
THE UNIVERSITY *of* EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

# Performance Optimizations for Compiler-based Error Detection

*Konstantina Mitropoulou*


Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2014

# Abstract

The trend towards smaller transistor technologies and lower operating voltages stresses the hardware and makes transistors more susceptible to transient errors. In future systems, performance and power gains will come at the cost of unreliable areas on the chip. For this reason, there is an increased need for low-overhead highly-reliable error detection methodologies. In the last years, several techniques have been proposed. The majority of them are based on redundancy which can be implemented at several levels (e.g., hardware, instruction, thread, process, etc).

In instruction-level error detection approaches, the compiler replicates the instructions of the program and inserts checks wherever they are needed. The checks evaluate code correctness and decide whether or not an error has occurred. This type of error detection is more flexible than the hardware alternatives. It allows the programmer to choose the protected area of the program and it can be applied without any hardware modifications. On the other hand, the replicated instructions and the checks cause a large slowdown making software techniques less appealing. In this thesis, we propose two techniques that aim at reducing the error detection overhead of compiler-based approaches and improving system's performance without sacrificing the fault-coverage.

The first technique, DRIFT, achieves this by *decoupling* the execution of the code (original and replicated) from the checks. The checks are compare and jump instructions. The latter ones tend to make the code sequential and prohibit the compiler from performing aggressive instruction scheduling optimizations. We call this phenomenon *basic-block fragmentation*. DRIFT reduces the impact of basic-block fragmentation by breaking the synchronized execute-check-confirm-execute cycle. In this way, DRIFT generates a scheduler-friendly code with more instruction-level parallelism (ILP). As a result, it reduces the performance overhead down to 1.29× (on average) and outperforms the state-of-the-art by up to 29.7% retaining the same fault-coverage.

Next, CASTED focuses on reducing the impact of error detection overhead on single-chip scalable architectures that are composed of tightly-coupled cores. The proposed compiler methodology *adaptively* distributes the error detection overhead to the available resources across multiple cores, fully exploiting the abundant ILP of these architectures. CASTED adapts to a wide range of architecture configurations (issue-width, inter-core communication). The results show that CASTED matches the performance of, and often outperforms, sometimes by as mush as 21.2%, the best fixed state-of-the-art approach while maintaining the same fault coverage.

# Lay Summary of Thesis

Designers of today's systems take for granted that the hardware is reliable in the sense that the computations are always correct. However, this is about to change as technology advances towards faster and smaller transistors which are pushed to operate at their physical limits. Future hardware will be unreliable leading to errors in the program's execution. To deal with this problem, new techniques should be developed. Their focus will be to detect and correct these errors in order to make the system reliable again and to guarantee the correct execution of an application on an unreliable hardware. This thesis studies software-based techniques for error detection.

Error detection can be done in hardware or in software. Hardware techniques require redesigning and rebuilding the actual chip. This can be prohibitively expensive and often infeasible. Another alternative is to modify the software without changing the hardware. The changes in the software can be automated by the tools that transform programmer's code to machine code (compiler). Therefore, the software technique can be applied to existing software at minimal cost. This thesis improves the existing state-of-the-art compiler-based error detection techniques.

Existing software-based error detection techniques suffer from low performance. A program with error detection support is considerably slower than a program without it. This is due to the additional program instructions that are needed for checking the program's correctness. The techniques proposed in the thesis reduce the performance overhead while maintaining the same level of reliability as the existing techniques. As a result, they improve the applicability of software-based error detection.

# Acknowledgements

I would like to thank my advisor, Marcelo Cintra, for giving me the opportunity to join his group and for his support and guidance throughout my Ph.D.

I would like to thank all my friends and colleagues for their help and support. In particular, Vasileios Porpodas for helping me getting started with GCC, George Tournavitis and Nikolas Ioannou for their guidance in my early steps, Fabricio Goes for his “never give up” attitude, Hsiu-Chin Lin for keeping me company late at night and in the weekends in the office and Chris Margiolas and Alberto Magni for fighting together against the paper deadlines.

Finally, I would like to thank my sister and my parents for their moral support.

# Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- "DRIFT: Decoupled compileR-based Instruction-level Fault-Tolerance"
  Konstantina Mitropoulou, Vasileios Porpodas and Marcelo Cintra
  International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2013

- "CASTED: Core-Adaptive Software Transient Error Detection for Tightly Coupled Cores."
  Konstantina Mitropoulou, Vasileios Porpodas and Marcelo Cintra
  International Parallel & Distributed Processing Symposium, 2013

(*Konstantina Mitropoulou*)

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The current techniques often used to improve the performance and to reduce the energy consumption of computer systems have made transistors more vulnerable to errors [19][79][88]. Error rate increases as we move to small transistor technologies since transistors become more sensitive to external conditions such as neutrons and alpha particle strikes. In addition, techniques like voltage scaling require transistors to operate close to their voltage limit. This increases the error rate further. Finally, the aging of the hardware is another source of hardware errors.

An important class of hardware errors is transient errors (a.k.a. soft errors) which occur only once and do not persist [84]. Although transient errors are temporal phenomena, they can alter the program's execution. For instance, in the year 2000, Sun Microsystems received several complaints from customers such as America On-line, eBay and Los Alamos Labs, who experienced system failures because of transient errors [51]. The common design practice to deal with transient errors is to replicate the critical components. This makes error detection as simple as comparing the outcomes of the identical components. A mismatch indicates the occurrence of a transient error.

Hardware-based error detection techniques are used in high-availability systems and mission critical environments. In this approach, hardware resources are replicated and the program is concurrently executed on both hardware components and the outcome is continuously checked, also by hardware, for divergence. Typical examples are IBM's G4 and G5 processors [81] where part of program's execution is replicated. HP NonStop series processors [9] are designed with Triple Modular Redundancy (TMR). Similarly, Boeing uses TMR for protection against transient errors [99]. In Power 6 [67] and in Fujitsu's SPARC64 [5], parity and residue checking are used to protect latches against transient events.

Not all users can afford the cost of the extra hardware and the design complexity of hardware-based error detection. Instruction-level error detection might be preferable instead. In this approach, the compiler replicates the code and adds extra comparison instructions (checks) at appropriate times. Then, original, replicated and check instructions are executed in the available non-redundant hardware. There are several advantages of this approach:

i. It is more flexible and cheaper than the hardware design and it can be applied on-the-fly on any system.

ii. It can operate at a higher abstraction level restricting the error detection only to errors that might affect application output.

iii. It gives the designer the flexibility to choose the program region that he wants to protect.

Its main drawback is that code duplication has negative impact on performance.

High fault-coverage instruction-level error detection methodologies face the challenge of effectively managing the error detection overhead without sacrificing reliability. Many approaches have been proposed to achieve this. One such approach is to conscientiously trade off high performance gains for small reliability compromises. *Synchronized* techniques require that the original and redundant code execute synchronously such that the execution is checked in strict intervals. In this way, the strict synchronization guarantees fail-stop behavior [1], but it has negative impact on the code's performance. On the other hand, *decoupled* approaches remove the strict synchronization between the original and the redundant code, and let them slip with respect to one another, while performing the checks slightly later, when convenient. Thus, the program runs faster. However, the system loses its fail-stop capability since the synchronization points are removed.

The first performance optimization we present in this thesis is DRIFT [2] and its main idea is based on decoupling. DRIFT decouples the execution of the original and replicated code from the checking code. In synchronized error detection, the checks frequently interrupt the execution of the original and replicated instructions. DRIFT's decoupling scheme reduces the impact of checks on the execution of the instructions of

[1] An error detection scheme has fail-stop capability, if the error is detected and the execution is stopped immediately after the occurrence of the error. In this way, the error does not produce any irreparable effects on the program execution and output.

[2] DRIFT: Decoupled compileR-based Instruction-level Fault Tolerance

the program. In this way, DRIFT manages to reduce the performance overhead without sacrificing any fault-coverage.

Another approach to improving performance of instruction-level error detection techniques is to exploit as much as possible the available hardware resources. The main problