = 14$ in this example) for this instruction.
2. Read all rows corresponding to the inputs of this instruction (r2 in this example), and compute the bitwise-union of these rows. Note that this would correspond to all the instructions on the backward-slice of this ET instruction.
3. Set the k-th bit of this union to 1.
4. Take a bitwise-union of this resulting vector with the contents of the Execute Mask, and overwrite the Execute Mask with the result.

Inserting a Flush Trigger: When an FT instruction enters the window, we only need to allocate a new column for this instruction. Since we are not updating the SliceEM in any other way, any instruction in its backward-slice would automatically be *killed* when the registers whose values they produced are overwritten by any future instruction (treated exactly the same as a dynamically dead instruction). Thus our mechanism is intended to eliminate instructions in the redundant thread (i) that do not lie on the backward slice of any store or branch, and (ii) that lie on

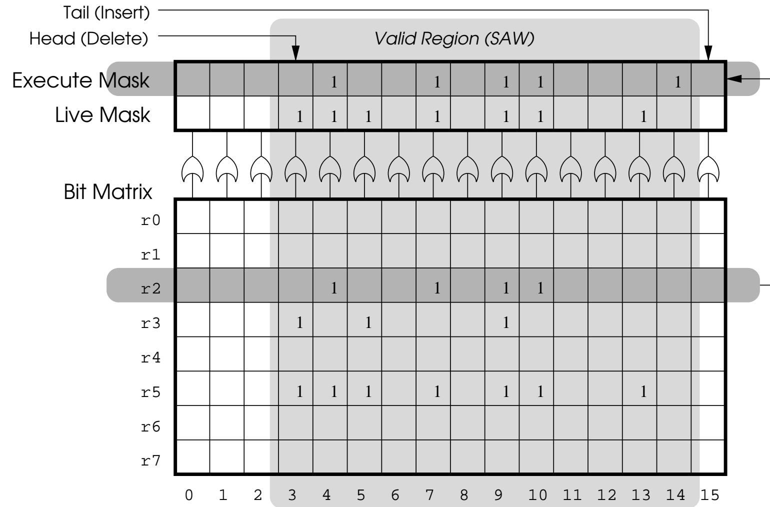


Figure 4.3. Process of inserting the ET store effective address calculation instruction `computeEA $r2 + 5` into the SliceEM. Notice the updated Execute Mask.

the backward slice(s) of only verifiable (FT) stores and/or branches. The Execute Mask and Live Mask bits are both 0 for such instructions.

Deleting (Popping) Entries: At any instant, the entry at the Head of the SAW can be examined and deleted from the SliceEM, and either sent for redundant execution or flushed. If the Execute Mask and Live Mask bits for the entry are both 0, then the instruction corresponding to this entry is guaranteed flushable. If the Execute Mask bit is 1, then the instruction requires redundant execution. If the Execute Mask bit is 0, but the Live mask bit is 1, then the instruction's status is unknown, and the logic that is removing this entry could choose to either conservatively execute the instruction redundantly, or allow it to stay in the SliceEM for a little longer. If the entry is deleted, then the Execute Mask bit, Live Mask bit and the entire column of the Bit Matrix for that instruction are reset to 0.

Observe that the SliceEM does not literally "extract" slices, but only identifies and marks instructions lying on the backward slices of certain triggers. It also does not necessarily identify complete slices. If the SAW is not wide enough to accommodate all instructions of a slice, then the oldest instructions can start spilling out with an "unknown" status. No additional logic is required to ensure that dependencies are correctly maintained in such scenarios.

The Bit Matrix and Live Mask together keep track of the "liveness" of each en-

try in the SAW. The Execute Mask identifies a set of instructions on the backward slice of a certain *class* of triggers. SlicK has only one such class: Execute Triggers. However, the SliceEM can also be used in scenarios where slices belonging to multiple classes of triggers need to be identified simultaneously. In such cases, a distinct Execute Mask needs to be used for each class, but the Bit Matrix and Live Mask can be shared.

4.3.3.5 Hardware Implementation Concerns

The key component of the SliceEM is the bit-matrix, and it primarily needs to perform 3 operations: (i) Read and write a given row, (ii) Compute the wired-OR of all bits in a column, and (iii) Reset all bits in a column. Since the number of rows (equal to the number of architected registers) is expected to be in the range of 8-32, we do not expect wire delay problems associated with operations (ii) and (iii). Further, as we will show later in our results, window sizes (columns) of 256 instructions suffice, and operation (i) is not expected to have significant overheads either. Its read/write ports need to be enough to support the commit bandwidth of leading thread instructions, which is typically less than 3 per cycle.

4.3.4 Integrating the Predictors and SliceEM into an SRT processor

Armed with a set of predictors and a slice extraction mechanism, we now need to integrate these into an SRT pipeline and complete the SlicK design. We were faced with multiple design choices with various aspects of this process. In the interest of brevity, we refrain from analyzing the pros and cons of each of these choices. Instead, we present the final design incorporating our design decisions along with brief explanations of the rationale behind these decisions.

Most of the core pipeline remains unchanged from the SRT design (Figure 4.4). As in SRT, the two PCs of a two-context SMT processor are provided for the redundant threads (leading and trailing), each independently fetching their instruction stream from the L1-cache into the SoR with a temporal slack separating the two. The trailing thread uses the Branch Outcome Queue (BOQ) for obtaining branch targets into which the leading thread would have previously placed the

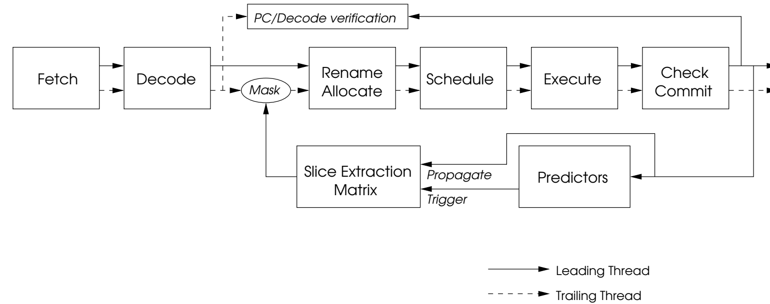


Figure 4.4. Overview of SlicK Pipeline

results of resolved branches.

Our first design decision for SlicK was to *make the leading thread send its committed instructions to look up the predictors and update the SliceEM.* The leading thread's runahead slack gives the SliceEM ample opportunity to discover the status of older instructions in the SAW by the time the trailing thread catches up. "Inserting" a committed leading thread instruction into the SliceEM simply involves updating the Bit Matrix and the two masks, and as far as slice identification is concerned, no additional information about the instruction need be maintained. All that the trailing thread needs to do is examine the Execute and Live mask bits in order to determine which instructions it needs to execute, and this can be done immediately after its Fetch stage.

There is a redundancy issue here that needs to be handled. Since it is the leading thread that looks up the predictors and constructs the execution masks to select the trailing thread's instructions, these operations themselves need to be verified by the trailing thread. This can be achieved by verifying all fetched PCs and the decoded architected register identifiers. Therefore, *we require every instruction in the trailing thread to go through the Fetch and Decode stages* (our second design decision), regardless of whether they are ultimately executed or flushed. Techniques such as fingerprinting [47] can be used to significantly reduce the storage and bandwidth overheads associated with buffering this data from the leading thread until the trailing thread catches up and performs the comparison.

After Decode, trailing thread instructions can look up the SliceEM masks and decide to either execute or flush themselves. Since the SliceEM supports early removal, the trailing thread can choose to delete an entry even if its status was unknown and execute it conservatively. This brings us to our third design decision:

if the trailing thread has decoded instructions available, then it always pops/deletes SliceEM entries, regardless of whether their status were known or unknown. In our experience, ensuring the smooth flow of the pipeline is a more critical factor for performance than removing a few additional instructions by stalling the thread for several cycles.

4.3.4.1 Memory Ordering for the Trailing Thread

Trailing thread Loads always pick up their operands from the LVQ, and trailing thread Stores have only one purpose: to provide redundancy for leading thread Stores waiting in the SCB. Therefore, as long as the LVQ and SCB semantics are correct, no additional ordering needs to be enforced between trailing thread Loads and Stores. We ensure that all trailing thread Loads and Stores perform the necessary synchronization with their associated structures (LVQ/SCB) regardless of whether they are executed or flushed.

We adhere to the SRT mechanisms for handling uncached operations and precise interrupt delivery.

4.4 Fault Coverage

4.4.1 Identifying Points of Vulnerability in SlicK

The speculative nature of the predictors in SlicK can cause certain rare redundancy violations. This is best illustrated through an example (Figure 4.5).

In this example, a store instruction, having produced the same data value 0x4000 for several iterations, warms up the data predictor. Consequently, the trailing thread does not execute the backward slice for this store data. However, the next iteration for some reason now produces the value 0x4100. In the error-free execution (Case (a)), this causes a mismatch with the predictor, which causes the redundant slice to be extracted and executed. In Case (b), the leading thread is corrupted by an error, but the error is detected. However, it is possible for a single-bit error to corrupt the leading thread data in such a way that it exactly matches with the predictor's output and result in SDC (Case (c)). From this example, we can observe the following key facts:

For 100 iterations		101st iteration	
Lead. Thr.:	`store 0x4000`	Lead. Thr.:	`store 0x4100`
Predictor:	`store 0x4000` (match – commit)	Predictor:	`store 0x4000` (mismatch – trigger Trail. Thr.)
		Trail. Thr.:	`store 0x4100` (no error)

(a)

For 100 iterations		101st iteration	
Lead. Thr.:	`store 0x4000`	Erroneous Lead. Thr.:	`store 0x4101`
Predictor:	`store 0x4000` (match – commit)	Predictor:	`store 0x4000` (mismatch – trigger Trail. Thr.)
		Trail. Thr.:	`store 0x4100` (error detected)

(b)

For 100 iterations		101st iteration	
Lead. Thr.:	`store 0x4000`	Erroneous Lead. Thr.:	`store 0x4000`
Predictor:	`store 0x4000` (match – commit)	Predictor:	`store 0x4000` (match – commit) **SDC**

(c)

Figure 4.5. SlicK Vulnerability Example, showing the execution of a Store instruction through 101 iterations of a loop in which the data value is 0x4000 for the first 100 iterations and 0x4100 in the final iteration. This is shown for (a) the Correct Execution, (b) one possible Erroneous Execution (caught by SlicK), and (c) another possible Erroneous Execution (not caught by SlicK).

- The scenario we are observing involves a Mismatch between the leading thread and the predictor in the error-free execution.
- Both Cases (b) and (c) involve a fault in the *leading thread*.
- SDC only occurs in Case (c) – if the Mismatch in the error-free case is transformed into a Hit (or Match) in the erroneous case.
- Only certain patterns of faults in the leading thread could result in Case (c); most faults will result in Case (b)-like situations.

In general, with a single-error model and with full Fetch and Decode redundancy as described in the previous section, any errors in either the predictor or SliceEM structures or in pipeline structures being occupied by trailing thread instructions cannot cause SDC, since any such errors in this model automatically implies that there are no errors in the leading thread's execution.

On the other hand, a fault in the pipeline structures occupied by *leading thread* instructions in SlicK can lead to SDC. As the example illustrated, an error in a

leading thread Store instruction (or its backward slice) that causes a predictor Mismatch in the error-free execution could potentially transform the Mismatch into a Hit and result in SDC.

We refer to these stores and their backward-slices (which lead to a Mismatch in the prediction structures) as being *unguarded*. Any arbitrary fault in an unguarded instruction would not necessarily cause such a Mismatch-to-Hit conversion; the fault has to be such that the resulting value of the Store exactly aliases with the value that the Predictor produces. However, this is tricky to quantify due to architectural and program level masking effects [25, 31], and we conservatively assume that any fault in an unguarded instruction would cause SDC. In terms of the example, we assume that any single-bit error in the leading thread *will* convert all Case (a) scenarios into Case (c) scenarios.

Note that instructions could be in more than one backward-slice, with different prediction outcomes for each slice. In our analysis, we count an instruction as being unguarded if the earliest trigger (in program order) of its forward slice is a Mismatch.

Identifying unguarded instructions requires analyzing register dependencies over a very large instruction window since the forward-slice trigger can occur millions of instructions later. However, our simulations were excessively slow with very large windows. We found that using a moderate window size of 16K instructions provided reasonable simulation speeds while still keeping the number of instructions with unknown status sufficiently small. In our results, we show coverage as ranges assuming worst (all unguarded) and best (all guarded) cases for these unknown instructions.

4.4.2 The Common Case – How SlicK detects soft errors

SlicK's predictor lookups can have three outcomes: No-Prediction, Hit and Mismatch. Of these, only Mismatches create vulnerability. Therefore, the predictors are configured so as to minimize the occurrence of Mismatches, with No-Predictions and Hits forming the majority of cases. No-Predictions result in full redundant execution as in SRT. Hits cause instructions to be flushed, gaining performance, while redundancy is maintained by the predictor output. In case an error corrupts

a leading thread trigger value that would have been a Hit in the error-free case, it is guaranteed that it will be a Mismatch in the actual erroneous execution, triggering redundant execution.

4.4.3 Quantifying the Fault Coverage of SlicK

We use the Architectural Vulnerability Factor (AVF) [31] approach to quantify the error coverage of our design. Calculating AVF requires identifying bits in a pipeline structure that are necessary for Architecturally Correct Execution (ACE). The ACE-ness of every physical bit changes from cycle to cycle, and is deduced from the ACE-ness of the logical entity (such as some component of an instruction) that was holding that bit in that cycle. Several factors contribute to making an instruction ACE or un-ACE, all of which are outlined in [31]. Once the ACE-ness of all bits in a structure are established on a cycle-by-cycle basis, the AVF of the structure is defined as the average fraction of ACE bits in the structure over the entire execution of a suite of benchmarks. The "unguarded" instructions defined above are ACE, and we consequently only need to track the bits of these instructions to quantify the AVF. Note that we are automatically discounting wrong-path instructions and dynamically dead instructions, since they will not be unguarded. However, we do not track dynamically dead instructions via memory dependencies, which again makes our estimation conservative. We perform AVF analysis for several non-speculative processor structures that contribute significantly to the total chip area, namely, the ROB and Physical Register File (which are coupled together into a unified RUU in our model), the Load Store Queue, and the Issue Queue.

4.5 Results

Our experiments were conducted via execution-driven simulation using an extended version of the SimpleScalar 3.0 toolset [4], where we implemented the SRT and SlicK models. We evaluate the schemes using all 26 applications from the SPEC2000 benchmark suite. The benchmarks were compiled for the Alpha ISA and use the reference input set. We measured the statistics for detailed simulation of 100 million instructions after fast-forwarding to the first SimPoint [44] with a

weight of at least 1% for each benchmark. The parameters of our baseline hardware model are shown in Table 4.1.

Parameter	Value
Fetch/Decode/Issue/Commit Width	8
Pipeline Stages	15
Fetch Queue Size	16
Load Value Queue (LVQ) Size	128
Branch Outcome Queue (BOQ) Size	128
Store Checking Buffer (SCB)	64
Branch-Predictor	Combined predictor with 16K-entry meta-table. 2-lev predictor with 16K-entry L1, 16K-entry L2, 14-bit history XORed with address
RAS Size	64
BTB Size	2K-entry 4-way
RUU Size	128
LSQ Size	64
Integer ALUs	6 (1-cycle latency)
Integer Multipliers/Dividers	2 (3,20)
FP ALUs	4 (2)
FP Mult./Div./Sqrt.	2 (4,12,24)
L1 D-Cache Ports	4
L1 D-Cache	64KB, 4-way with 32B block (2)
L1 I-Cache	64KB, 4-way with 32B block (2)
L2 Unified Cache	512 KB, 4-way with 64B line-size (12)
I-TLB	512-entries 4-way set-associative
D-TLB	1K-entries 4-way set-associative
TLB Miss-Latency	30 cycles
Memory Latency	200 cycles
SliceEM Columns	256
Store Address and Data Predictors	– Last-Value, 1024-entry – FCM, 1024/8192-entry L1/L2, Order 3
Branch Conf. Estimator	Gshare, 14-bit Global BHR, 16K L2

Table 4.1. Simulation parameters. Latencies of ALUs/caches are given in parentheses. All ALU operations are pipelined except division and square-root.

Table 4.2 gives the IPCs for single-thread and SRT execution. IPC losses due to SRT range from as low as 3.67% (in *mcf*) to as high as 39.73% (in *sixtrack*), with an average loss of 17.24% across these 26 benchmarks. In order to estimate an optimal size for the slice analysis window (number of columns in the SliceEM) for SlicK, we ran simulations with relatively large window sizes to find out how many younger instructions need to appear in the dynamic instruction sequence before the status (whether to execute or not) of each instruction is known. We then plotted this information as a CDF graph, and in Figure 4.6 we show representative results for a few applications. For most benchmarks, the status of 90-95% of the instructions is known within a window size of 256 instructions. Results for most other applications are similar except for *wupwise* and to a certain extent *galgel* where the status of around 75-80% of the instructions is known within a window

Benchmark	Single-Thread IPC	SRT IPC	% IPC Loss
164.gzip	1.9614	1.7924	8.62
168.wupwise	0.6664	0.4818	27.70
171.swim	0.8704	0.6046	30.54
172.mgrid	1.1261	1.0245	9.02
173.applu	1.1093	0.8398	24.29
175.vpr	0.5936	0.5578	6.03
176.gcc	1.5293	1.4369	6.04
177.mesa	2.5172	2.0216	19.69
178.galgel	2.4665	1.8210	26.17
179.art	0.6030	0.4039	33.02
181.mcf	0.1715	0.1652	3.67
183.equake	0.6657	0.5628	15.46
186.crafty	1.9653	1.7603	10.43
187.facerec	2.6073	1.7662	32.26
188.ammp	0.8191	0.6903	15.72
189.lucas	0.7389	0.6982	5.51
191.fma3d	2.3089	2.0712	10.29
197.parser	1.0874	0.9984	8.18
200.sixtrack	3.6739	2.2143	39.73
252.eon	2.3617	1.9697	16.60
253.perlbmk	1.5293	1.4704	3.85
254.gap	0.2606	0.2257	13.39
255.vortex	2.7971	1.9827	29.11
256.bzip2	1.8440	1.5976	13.36
300.twolf	0.7790	0.7479	3.99
301.apsi	3.2301	2.0835	35.50

Table 4.2. IPC of single-thread (non-redundant) execution of the benchmarks and redundant execution using SRT. The loss of IPC with SRT is also given, with the average loss across the applications being 17.24%. All the data is for the execution of 100 million instructions from the first SimPoint with a weightage at least 1% for each benchmark.

size of 256. We consequently fix the number of columns in the SliceEM at 256.

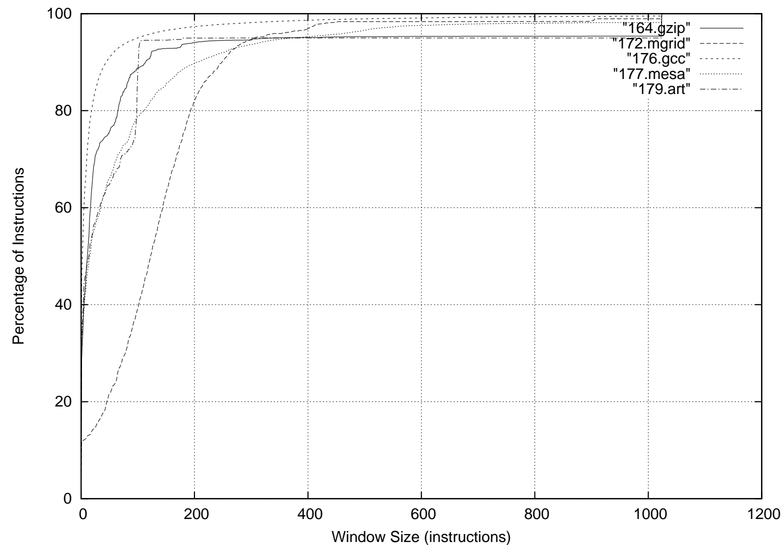


Figure 4.6. Percentage of instructions whose status (whether to execute or not) becomes known after x instructions in the dynamic instruction stream.

4.5.1 Performance Results

Figure 4.7 gives the IPC results (normalized with respect to single thread performance) of our SlicK mechanism using both Last-Value (LV) and Finite-Context Method (FCM) store predictors, and compares them with the normalized IPCs of the SRT execution. FCM provides only marginal improvements over LV but at a much higher hardware cost, and we consequently focus on the LV store predictors for the rest of this section. Table 4.3 shows the actual IPC values and some other statistics about the SlicK execution, including the percentage of instructions that are flushed, number of branch FTs (and as a percentage of number of branches), and the number of store address/value FTs (and as a percentage of the number of stores).

	IPC	**Flushed**	**Branch FT**	**St. Addr FT**	**St. Data FT**
gzip	1.8855	34.48%	7.64M (81%)	3.17M (45%)	0.24M (3%)
wupwise	0.5957	9.32%	3.25M (99%)	0.00M (0%)	0.00M (0%)
swim	0.8167	12.27%	1.16M (99%)	0.75M (6%)	0.00M (0%)
mgrid	1.0485	5.49%	0.27M (96%)	2.85M (59%)	0.02M (0%)
applu	1.0154	12.51%	0.20M (97%)	2.90M (27%)	3.71M (35%)
vpr	0.5850	33.67%	9.10M (88%)	4.71M (43%)	3.87M (35%)
gcc	1.4696	36.25%	4.78M (30%)	2.00M (19%)	3.53M (34%)
mesa	2.1834	43.35%	7.11M (83%)	7.69M (63%)	6.54M (54%)
galgel	2.1094	28.26%	4.32M (93%)	2.51M (50%)	0.01M (0%)
art	0.5765	33.59%	9.08M (81%)	4.28M (55%)	0.76M (9%)
mcf	0.1712	35.07%	14.26M (61%)	0.06M (1%)	2.68M (40%)
equake	0.6246	24.97%	3.11M (97%)	2.14M (51%)	2.97M (71%)
crafty	1.8712	38.44%	5.48M (49%)	2.33M (40%)	1.88M (32%)
facerec	2.0351	12.97%	3.12M (89%)	0.85M (6%)	0.52M (3%)
ammp	0.7505	11.44%	3.50M (90%)	2.47M (37%)	0.31M (4%)
lucas	0.7133	1.00%	0.53M (99%)	0.00M (0%)	0.00M (0%)
fma3d	2.2679	52.46%	14.82M (82%)	2.15M (28%)	3.74M (50%)
parser	1.0110	35.64%	9.58M (61%)	4.06M (50%)	3.00M (37%)
sixtrack	2.3898	10.59%	2.21M (98%)	1.19M (22%)	0.00M (0%)
eon	2.1179	40.70%	9.16M (83%)	8.36M (49%)	7.70M (45%)
perlbmk	1.5021	43.09%	10.01M (71%)	5.25M (33%)	6.03M (38%)
gap	0.2282	19.51%	9.04M (69%)	4.71M (40%)	2.56M (21%)
vortex	2.3286	57.57%	15.56M (87%)	3.09M (24%)	6.86M (55%)
bzip2	1.7241	45.46%	9.39M (84%)	3.92M (48%)	2.47M (30%)
twolf	0.7778	28.35%	7.27M (59%)	2.50M (35%)	1.54M (21%)
apsi	2.2848	11.94%	3.00M (91%)	1.87M (14%)	0.65M (4%)

Table 4.3. Execution Statistics for SlicK with LV predictor. We show the percentage of flushed/eliminated instructions, the number (and as a percentage of total branches) of Branch FTs, the number (and as a percentage of total stores) of Store FTs.

From Figure 4.7, we see that SlicK can provide a significant bridge for the performance gap between SRT and Single Thread executions. For instance, in *vortex*, *mesa* and *eon*, where SRT incurs about 29%, 20% and 17% IPC loss respectively, SlicK is able to cut this loss by 42.47%, 32.65%, and 37.81% respectively by elim-

Figure 4.7. IPC results for SlicK executions with a simple Last Value (LV) and a more extensive Finite-Context Method (FCM) store address/value predictor, shown in comparison with SRT. All values are normalized with respect to Single Thread (non-redundant) execution.

inating a substantial number of instructions (57%, 43% and 41%) in the trailing thread.

For most applications, we find an expected correlation between the percentage of instructions flushed to the improvements in IPC for the SlicK executions. As

noted above, *vortex*, *mesa* and *eon* are representative of those with significant improvements due to removal of instructions based on locality. At the other end, *sixtrack*, despite the high IPC loss (40%) due to SRT, SlicK can only provide 8% improvement due to the low number of flushed instructions. This reduction in flushed instructions can be attributed to the lower locality of both store addresses and data for this application – even though branch prediction accuracy rates are high (98.46%), the number of branches themselves (2.2M) are not as high as the number of Store ETs (9.5M).

mgrid experiences good locality (visible from the high FTs for this application), but the number of flushed instructions for this application is not as high as for others. We found that this was because of our in-order extraction of entries from the SAW, wherein we send trailing thread instructions corresponding to these entries into the pipeline if these instructions have arrived and pipeline resources are available, even if their status is unknown in the SAW. Although our maximum window size is 256, the SAW rarely fills up to its full capacity. For most applications, this is not a problem, but *mgrid* can discover the status of a very small number of instructions with a smaller window of information (16 or lower) as is evident from its low initial slope in Figure 4.6.

There are cases (as in *parser*) where the SRT performance loss is itself not very high (around 8%) since the datapath resource contention between the threads is not very significant. In such cases, the effect of eliminating 35% of the instructions for redundant execution by SlicK does not produce substantial savings over SRT performance (only 1.26%). SlicK seems to perform well even for memory-bound applications such as *mcf, art* and *swim* compared to SRT, but we are slightly less interested in them since these are very low IPC applications to begin with, and SlicK's focus is on alleviating datapath resource contention for the trailing thread.

Overall, SlicK provides around 10.6% performance improvement over SRT, and buys back over 50% of the performance loss incurred by SRT with respect to Single Thread execution.

	RUU	Issue Queue	LSQ
164.gzip	0.19% – 5.66%	0.08% – 2.16%	0.07% – 2.40%
168.wupwise	0.01% – 27.23%	0.01% – 26.81%	0.01% – 14.74%
171.swim	0.00% – 0.70%	0.00% – 0.68%	0.00% – 0.00%
172.mgrid	0.00% – 0.00%	0.00% – 0.00%	0.00% – 0.00%
173.applu	0.00% – 0.38%	0.00% – 0.37%	0.00% – 0.00%
175.vpr	0.48% – 0.49%	0.37% – 0.38%	0.71% – 0.71%
176.gcc	0.21% – 0.21%	0.11% – 0.11%	0.18% – 0.18%
177.mesa	0.14% – 1.28%	0.08% – 1.19%	0.09% – 0.10%
178.galgel	0.14% – 0.17%	0.13% – 0.15%	0.01% – 0.02%
179.art	0.35% – 2.56%	0.35% – 1.65%	0.00% – 0.16%
181.mcf	0.18% – 0.44%	0.12% – 0.39%	0.16% – 0.16%
183.equake	0.16% – 0.27%	0.11% – 0.22%	0.13% – 0.13%
186.crafty	0.41% – 0.41%	0.24% – 0.24%	0.32% – 0.32%
187.facerec	0.04% – 1.53%	0.03% – 1.32%	0.01% – 0.88%
188.ammp	0.45% – 1.96%	0.21% – 0.75%	0.22% – 2.10%
189.lucas	0.00% – 0.05%	0.00% – 0.04%	0.00% – 0.00%
191.fma3d	0.27% – 0.27%	0.14% – 0.14%	0.14% – 0.14%
197.parser	0.40% – 0.40%	0.16% – 0.17%	0.18% – 0.18%
200.sixtrack	0.00% – 0.04%	0.00% – 0.04%	0.00% – 0.00%
252.eon	0.59% – 0.60%	0.42% – 0.42%	0.64% – 0.64%
253.perlbmk	0.24% – 0.24%	0.12% – 0.12%	0.22% – 0.22%
254.gap	0.16% – 1.03%	0.03% – 0.89%	0.10% – 0.10%
255.vortex	0.20% – 1.60%	0.15% – 1.28%	0.21% – 0.22%
256.bzip2	0.20% – 0.98%	0.13% – 0.86%	0.11% – 0.11%
300.twolf	0.94% – 0.94%	0.47% – 0.47%	1.08% – 1.08%
301.apsi	0.02% – 0.02%	0.02% – 0.02%	0.00% – 0.00%

Table 4.4. Architectural Vulnerability Factors for important structures in a SlicK processor with LV store predictors. Vulnerability comes from unguarded instructions due to predictor Mismatches (0.58% on the average), while performance comes from flushed instructions due to predictor Hits (46% on the average).

4.5.2 Fault Coverage Results

Table 4.4 gives our measured AVF numbers for important structures in a SlicK processor with LV store predictors. As mentioned previously, we provide AVF ranges instead of single values due to simulation time limitations. For most benchmarks these ranges are fairly small, except for *wupwise*, which has a very large number of instructions whose status cannot not be determined within the finite window sizes that our simulators can handle.

Barring *wupwise*, we notice from Table 4.4 that the AVF numbers are very low when compared against the structural AVFs of either a processor that is running in non-redundant mode [31], or one that is employing a more performance-oriented redundancy mechanism [17]. With SlicK, maximum RUU (where instructions usually have the highest residency), AVFs are less than 1% for 18 of the 25 benchmarks, between 1% and 2% for 5 benchmarks, and greater than 2% for only 2 benchmarks. Minimum RUU AVFs are less than 1% for all 26 benchmarks (including *wupwise*),

and less than 0.5% for 24 of them. Issue Queue and LSQ numbers are similarly low. These low AVFs are the result of very few unguarded instructions (0.61% on the average). Unguarded instructions are low due to the extremely low rate of Mismatches (0.58%).

To understand the impact of selectively dropping slices based on locality on vulnerability, we present an anecdotal analysis comparing locality-based slice removal versus an approach that removes slices randomly. We show results from experiments where we use the SliceEM to (a) drop only dynamically dead instructions (Dead), (b) randomly drop slices for 20% of the triggers (Random-20), and (c) randomly drop slices for 30% of the triggers (Random-30). Figure 4.8 shows the normalized IPC and maximum RUU-AVF results for these three experiments and compares them with SRT and SlicK (locality-based removal) results. For these experiments, we focus on a small subset of high-IPC benchmarks that are primarily datapath-limited, and suffer high performance degradation due to SRT.

At one end, baseline SRT provides full coverage (0% AVF due to full dual modular redundancy) but suffers a relatively high performance impact (65-90%). Starting from SRT, dropping only dynamically dead instructions provides a moderate improvement in performance, while maintaining 0% AVF. From here, if we start randomly selecting slices to not execute redundantly, while the performance does improve, vulnerability increases rapidly as the fraction of dropped instructions increases. As we reach 30% dropped instructions (which is approximately the same as the SlicK average drop rate), we achieve IPCs between 74-94% of single thread execution, but with RUU AVFs that can range anywhere between 7.5% to 20%. This is where the advantages of SlicK's value locality exploitation becomes apparent. At a performance level slightly better than Random-30 (71-97% of single thread IPC), SlicK brings the AVF down to 0.02-1.59%. Value locality is a redundancy mechanism by itself, thus allowing the option of avoiding redundant execution of the corresponding slices.

4.5.3 Discussion

The previous experiment illustrates the reason why SlicK is very attractive as a *baseline* for exploring the performance-reliability tradeoff space via partial thread-

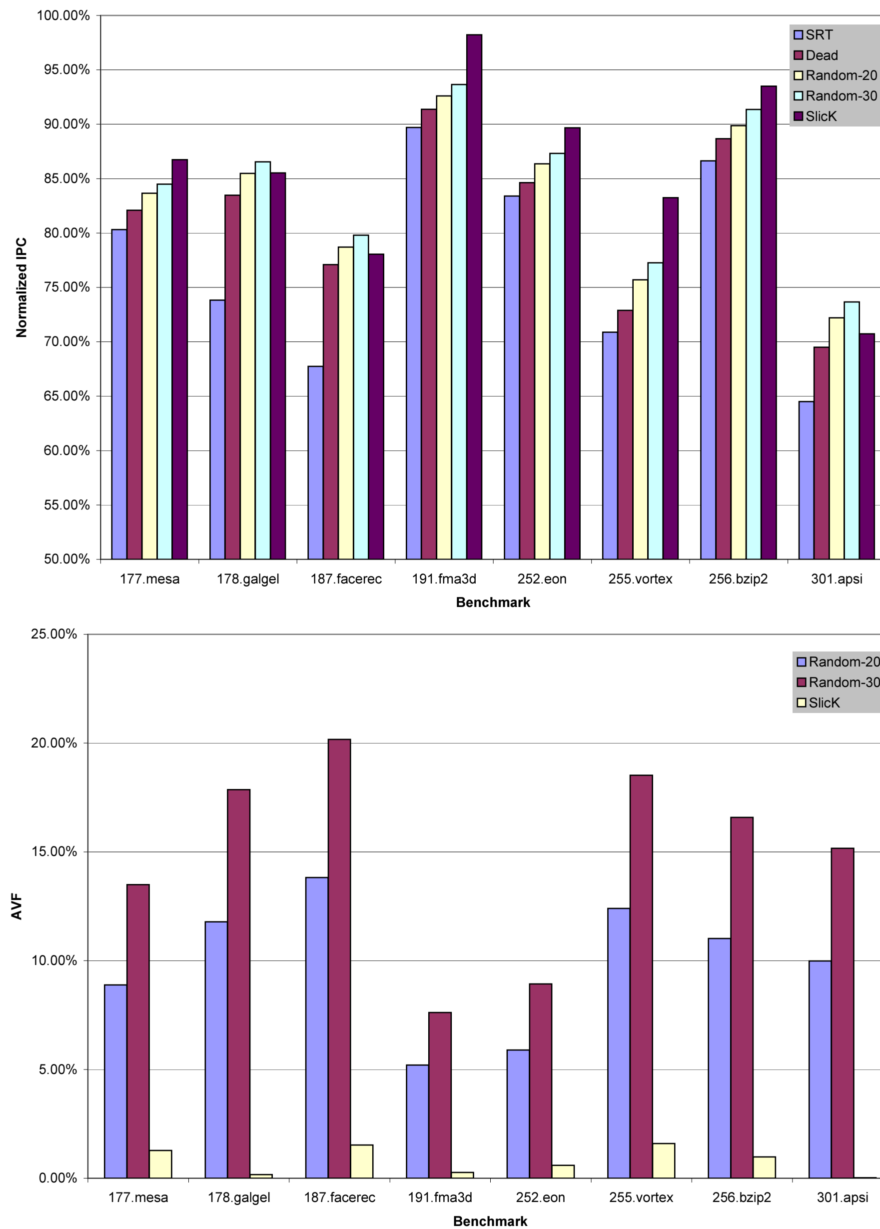


Figure 4.8. Normalized IPC and maximum RUU-AVF that were run with a subset of trigger instructions randomly chosen to be dropped together with a redundant execution that drops only dynamically dead instructions. Note that the AVFs of SRT and Dead experiments are 0.

ing.

On the one hand, designers may find SlicK's vulnerability baseline too conservative and want to improve on performance. Slice-based execution simplifies the decision making (of what instructions to execute and what to drop) by allowing

heuristics to focus directly on entities that exit the processor's Sphere of Replication and affect the rest of the system. For instance, one heuristic could be to avoid redundant execution of register-spill ET Stores (FTs are always dropped), indicating that the value would probably undergo several levels of masking before it propagates to an I/O device or another processor. Another heuristic could be to increase the threshold of dropping if the processor resources are being overly stressed, and to lower it if there is spare execution bandwidth available (similar to [17]). The slice-based approach avoids any complications related to copying state from one thread to another.

On the other hand, mission-critical systems may desire even more reliability than that afforded by SlicK's baseline. In this case, a certain subset of FT slices could be redundantly executed (recall that in a real world scenario, an error can convert an ET-mismatch into an FT). Once again, different heuristics could be used to select these.

Our framework can also allow a compiler to mark critical stores for redundant execution to over-ride the predictor outcomes, or mark them for avoiding redundant execution.

4.6 Concluding Remarks

In this chapter, we proposed the idea of using slice-level redundant execution of instructions based on the value and control-flow locality in the program. Redundant execution at the granularity of slices is a novel paradigm that enables partial threading policies to focus directly on entities that exit the processor and affect the outside world. We have demonstrated the effectiveness of using control-flow and value locality as a slice-selection policy. With predictors being used to verify the execution of the leading thread, redundant execution of a large fraction of instructions can be avoided without significantly compromising on the vulnerability of processor structures. With a detailed cycle level model of our SlicK architecture, we show that SlicK outperforms SRT by 10.6%, buying back over 50% of the performance loss incurred by SRT due to redundant execution. At the same time, elimination of redundant execution of a significant fraction of the slices has only marginal effect on the AVF of processor structures, keeping them typically in the

0%-2% range.

The backward-slice extractor proposed in this chapter is is able to track the instruction lying on slices of several trigger instructions simultaneously and does not need to stall the instruction flow while tracking is occurring. It is simple to implement, and though we have adapted it for redundant threading, there are several other applications for backward slice extraction.

Chapter 5

Bounding Vulnerabilities of Processor Structures

5.1 Introduction

In this thesis thus far, we have explored certain architecture-level techniques aimed at achieving transient fault tolerance for processor cores. Architecture-level approaches are attractive because of the flexibility they offer to explore a range of alternatives in the cost vs. performance vs. reliability design space. However, high-level architecture design lies at an early stage in the processor design cycle, and it is difficult to determine at this stage whether an architectural fault-tolerance mechanism will necessarily satisfy the hard reliability budgets that real systems are required to meet. Therefore, architectural fault-tolerance mechanisms must either:

(a) conservatively provide full redundancy to reduce the effective architectural vulnerability of the protected structures to zero while incurring heavy performance or implementation costs, or

(b) provide flexible mechanisms with controllable *knobs*, such that when the design matures and raw error rates are known, the knobs can be adjusted to ensure that the mechanism *guarantees* to satisfy any required vulnerability bound.

The techniques presented in this thesis thus far have used the approach specified in option (a) above, and attempted to achieve perfect or near-perfect redundancy. Other proposals that have followed this approach include [40, 56, 30]. All of these techniques incur significant area and/or performance overheads as a result of redundant execution. While we have shown how value locality can be used to offer a certain degree of redundancy, the efficacy of such mechanisms is necessarily dictated by the amount of inherent value locality in the dataflow.

On the other hand, proposals such as [54, 17] provide performance close to (or sometimes better than) that of single-thread execution, but with greatly reduced fault coverage. More importantly, this reduced fault coverage is *unbounded*, and there is no guarantee that the technique will adhere to any specified reliability budget.

Instead of exploiting any specific program artifact or performance feature to enhance reliability, we examine in this chapter a system for vulnerability control of processor structures in order to compulsorily remain within specified reliability budgets [52]. Our system has fixed implementation (area) costs (which are comparable to previously proposed approaches for full/partial fault coverage), and trades off performance to achieve as much fault tolerance as is required to meet the specified vulnerability bound.

Vulnerability control can be specified and performed for any processor structure. In this work, we primarily apply our techniques to the Reorder Buffer (ROB), a structure that contributes significantly to the processor's real estate. We first present an implementable technique to monitor the vulnerability of the ROB online in hardware. Next, we present a vulnerability control (VC) mechanism which, with the assistance of the monitoring infrastructure, detects vulnerability bound violations and then reactively performs redundant execution of those instructions which could have caused these violations.

We model and simulate these techniques on a detailed, cycle-accurate processor model, and provide simulation results for all 26 SPEC CPU2000 benchmarks.

5.2 Background and Motivation

As explained in Chapter 2, vendors need to ensure that their systems meet the requirements of certain *reliability budgets* specified in terms of Mean Time To Failure (MTTF) or Faults in Time (FIT rates). System reliability budgets can be translated into individual component FIT budgets. Given a target FIT budget for a component or structure, and an estimated circuit error rate, it is possible to determine an Architectural Vulnerability Factor (AVF) bound that an architectural transient fault-tolerance solution must satisfy in order to meet the targeted system reliability budget. However, the circuit-level estimates are usually not known during the high-level architectural design stages of the processor design cycle, thereby mandating architectural solutions to be able to meet any arbitrary AVF bound that can be "dialed-in" when the final circuit estimates become available. If an architectural fault tolerance mechanism is able to adjust itself to meet any specified AVF bound, then this is a sufficient condition to guarantee that this mechanism can be used to satisfy any arbitrary FIT budget for the component, and in turn the entire system.

Temporal Variance of AVF: Figure 5.1 shows the variation of the ROB-AVF for *177.mesa* over a period of 5000 cycles, at cycle-level and 100-cycle granularities. The plot reveals that the vulnerability varies significantly, reaching near-0% levels at times, and growing as high as 95% at other instances. The *average* AVF of the benchmark is about 40%. This brings up the issue of the *time granularity* at which a certain AVF bound needs to be satisfied. Guaranteeing to meet an average AVF bound over the execution of the entire benchmark is a lot simpler than ensuring that a stringent vulnerability bound is maintained every cycle, but could result in time periods where structure vulnerabilities are extremely high, which could be unacceptable for critical systems. In this work, we design our monitoring and control mechanisms to provide vulnerability guarantees at strict cycle-level granularities, giving us worst-case performance estimates. If the bounds are relaxed, more performance could be achieved using our mechanisms.

Having established that the ability to bound the AVFs of all processor structures to any specified limit at a cycle-accurate granularity is sufficient to meet

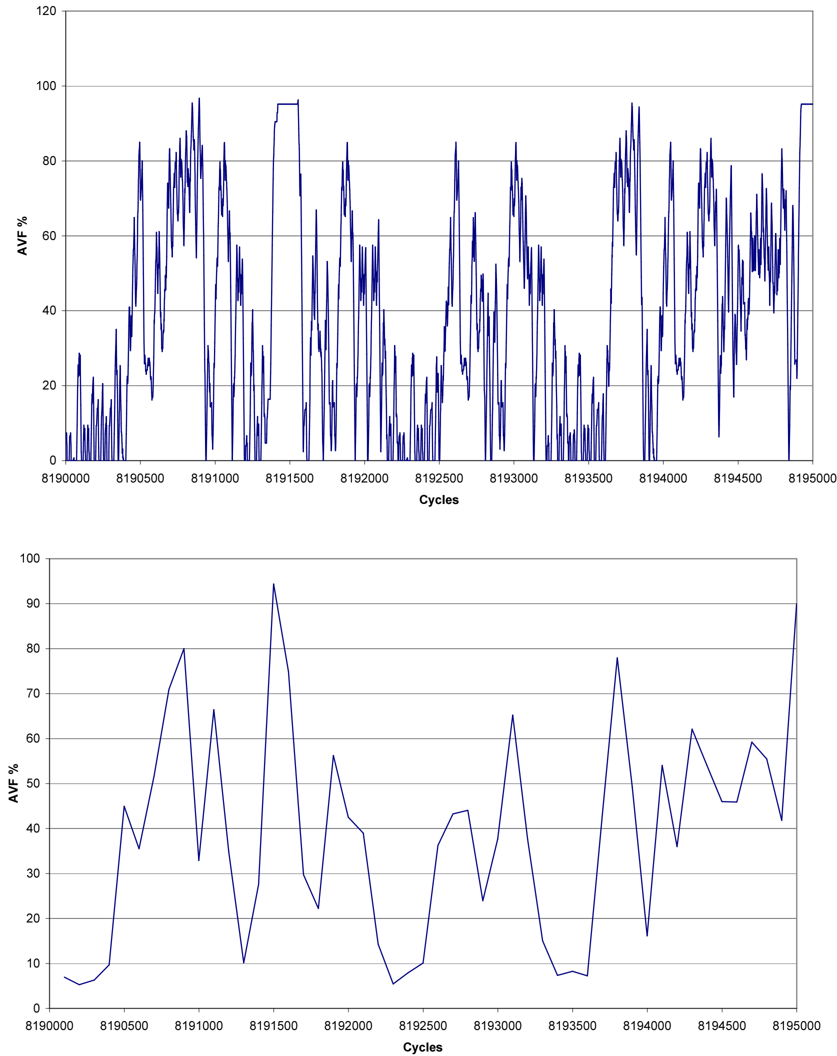


Figure 5.1. Dynamic AVF Variation in *177.mesa* at Cycle (top) and 100-Cycle (bottom) Granularity

any target reliability budget under any underlying process-technology constraints, we first present a simple online technique to dynamically monitor the AVF of the ROB. Next, we present the design and implementation details of a control mechanism to bound the AVF of the Re-Order Buffer (ROB) to any specified limit. The mechanism has fixed one-time implementation costs in terms of area and design overheads, but is flexible in its ability to accommodate any target AVF bound by dynamically trading off as much performance as is necessary to achieve the required reliability.

5.3 Dynamic Vulnerability Monitoring

The first step in building a dynamic vulnerability control system for a processor core is to design an online AVF monitoring infrastructure. This is a non-trivial task, since near-accurate AVF estimation is an exceedingly complex process, even for offline analysis. Detection of many of the conditions that cause un-ACE-ness requires dependency chain analysis across a large sequence of instructions, which is not feasible to carry out online. There have been efforts to predictively estimate AVF by observing phased behavior of applications [15], but such statistical estimates are unsuitable for our purpose since we wish to *guarantee* that the architectural vulnerability of the structure does not exceed a given bound during any time interval.

Baseline Microarchitecture: We assume a microarchitecture that uses a coupled Re-Order Buffer/Physical Register File, similar to that used in the Intel P6 [43]. Henceforth, we will use the term ROB to refer to this integrated structure. The Architected Register File is maintained as a separate structure into which instructions retiring from the ROB write their results. The Issue Queue stores the operand values (whenever available) along with tags for un-scheduled instructions.

Obtaining Upper Bounds on ACE-bit Counts: Although ACE-ness/un-ACE-ness can be caused by several factors, a significant fraction of the AVF of a structure can be accounted for by tracking the residencies of non-speculative instructions in the structure, and conservatively ignoring un-ACE-ness arising from complex dependence-based factors. Using residency periods alone, it is possible to compute an *upper bound* on the AVF of a structure. For example, if 64 of the 128 entries in a processor's ROB are un-occupied in a certain cycle, then (assuming that the Head and Tail pointers are invulnerable to faults) it is guaranteed that the AVF of ROB can be at most 50% during that cycle. Next, if it is known that the results of certain instructions have not yet arrived, then this factor can be further de-rated. Finally, the contribution of wrong-path (mis-speculated) instructions to ACE-ness can also be discounted. These are all observable phenomena that can be tracked using simple monitoring logic to provide an upper bound on the ROB AVF at a cycle-accurate granularity.

Incremental Counting: Given the number of ACE bits in the structure in any cycle, it is possible to compute the number of ACE bits in the next cycle by observing the number of dispatches, writebacks and commits into the structure in the current cycle. If the initial number of ACE bits in the structure (when the processor is initialized) is assumed to be zero, it is possible to determine the number of ACE bits in any cycle.

We use the following notation henceforth:

		Known Parameters
N	=	ROB entries
B	=	Bits in a ROB entry
R	=	Bits in result field of ROB entry
AVF_{max}	=	AVF bound (specified as a fraction)
		Tracked Quantities
d_i	=	# dispatches in cycle i
d_i^{cp}	=	# correct-path dispatches in cycle i
w_i	=	# writebacks in cycle i
c_i	=	# commits in cycle i

Apart from d_i^{cp}, which requires oracle knowledge or explicit back propagation of information, all of the other quantities can be easily monitored. We show later that VC mechanisms are sometimes unable to exploit the information available from these tracked quantities. Therefore, we illustrate three different quantities (of varying accuracy) that can be used to estimate the total number of vulnerable bits (TVB) in the structure. The estimate that is actually used depends on the requirements and characteristics of the particular VC implementation. (1) TVB^{base} is the most inaccurate count that does not account for un-ACE-ness due to unresolved mis-speculated instructions or pending writebacks, (2) TVB^{wb} accounts for un-ACE-ness due to pending writebacks, and (3) TVB^{wb+ms} takes into account un-ACE-ness due to pending writebacks and un-resolved mis-speculated instructions in the ROB. Given the TVBs for any cycle i, the TVBs for the next cycle

can be computed using the following equations:

$$\begin{aligned} TVB_{i+1}^{base} &= TVB_i^{base} + (d_i \times B) - (c_i \times B) \\ TVB_{i+1}^{wb} &= TVB_i^{wb} + (d_i \times (B - R)) + (w_i \times R) - (c_i \times B) \\ TVB_{i+1}^{wb+ms} &= TVB_i^{wb+ms} + (d_i^{cp} \times (B - R)) + (w_i \times R) - (c_i \times B) \end{aligned}$$

When a branch misprediction is detected, TVB^{base} and TVB^{wb} need to be re-constructed by subtracting the contribution of the mis-speculated instructions to the respective counts. This can be done in parallel with the pipeline flush and rename-table reconstruction. While TVB^{base} and TVB^{wb} can be counted online, tracking TVB^{wb+ms} is much more complex. Therefore, we use it only for illustrative purposes and do not use it with our VC mechanism. Note that if ACE_i is the actual number of ACE bits in the ROB in cycle i (as established by a hypothetical "perfect" analysis technique), then:

$$ACE_i \leq TVB_i^{wb+ms} \leq TVB_i^{wb} \leq TVB_i^{base} \tag{5.1}$$

Interaction with VC mechanism: To guarantee that the AVF bound is satisfied, a VC mechanism first computes maximum number of allowable ACE bits in the structure as $ACE_{max} = AVF_{max} \times N \times B$. Then, it needs to ensure that in every cycle i, $ACE_i \leq ACE_{max}$. From Equation 5.1 it should be clear that for any of the TVBs, ensuring that $TVB_i \leq ACE_{max}$ will satisfy the vulnerability bound.

5.4 Vulnerability Control via Selective Redundancy (VCSR)

Selective Redundancy is based on the observation that if an instruction is redundantly executed through the processor pipeline, then the bits in a structure through which both copies of the instruction flowed during their execution are rendered effectively invulnerable. In fact, this property is inherently exploited by all Redundant Multithreading techniques (including the approaches described in this thesis thus far) that fully replicate an execution thread to achieve fault tolerance [40, 39, 30, 34].

The goal of our VCSR approach is to satisfy a hard vulnerability bound at all times, while attempting to optimize performance within the constraints of the vulnerability bound. We demonstrate a VCSR implementation for the ROB of a modern processor, but the key principles and mechanisms that we propose should be adaptable to other structures as well.

VCSR monitors the flow of ACE bits through the ROB every cycle and attempts to guarantee that the total number of ACE bits do not exceed a fixed bound. In any cycle, if the number of ACE bits exceed the target bound, then a subset of instructions that contributed to the ACE bits in that cycle are selected and redundantly executed, thereby effectively transforming them to un-ACE bits.

5.4.1 Achieving Selective Redundancy

In order to support the execution of two redundant threads in the pipeline (although one of the threads only executes a subset of instructions), we start with a baseline Simultaneous Redundant Threading (SRT) microarchitecture. For VCSR, the leading thread is the primary execution thread and is executed in full. If the AVF bound of the ROB is violated in any cycle due to excessive number of ACE bits belonging to leading thread instructions [1], then a set of instructions are chosen for redundant execution via the trailing thread. Therefore, a mechanism is required that allows selective redundancy in the trailing thread.

The primary issue that must be dealt with in order to achieve selective redundancy for the trailing thread is maintaining correct architectural dataflow. Since a partial set of instructions are being executed in this thread, instructions whose producers were not executed require their source operands to be forwarded from the leading thread. Despite this forwarding, redundancy can be maintained if all instructions are checked for correctness before being committed.

Figure 5.2 shows a block diagram of our microarchitecture, which is similar to that employed in [17]. Committing leading thread instructions push their results into a FIFO queue which we call the Result Buffer. The purpose of the Result Buffer is to: (a) temporarily buffer outputs until the trailing thread can compare them against its own outputs for redundancy, and (b) provide source operands for

[1] In a Single Event Upset model, all bits belonging to the trailing thread are un-ACE.

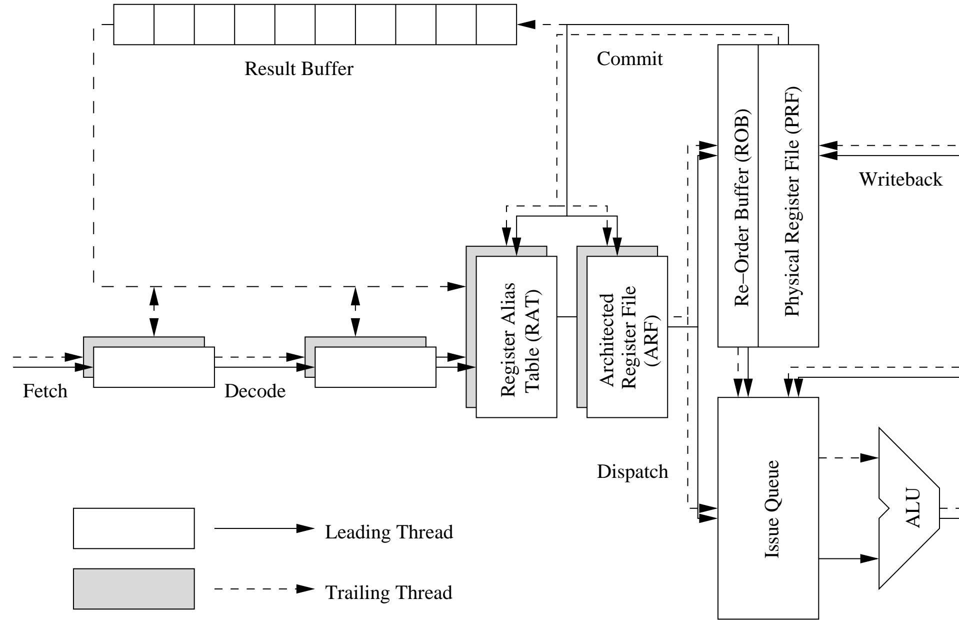


Figure 5.2. VCSR Pipeline Overview.

instructions that the trailing thread does not execute. In addition to the results, the result buffer also contains a single bit that specifies whether the instruction needs to be redundantly executed or not.

5.4.1.1 Maintaining a Consistent Architected State

The trailing thread maintains a *consistent architected state* in its Architected Register File (ARF). Recall that the trailing thread does not mis-speculate control flow due to the presence of the BOQ. In VCSR, the trailing thread fetches and pre-decodes all non-speculative instructions regardless of whether they are ultimately chosen for selective re-execution or not. The Result Buffer is examined to determine whether the instruction needs redundant execution. If it does, then the instruction is sent through the rest of the pipeline. The Result Buffer entry needs to be held until the instruction commits so that the redundancy check can be performed. If the instruction does not need to be redundantly executed, then the Result Buffer entry is used to update the trailing thread's ARF entry so that its consumer instructions can obtain the value.

Careful synchronization is needed to ensure that this register update is consistent; the actual update must be performed when this instruction would have

been renamed, although the instruction does not need to physically go through the decode and rename logic. This ensures that the RAT is consistent with the instruction flow. We also ensure that all trailing thread Loads, Branches and Stores perform the necessary synchronization with the LVQ, BOQ and SCB so that these structures remain consistent. Since the trailing thread's ARF is maintained in a consistent state, and stores to the system are synchronized with the trailing thread's commit point for the stores, this ARF can be defined as a precise state for establishing checkpoints, both for interrupt handling as well as error recovery.

5.4.2 Selecting Instructions for Redundant Execution

Given an infrastructure to dynamically monitor AVFs and a mechanism to perform selective redundancy, we now need to come up with a policy that selects a set of instructions for redundant execution such that the AVF bounds for the ROB are met every cycle. Ideally, we would like to select the set of instructions that provides the best possible performance among all possible sets that satisfy the vulnerability criteria.

While it is obvious that arriving at such an optimum solution is out of the question for a real implementation, it is also infeasible to perform any offline analysis to determine an "OPT" that can aid in evaluation of implementable heuristics. In this thesis, we provide one implementable *greedy* heuristic to satisfy the constraint, but we leave a comparative evaluation against other possible heuristics to future work.

The only two events that can increase the number of ACE bits in the ROB are Dispatch and Writeback. Since VCSR is a reactive technique, it can nullify the ACE-ness increase caused by either a dispatch or a writeback by simply tagging the instruction that caused a bound overflow for redundant execution (VCT can only control dispatches). We use TVB^{wb} as the vulnerability estimate for VCSR.

The VCSR heuristic can flag an instruction for redundant execution at either of the following two stages:

- **During Dispatch:** If it is determined that dispatching an instruction will cause a violation, i.e., if $(B - R) + TVB^{wb} > ACE_{max}$, then the instruction needs to be marked as requiring redundant execution. This flag is stored in

the ROB until commit, upon which it is transferred to the Result Buffer.

- **During Writeback:** If a writeback is causing a violation, i.e., $R + TVB^{wb} > ACE_{max}$, then the instruction needs to be flagged for redundant execution.

The above description is a simplified view of the process. In reality, several instructions could be undergoing dispatch, writeback and commit simultaneously in a single cycle, and the flagging mechanism needs to take all of these as input to determine the set of instructions to be flagged. We give priority to younger instructions for removal since they have a greater probability of being mis-speculated instructions (see discussion below).

The detection and flagging logic can be kept off the critical paths of the dispatch and writeback logic by using independent ports to the access the flag bit in the ROB, and allowing for an additional cycle to write the flag bit if necessary. This is not unusual since the different fields in a ROB are typically accessed via independent ports in real microprocessor implementations.

Note that an instruction selection logic could have potentially chosen from amongst any of the instructions currently residing in the ROB, and not necessarily from amongst the instructions currently undergoing dispatch or writeback. However, selecting in this manner simplifies the decision logic, maximizes selection of possibly mis-speculated instructions (see below), and allows the flag bit to potentially share write ports and address decoders with the status bits.

Impact of Mis-Speculation: Although TVB^{wb} does not account for un-resolved mis-speculated instructions, observe that if there are any mis-speculated instructions in the ROB, then *it is guaranteed that any instructions that are waiting for dispatch are also on the mis-speculated path*. Since mis-speculated instructions are never executed by the trailing thread, VCSR is guaranteed to never execute any dispatching instructions that were unnecessarily tagged due to the over-counting of mis-speculated instructions by TVB^{wb}.

However, it is not possible to guarantee that instructions flagged during writeback will not lead to over-conservative re-execution. To minimize this occurrence, we try to flag for re-execution the youngest among the currently writing-back instructions.

5.5 Results

Our experiments were conducted via execution-driven simulation using processor models that we implemented using the SimpleScalar 3.0 toolset [4]. We evaluated our techniques using all 26 applications from the SPEC CPU2000 benchmark suite. The benchmarks were compiled for the Alpha ISA, and reference input sets were used. We measured the statistics for detailed simulation of 100 million instructions after fast-forwarding to the single SimPoint [44] of each benchmark. The parameters of our baseline model are shown in Table 5.1.

Baseline Datapath Parameters

Parameter	Value
Fetch/Decode/Issue/Commit Width	6
Pipeline Stages	15
Fetch Queue Size	16
Load Value Queue (LVQ) Size	128
Branch Outcome Queue (BOQ) Size	128
Store Checking Buffer (SCB)	64
Branch-Predictor	Combined predictor with 16K-entry meta-table. 2-lev predictor with 16K-entry L1, 16K-entry L2, 14-bit history XORed with address
RAS Size	64
BTB Size	2K-entry 4-way
RUU Size	128
LSQ Size	64
Integer ALUs	4 (1-cycle latency)
Integer Multipliers/Dividers	2 (3,20)
FP ALUs	2 (2)
FP Mult./Div./Sqrt.	1 (4,12,24)

VCT/VCSR Parameters

Parameter	Value
AVF bounds	0%, 2%, 5%, 10%, 20% 50%, 100%
Result Buffer	128-entry RAM segment + 384-entry FIFO segment

Memory System Parameters

Parameter	Value
L1 D-Cache Ports	2
L1 D-Cache	64KB, 4-way with 32B block (2)
L1 I-Cache	64KB, 4-way with 32B block (2)
L2 Unified Cache	512 KB, 4-way with 64B line-size (12)
I-TLB	512-entries 4-way set-associative
D-TLB	1K-entries 4-way set-associative
TLB Miss-Latency	30 cycles
Memory Latency	200 cycles

Table 5.1. Simulation parameters. Latencies of ALUs/caches are given in parenthesis. All ALU operations are pipelined except division and square-root.

Figure 5.3 shows the VCSR results for different AVF bounds, in comparison

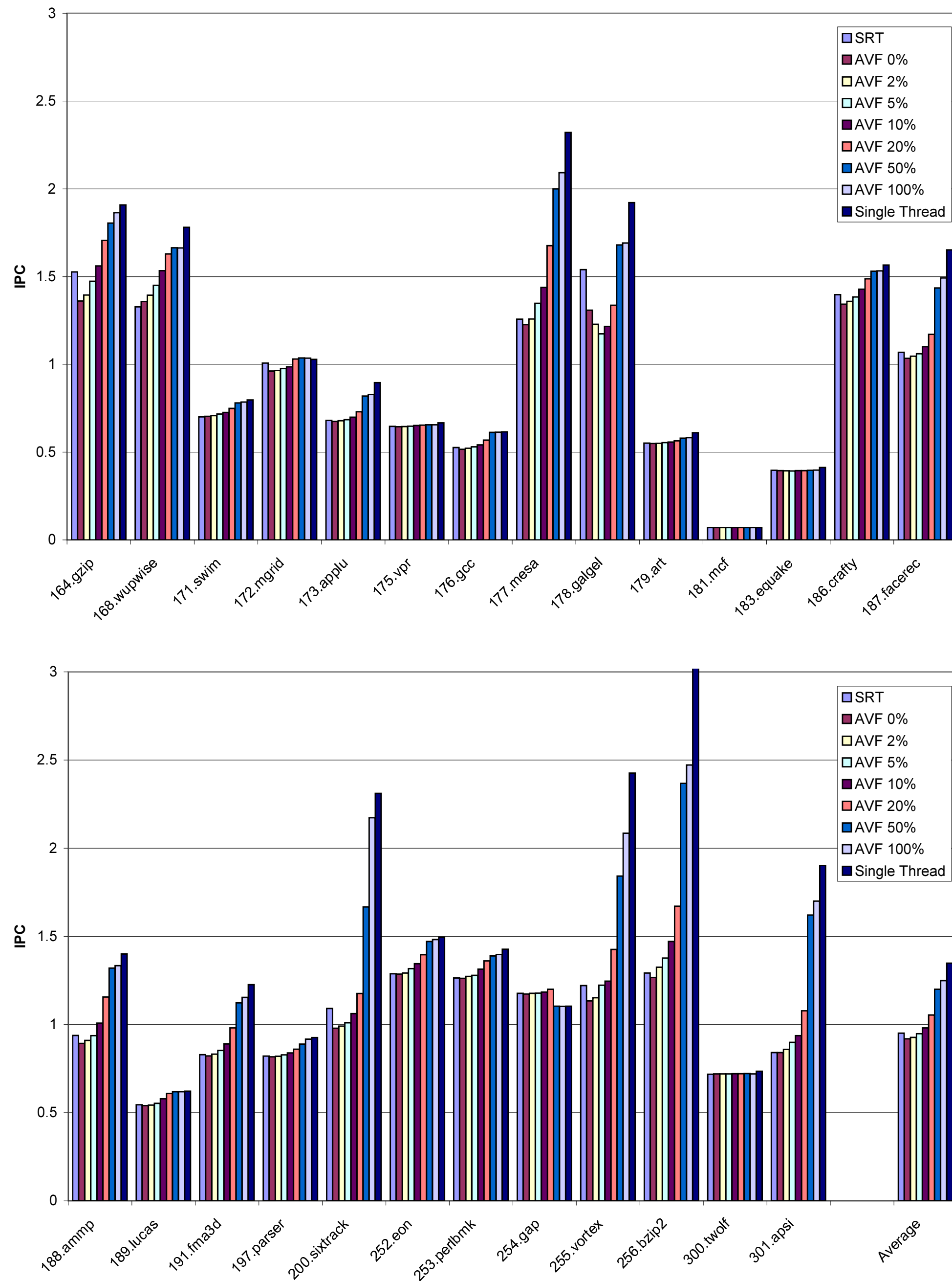


Figure 5.3. IPC with VCSR for various AVF bounds. Also provided for each benchmark are IPCs for single-thread execution (rightmost bar) and SRT (leftmost bar).

with SRT and single-thread executions. A good VCSR implementation should perform close to single-threaded execution for a 100% AVF bound (no instructions are redundantly executed), and close to SRT for a 0% AVF bound (all instructions are selected for re-execution). In these two scenarios, our VCSR implementation

suffers a performance degradation of 7% and 3% respectively, which we consider to be fairly efficient.

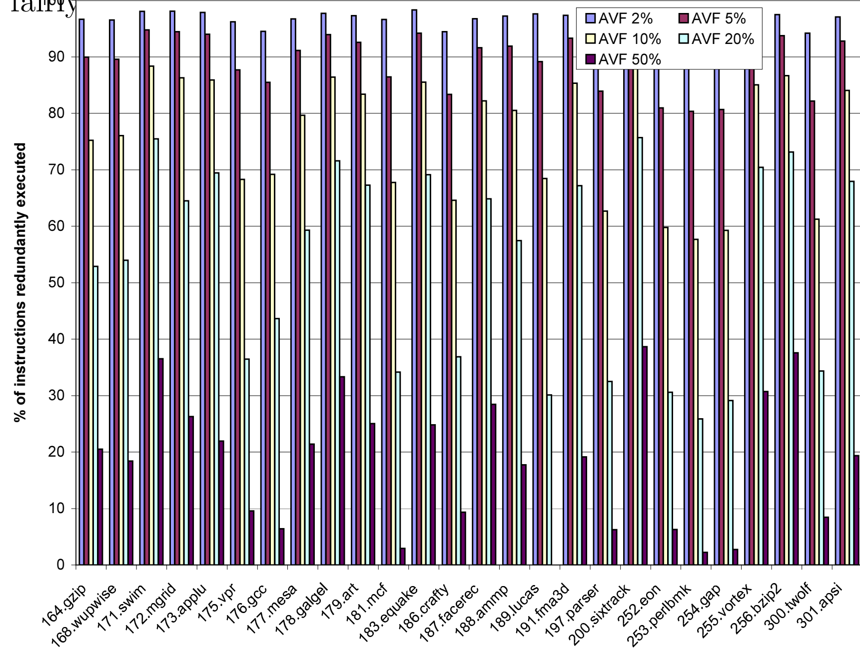


Figure 5.4. Percentage of instructions redundantly executed with VCSR for different AVF bounds.

The graphs show that VCSR is able to provide a wide spectrum of operating points in the performance-reliability tradeoff space. As expected, lowering the AVF bound reduces the IPC of the execution. This is because more instructions are picked for redundant execution as is shown in Figure 5.4. Memory bound applications such as *171.swim*, *179.art*, *181.mcf* and *189.lucas*, do not show much variation in IPC since memory stalls dominate execution time, and there is not much contention for datapath resources. Consequently, even when over 95% of the instructions are redundantly executed, the performance loss compared to single-thread execution is only around 5% in these applications. This is also the reason why SRT performance in these applications is not significantly worse than single-thread execution.

Resource contention in VCSR is expected to play a more detrimental role in applications with high IPC. For instance, applications such as *256.bzip2*, *255.vortex* and *200.sixtrack* show significant drop in IPC even with AVFs bounds of 20% with VCSR. The resource contention in these high ILP applications is more detrimental to performance.

In general, VCSR causes resource contention for high ILP applications when the AVF bounds are loose (i.e., the drop in IPC is steeper at high AVF bounds than at low AVF bounds) and incurs overheads in its implementation (i.e., at 100% AVF bound, the performance is still not the same as single-thread). We pictorially illustrate this observation in Figure 5.5. This graph plots the performance normalized with respect to single-thread execution for different AVF bounds. Rather than show this for each application, the graph has been drawn by taking the geometric mean of the slowdowns observed by all 26 applications.

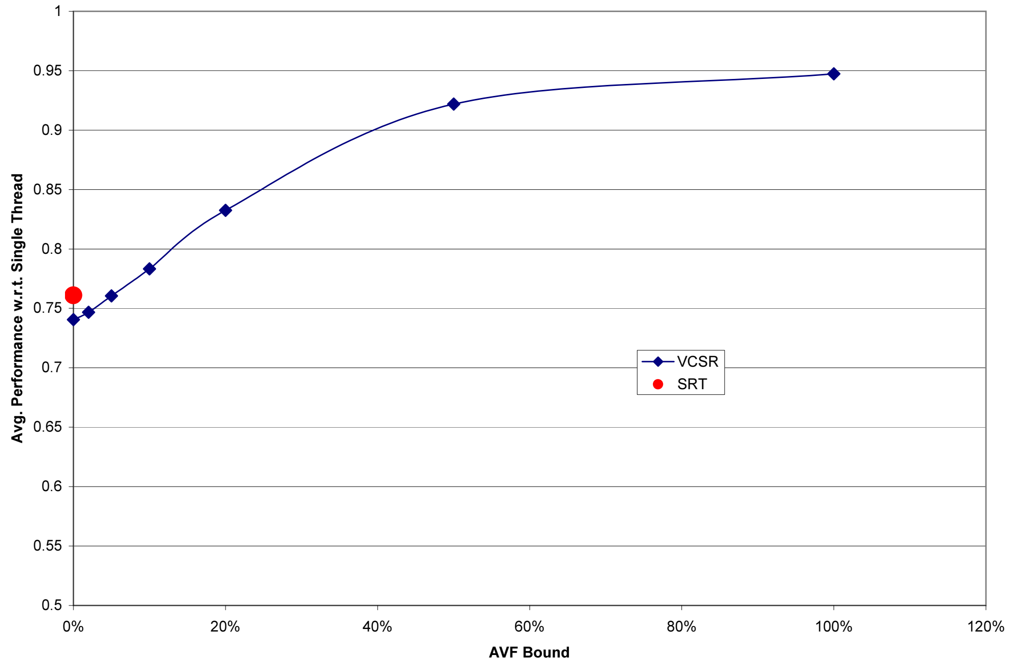


Figure 5.5. Summary of Results. Geometric Mean of IPC across Applications Normalized w.r.t. Single Thread IPC for different AVF bounds. Also shown is SRT performance on the y-axis (i.e. AVF=0).

5.6 Concluding Remarks

In this chapter, we presented knobs for controlling the vulnerability of processor structures that can be modulated to meet target reliability budgets specified by system designers. We have proposed mechanisms to monitor and control the vulnerability of the ROB of a processor at a cycle-level fidelity. The control mechanism

operates by reactively (by re-execution) ensuring that the number of vulnerable bits in the ROB do not exceed a specified threshold. Using detailed simulations we have evaluated the pros and cons of our approach and compared them with default single-threaded and fully redundant executions.

Our results show that the selective redundancy mechanism can span the entire performance spectrum between complete redundancy and single thread execution while meeting any specified budget.

Chapter 6

Future Work

6.1 Vulnerability Control

6.1.1 VC on Multiple Structures

In the vulnerability control mechanisms described in this thesis, we focused primarily on bounding the vulnerability of the ROB. However, in order to meet the overall system reliability budget, the vulnerability of multiple processor structures would need to be controlled. One way to achieve this could be to "apportion" the total allowable processor vulnerability to multiple pipeline structures either statically or dynamically while simultaneously attempting to maximize overall processor performance.

In principle, selective redundancy as a VC mechanism should be applicable to most pipeline structures, but simultaneously controlling vulnerabilities of multiple structures leads to a multi-dimensional optimization problem. This is especially interesting because of the fact that controlling the vulnerability of one structure could impact the vulnerability of other structures.

Fortunately, redundant execution in general benefits all structures in the pipeline. For example, reduction in vulnerability of the ROB should lead to reduction in vulnerability of structures like the Issue Queue and Load-Store Queue as well. We provide some preliminary results which show the observed impact of bounding the ROB's AVF (shown on the x-axis) on the AVF of the Issue Queue in Figure 6.1. As expected, VCSR on the ROB also reduces AVF of the Issue Queue.

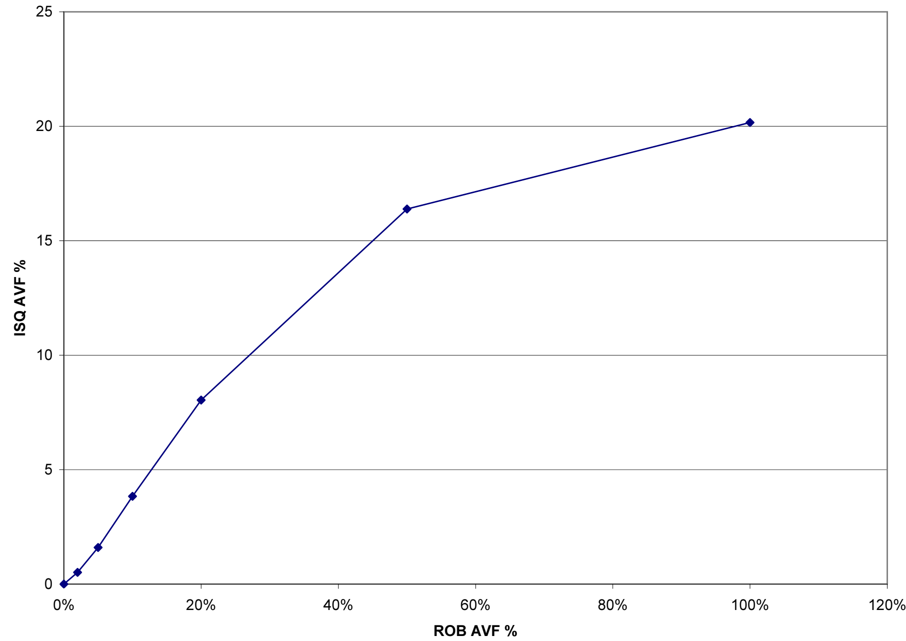


Figure 6.1. Effect of controlling the AVF of the ROB on the AVF of the Issue Queue.

Our future work involves examining the impact of vulnerability control of several pipeline structures on observed vulnerabilities of other structures, and developing heuristics to dynamically perform multi-structure VC to maximize performance while remaining under the overall processor reliability budget.

6.1.2 VCSR Implementation Issues

Implementing instruction-level selective redundancy via the Result Buffer is a moderately complex approach, since the result buffer is a frequently accessed structure requiring sufficient ports to maintain the instruction-commit throughput. Our implementation is also unable to reduce pressure on the fetch engine compared to full redundancy since all instructions are redundantly fetched and decoded.

These issues could be addressed to a certain extent by performing selective redundancy at a coarser granularity, e.g., at an instruction cache block or trace level. However, ensuring that AVF bounds are satisfied at a cycle-level fidelity could become more difficult to achieve.

As we showed in Chapter 4, performing partial redundancy at the granularity of slices avoids producer-consumer dependency issues due to the partial set of executed instructions. However, instruction residence periods, and in turn their

contribution to structure vulnerabilities seem to follow a more sequential pattern that is not necessarily related to any dependence relationships. Therefore, while convenient for implementation, slice-granularity selective redundancy could turn out to be unsuitable for use with vulnerability control. We will analyze these issues in greater detail in our future work.

6.2 Chip-Level Redundancy

With thermal, power and performance constraints limiting the growth of single-threaded performance, chip multiprocessors (CMP) are gaining popularity as a means to exploit parallelism at thread and task-level granularities. CMPs also provide a convenient platform to implement redundant threading.

Studies on CMP-based redundant threading [30, 16] have shown that executing the redundant threads on multiple cores of a CMP would require implementing cross-core LVQs and SCBs. This is not very straightforward since high-bandwidth, low-latency access is required to these structures from both cores. The requires the cores need to be laid out in a particular manner in order to ensure that latencies and area overheads are minimized. If it is desirable to place the two redundant threads physically far from each other (for example, on cores that are physically far apart or even on different chips, due to thermal issues or for enhanced reliability), a set of dedicated, sequential buses between the threads would be infeasible. This motivates exploration of alternative techniques for performing IR/OC for chip-level redundant execution systems.

Checkpointing System: Our goal is to design a transient fault detection mechanism that is able to operate in tandem with a checkpoint-based Backward Error Recovery system similar to that described in [36]. Checkpoints are created using logging, and for multi-threaded applications, all threads are made to synchronize and halt before a consistent checkpoint is marked.

A detection system designed to work with such a recovery model needs to be able to accept a system-wide *Checkpoint* signal and perform the following actions upon receipt of the signal:

- All internal state needs to be flushed into into main memory so that it can

be seen by the logging logic.

- All data seen by the logging/checkpointing mechanism since the previous checkpoint must be guaranteed to be correct (i.e., free from the class of errors that the detection system is designed to detect).

The first requirement can be implemented by first making the processor insert Store micro-operations which spill the register file contents into memory space upon receipt of the Checkpoint signal, and then flushing all dirty data from the caches to main memory. The second requirement is easily satisfied by ensuring that all traffic to main memory has undergone Output Comparison so that no erroneous data is ever seen by the logging mechanism.

Threading Model: While implementing an efficient cross-core redundant execution system for multi-threaded applications is a complex task, achieving this for simple single-threaded applications is also non-trivial. In the following discussion, we highlight a few design issues and tradeoffs that could arise while implementing IR/OC at different points in the system for single-threaded applications.

6.2.1 The View of Memory

We refer to the two redundantly executing copies of a single-threaded program as Redundant Copies (RCs) or Redundant Siblings (RSs). The memory images of the RCs of a single-threaded program can be conceptually viewed as completely independent address spaces. In the absence of any external events, the two executions are self-sufficient and can be allowed to proceed completely independently, without any need to perform input replication or output comparison.

Output Comparison for a single-threaded application is only required (a) for checkpoint creation, and (b) on the occurrence of interrupts and exceptions. For simplicity of implementation, it makes sense to force a checkpoint before handling any interrupt or exception. Thus, output comparison needs to be performed only to maintain consistency with the checkpointing mechanism.

Purely from a correctness point of view, explicit Input Replication for a single-threaded application is not required for clean data that no RC has written. However, once an RC writes a memory location, this dirty data must be maintained

separately for the writer RC until an output comparison has taken place with the Redundant Sibling. This is trivial to achieve in an infrastructure built to support OC; all that needs to be ensured is that a Read always returns the RC's most recent write for data that has not undergone OC.

6.2.2 IR/OC at the Processor-L1 Cache interface

This is the traditional approach to implementing a SoR using an LVQ and SCB, and has been used by a several RMT proposals to date [30, 56, 16, 34, 17]. Performing output comparison at the processor-L1 cache interface is straightforward because in the absence of errors, the Store addresses and data generated by the processor follow a deterministic, sequential pattern. Even in the presence of benign manufacturing defects in the processor (faulty branch predictor, disabled functional units, etc.), this sequence will be deterministic both in terms of content as well as sequence.

In such a scenario, output comparison can be performed by pairwise comparisons of Stores generated by the two RCs. This comparison requires an interconnection network between the physical cores where the RCs are executing. The interconnect should have sufficient bandwidth to support the rate at which the processor is expected to generate Stores. In case some amount of slack tolerance is required between the two RCs, sufficient buffering needs to be provided to avoid stalling either of the two executions.

Buffering creates a problem with Store-to-Load forwarding. If Stores generated by a processor are kept in a buffer until they are verified by the Redundant Sibling, loads following the stores could require data to be forwarded to them from the buffer. This requires the buffer to be made searchable associatively, thereby constraining its size. Further, this could lead to Load latency issues (depending on physical placement of the buffer with respect to the two RCs). Some of these issues have been addressed in [38], but the proposal does not address the primary cross-core interconnect and bandwidth issue.

6.2.3 IR/OC at an On-Chip location in the Memory Hierarchy

Placing the output comparison point between any two cache levels in the memory hierarchy gives rise to two potential concerns:

- The order of writebacks (outputs from the cache) from the last cache level within the SoR is not necessarily deterministic. It is possible to make this deterministic by necessitating that both Redundant cache Siblings are exactly identical in terms of physical layout, presence/absence of defects, enabled/disabled cache lines, initial state, interference with other applications that may be sharing the cache space, etc. Due to the large areas that future on-chip caches are expected to occupy, as well as the increasingly large susceptibility of circuits to defects and process variations, constraining sibling caches to be physically identical could result in reduced yields. Further, requiring these caches to be free of interference effects from other applications would also unnecessarily constrain cache utilization in future massively-multi-core architectures.

- The order of writebacks from the last cache level within the SoR has no correlation with program sequence or ordering. This implies that even if a dirty-line writeback is verified by both RCs, the writeback cannot necessarily be immediately sent out of the SoR. This is because *older* (according to program order) unverified writes could still be residing in the caches within the sphere. If an error is later detected in any of these writes, then the younger verified writeback that was sent out will need to be rolled back.

  In practice, if a checkpointing recovery model is used, then this issue is automatically resolved. Establishing a checkpoint requires flushing all caches within the sphere, which ensures that checkpoints are consistent, and the order in which writebacks are verified and sent out within a checkpoint interval is irrelevant.

6.2.3.1 The Redundancy-Merging Buffer

We examine the feasibility of using a fully-associative buffer (which we call the Redundancy-Merging Buffer or RMB) to perform output comparison. The RMB is searchable using aligned cacheline addresses. Every buffer entry requires space for storing the search key (address), and one copy of cacheline data for each redundant sibling. A standard writeback buffer could be augmented with merging capabilities in order to transform it into a RMB.

A writeback from a RC first searches the buffer for an address match. If an entry is found, the data is copied over into the data location corresponding to this RC. If no match is found, then a new entry is allocated for this address. Reads to a memory hierarchy level higher than the place where the RMB is located need to first search the buffer for younger writes (similar to a writeback buffer). Since the buffer maintains independent copies of the data for both RCs, input replication is correctly performed.

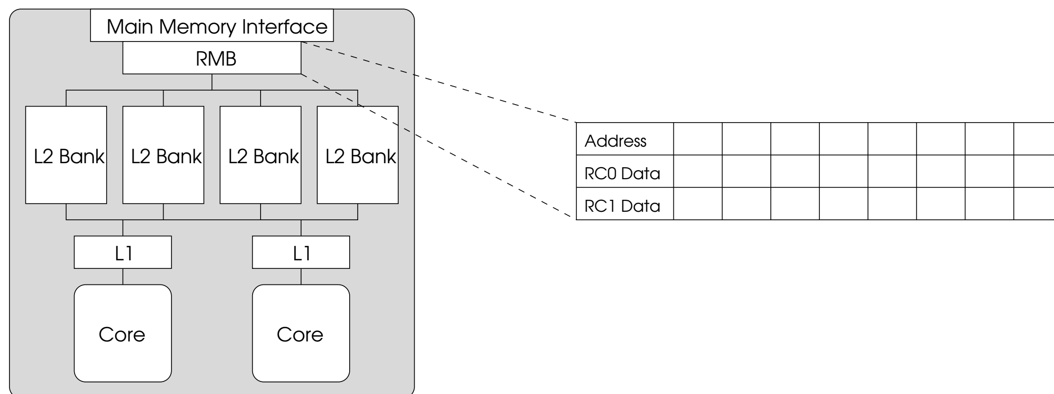


Figure 6.2. The Redundancy-Merging Buffer.

Figure 6.2 shows a RMB designed to perform output comparison at the L2-main memory interface. The buffer is placed on the chip at the interface to the main memory bus. Writebacks from the L2 cache of an RC are sent to the buffer where they are buffered until the corresponding writeback from the redundant sibling arrives, following which the write can be sent out over the FSB to the checkpoint/logging logic and main memory.

Data Versions: Note that it is possible for different versions of writeback data from the two RCs to appear to match in the RMB. For example, consider a loop that repeatedly writes the same value to a particular address. If the two RCs are executing in different iterations of the same loop, then it is possible for the RMB to see writebacks from the two RCs that have exactly the same data values, but are actually from different versions (in terms of number of processor stores) of the data. Fortunately, the RMB functionality can be agnostic to this issue without any impact on correctness.

We present some preliminary results obtained via a 2-stage simulation process. In the first stage, we run independent simulations for each RC with realistic cache configurations but a perfect zero-latency main memory using cycle-accurate processor and cache models. In the second stage, we use a custom RMB and main-memory simulator that takes as input read and writeback traces from the two first-stage RC simulations.

Bandwidth Requirements: We measure and compare the on-chip and off-chip bandwidth requirements of a traditional SoR with a cross-core Store Checking Buffer (SCB) against the requirements of a RMB-equipped configuration performing IR/OC at the main memory interface on the chip. Since we wish to measure the bandwidth *demand* of the two configurations (as opposed to the effective observed bandwidth over a particular simulation run), we observe the total bandwidth requirement over a simulation run for a fixed number of instructions.

Our simulations take into account the fact that the RMB-based implementation would lead to an effective reduction in L2 cache capacity due to replication. We assume that the entire L2 cache area can be used by any core on the chip, thereby enabling the SCB implementation to enjoy a much larger usable cache capacity.

Figure 6.3 shows the On-Chip bandwidth required (in terms of total bytes for a 100 million simulation run) for implementing a RMB-based SoR compared to a SCB-based SoR. Data is shown for several SPEC CPU2000 applications. In addition to the difference in bandwidth requirements, note that while the SCB implementation would require *additional* data buses purely for IR/OC, the RMB implementation simply uses the existing on-chip data channels from the L2 to the off-chip interface. From the figure, the RMB implementation requires anywhere

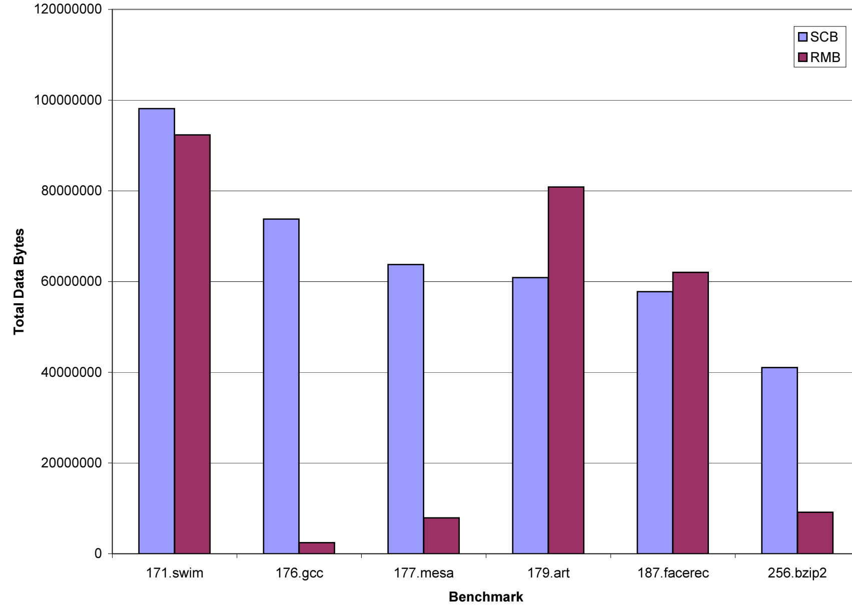


Figure 6.3. Total data transferred on-chip for Output Comparison, shown for SCB and RMB implementations.

from 3% (for *177.mesa*) to 132% (for *179.art*) of the on-chip bandwidth of an SCB implementation, with a geometric mean of about 33%. The increased RMB bandwidth for the memory-intensive applications such as *179.art* is due to the reduction in effective L2 capacity because of replication.

This replication is also responsible for the increased Off-Chip bandwidth requirements of a RMB implementation, as shown in Figure 6.4. The off-chip bandwidth demand for RMB compared to SCB ranges from 100% (for *176.gcc*) to 318% (for *179.art*), with a geometric mean of 135%.

Deadlocks: One of the primary purposes of the RMB is to be able to handle situations where the cache hierarchies for the RCs within the SoR may not be exactly identical in terms of physical organization or intermediate state. Although it is guaranteed that the processor cores in both RCs will generate the exact same sequence of Store instructions, differences in the state or organization of the cache hierarchy would lead to different patterns of writeback data from the two RCs into the RMB. In certain situations, this could lead to deadlocks if the RMB gets filled up.

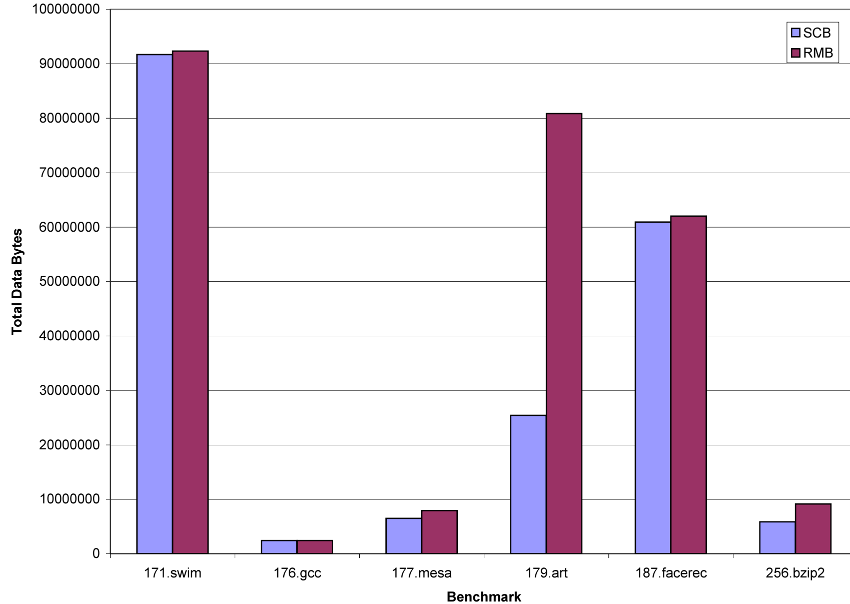


Figure 6.4. Total data transferred off-chip for SCB and RMB implementations.

Breaking Deadlocks: It is difficult to break out of these deadlock situations. One solution is to roll back to the previous checkpoint and either: (a) proceed execution with the guarantee that the caches in the two RCs will now behave identically (possible if the caches are physically identical and the deadlock came about because of transient interference effects from other threads in the cache), or (b) proceed execution with caches disabled (i.e., treating all memory accesses as uncached accesses) until the deadlock is resolved.

Deadlock Analysis: For this analysis, we configure the two RCs to have slightly different cache configurations, which result in different sequences of writeback data, leading to deadlocks. We count the number of simulated cycles in before the RMB-simulator signals a deadlock. We run two sets of experiments in which the RCs are configured as follows:

- **8,4:** 8-way L2 cache vs. 4-way L2 cache with equal capacity (i.e., the 4-way cache has twice the number of sets that the 8-way cache does).
- **8,reduced7:** 8-way L2 cache vs. 7-way reduced-capacity L2 cache (this simulates a defective cache which for some reason had to have one of its

ways shut down).

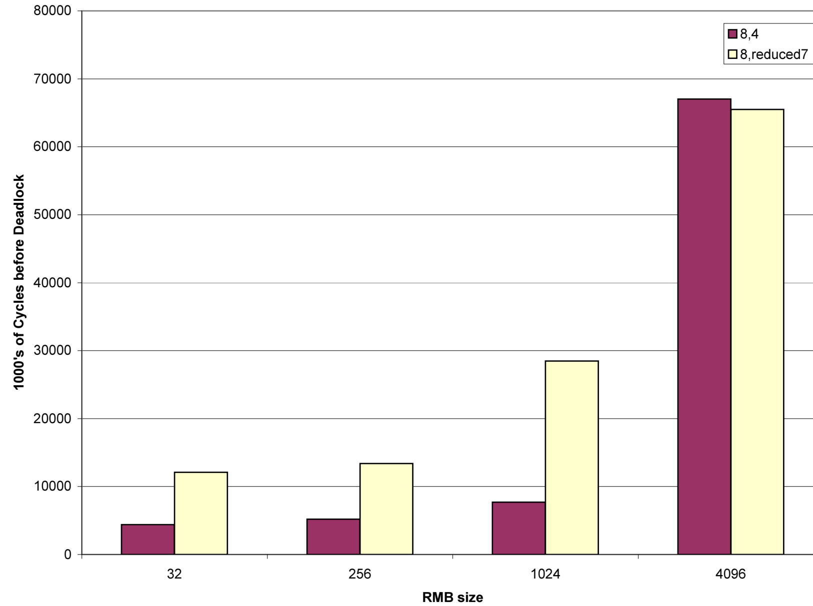


Figure 6.5. 1000's of cycles executed before RMB is deadlocked, for multiple RMB sizes.

The results are shown in Figure 6.5, which shows the average number of cycles the benchmarks were able to run before hitting a deadlock. RMB sizes ranging from 32 to 4096 are shown. Note that large sizes such as 1024 and 4096 are shown purely for illustrative purposes, since it is infeasible to construct associative structures with such sizes. Note, however, that the RMB needs to be clocked at a far less frequency than the core frequency, since it only handles L2 writeback data. The results indicate that buffer sizes of 4096 or more are required to avoid any incidence of deadlocks if the RCs differ significantly in their cache organization.

Since deadlock resolution is expensive and could potentially require rollbacks to a previous checkpoint, we recommend that practical implementations actively enable/disable cache ways in order to make sure that the RCs are as similar as possible in organization. Such a step can be taken, for example, if a large number of rollbacks are detected in a short amount of time.

Other variations: We experimented with other combinational variations of RCs, such as siblings operating under different frequencies, siblings starting up

within a certain interval of each other, but these combinations cannot lead to deadlocks in our model. The reason is that as long as the *sequence* of writebacks to the RMB is the same from both siblings, a deadlock cannot occur. While deadlocks are *unlikely* in such situations in a real implementation, they simply cannot occur in our 2-stage simulation model since we cannot back-propagate the effects of stalls and delays from the RMB simulator to the first stage (stalls could potentially lead to re-ordering of memory operations within the processors, which in turn could lead to re-ordering of cache evicts and writebacks).

6.2.4 Off-chip IR/OC

Extending the sphere of replication beyond the chip and into main memory leads to several undesirable effects, including reduction of effective main memory capacity, over-utilization of precious off-chip bandwidth, need for complex memory-renaming mechanisms, etc.

For example, consider the scenario where both RCs are allowed to maintain independent address spaces in physical memory (perhaps set up with the assistance of the Operating System). This leads to an effective halving of main memory capacity, as well as over-utilization of bandwidth on the off-chip memory bus. Off-chip memory bandwidth is considered to be one of the most performance- limiting bottlenecks in modern and future systems.

Further, allowing for redundant storage space within the main memory arrays would require microarchitectural memory renaming, which is a significantly complex issue in itself. Multiple data values with the same physical addresses would need to be stored in the memory array, with support for checking during output-comparison. Note that this is much easier to do in a set-associative cache, due to its tag-based design.

6.2.5 Discussion

Our preliminary experiments have shown that designing an efficient IR/OC mechanism could be a complex task even for single-threaded applications. While at first glance the RMB appears to be a simple structure to implement, our analysis showed that deadlocks could lead to design complexity and performance issues.

An RMB based approach provides significant savings in on-chip bandwidth, and is able to leverage existing on-chip buses to perform IR/OC. However, due to data replication within the L2 cache, its effective capacity is reduced, leading to an increase in off-chip bandwidth demand for most applications, and a small increase in on-chip bandwidth demand for some applications.

Shared-memory multi-threaded applications introduce an entirely new set of coherence and consistency-related issues, some of which have been addressed in recent work [49]. RMB-based IR/OC implementations will need to address the requirements of multi-threaded applications as well. Detailed analysis and evaluation of all such issues is an intended goal for this work.

Despite its drawbacks, one of the most appealing reasons to use an RMB-based solution is that no dedicated core-to-core interconnection networks are required to implement the redundancy. Most of the implementation complexity is restricted to the chip area where the buffer will reside. Therefore, RMB-style redundancy solutions are a promising direction for further exploration towards the design of efficient reliable, high-performance microarchitectures.

Bibliography

[1] P. Ahuja, D. Clark, and A. Rogers. The Performance Impact of Incomplete Bypassing in Processor Pipelines. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, November 1995.

[2] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 196–207, November 1999.

[3] M. Brown, J. Stark, and Y. Patt. Select-Free Instruction Scheduling Logic. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 204–213, December 2001.

[4] D. Burger and T. Austin. The SimpleScalar Toolset, Version 3.0. http://www.simplescalar.com.

[5] M. Burtscher. An Improved Index Function for (D)FCM Predictors. *ACM SIGARCH Computer Architecture News*, 30(3):19–24, June 2002.

[6] C. Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4):14–19, July-August 2003.

[7] Cacti 3.2. http://research.compaq.com/wrl/people/jouppi/CACTI.html.

[8] S. Chatterjee, C. Weaver, and T. Austin. Efficient Checker Processor Design. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 87–97, December 2000.

[9] D. Citron and D. Feitelson. The Organization of Lookup Tables in Instruction Memoization. Technical Report 2000-4, Hebrew University of Jerusalem, March 2000.

[10] D. Citron and D. Feitelson. Revisiting Instruction Level Reuse. In *Proceedings of the Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, May 2002.

[11] D. Citron and D. Feitelson. "Look It Up" or "Do The Math": An Energy, Area, and Timing Analysis of Instruction Reuse and Memoization. In *Proceedings of the Workshop on Power-Aware Computer Systems*, December 2003.

[12] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic Speculative Precomputation. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 306–317, December 2001.

[13] E. Duesterwald, R. Gupta, and M. L. Soffa. Distributed slicing and partial re-execution for distributed programs. In *Languages and Compilers for Parallel Computing*, pages 497–511, 1992.

[14] T. Ehrhart and S. Patel. Reducing the Scheduling Critical Cycle using Wakeup Prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, February 2004.

[15] X. Fu, J. Poe, T. Li, and J. A. B. Fortes. Characterizing microarchitecture soft error vulnerability phase behavior. In *MASCOTS '06: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 147–155, Washington, DC, USA, 2006. IEEE Computer Society.

[16] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 98–109, June 2003.

[17] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 172–183, 2005.

[18] D. Grunwald, A. Klauser, S. Manne, and A. R. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 122–131, 1998.

[19] S. Gurumurthi, A. Parashar, and A. Sivasubramaniam. SOS: Using Speculation for Memory Error Detection. In *Proceedings of the Workshop on High Performance Computing Reliability Issues (held in conjunction with HPCA)*, February 2005.

[20] HP NonStop Himalaya. http://nonstop.compaq.com/.

[21] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar. The Optimal Logic Depth Per Pipeline State is 6 to 8 FO4 Inverter Delays. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 14–24, June 2002.

[22] J. J. Koppanalil and E. Rotenberg. A simple mechanism for detecting ineffectual instructions in slipstream processors. *IEEE Transactions on Computers*, 53(4):399–413, 2004.

[23] S.-J. Lee and P.-C. Yew. On Some Implementation Issues for Value Prediction on Wide-Issue ILP Processors. In *Proceedings of the International Conference on Parallel Architectures and Compliation Techniques (PACT)*, pages 145–156, October 2000.

[24] K. Lepak, G. Bell, and M. Lipasti. Silent Stores and Store Value Locality. *IEEE Transactions on Computers*, 50(11):1174–1190, November 2001.

[25] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. Softarch: An architecture level tool for modeling and analyzing soft errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 496–505, 2005.

[26] M. Lipasti and J. Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 226–237, December 1996.

[27] A. Mendelson and N. Suri. Designing high-performance and reliable superscalar architectures-the out of order reliable superscalar (O3RS) approach. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 473–481, June 2000.

[28] E. Morancho, J. Llabera, and A. Olive. Recovery mechanism for latency misprediction. In *Proceedings of the 2001 ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[29] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-processors: an implementation of operation-based prediction. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 321–334, 2001.

[30] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 99–110, May 2002.

[31] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 29–40, December 2003.

[32] S. Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, University of Wisconsin - Madison, 1998.

[33] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 206–218, June 1997.

[34] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 376–386, June 2004.

[35] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. Slick: Slice-based locality exploitation for efficient redundant multithreading. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

[36] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 111–122, Washington, DC, USA, 2002. IEEE Computer Society.

[37] M. K. Qureshi, O. Mutlu, and Y. N. Patt. Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 434–443, 2005.

[38] M. W. Rashid, E. J. Tan, M. C. Huang, and D. H. Albonesi. Exploiting coarse-grain verification parallelism for power-efficient fault tolerance. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 315–328, Washington, DC, USA, 2005. IEEE Computer Society.

[39] J. Ray, J. Hoe, and B. Falsafi. Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 214–224, December 2001.

[40] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 25–36, June 2000.

[41] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*, pages 84–91, June 1999.

[42] S. R. Sarangi, J. T. Wei Liu, and Y. Zhou. Reslice: Selective re-execution of long-retired misspeculated instructions using forward slicing. In *Proceedings of*

the 38th annual IEEE/ACM International Symposium on Microarchitecture, pages 257–270, 2005.

[43] J. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors (Beta Edition)*. McGraw Hill, 2003.

[44] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[45] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on Soft Error Rate of Combinational Logic. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2002.

[46] T. Slegel et al. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, 19(2), March 1999.

[47] J. Smolens, B. Gold, J. Kim, B. Falsafi, J. Hoe, and A. Nowatzyk. Fingerprinting: Bounding Soft-Error Detection Latency and Bandwidth. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 224–234, October 2004.

[48] J. Smolens, J. Kim, J. Hoe, and B. Falsafi. Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 257–268, December 2004.

[49] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *MICRO '06: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 223–234, Washington, DC, USA, 2006. IEEE Computer Society.

[50] A. Sodani and G. Sohi. Dynamic Instruction Reuse. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 194–205, June 1997.

[51] A. Sodani and G. Sohi. Understanding the Differences Between Value Prediction and Instruction Reuse. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 205–215, December 1998.

[52] N. Soundararajan, A. Parashar, and A. Sivasubramaniam. Mechanisms for Bounding Vulnerabilities of Processor Structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA) (to Appear)*, June 2007.

[53] J. Stark, M. Brown, and Y. Patt. On Pipelining Dynamic Instruction Scheduling Logic. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 57–66, December 2000.

[54] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 257–268, 2000.

[55] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 392–403, June 1995.

[56] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery via Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 87–98, May 2002.

[57] K. Wang and M. Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 281–290, December 1997.

[58] N. J. Wang and S. J. Patel. Restore: Symptom based soft error detection in microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 30–39, 2005.

[59] T.-Y. Yeh and Y. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 124–134, May 1992.

[60] J. Yi, R. Sendag, and D. Lilja. Increasing Instruction-Level Parallelism with Instruction Precomputation. In *Proceedings of Euro-Par*, August 2002.

[61] J. Zeigler. Terrestrial Cosmic Rays. *IBM Journal of Research and Development*, 40(1):19–39, January 1996.

[62] H. Zhou, J. Flanagan, and T. Conte. Detecting Global Stride Locality in Value Streams. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 324–335, June 2003.

[63] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 172–181, June 2000.

Vita

Angshuman Parashar

Angshuman was born and raised in the city of Guwahati in the north-east of India. After his fifth grade he moved to New Delhi, the capital city of India. He completed his Bachelor of Technology degree in Computer Science and Engineering from the Indian Institute of Technology, Delhi, and joined the Ph.D. program at the Pennsylvania State University in the Fall of 2002. Angshuman spent three months as an intern at the IBM T.J. Watson Research Center in New York, and seven months at Intel Massachusetts, Inc. in Hudson, MA.

Angshuman's research interests include Computer Architecture, Operating Systems and Fault Tolerance. He is a member of the ACM.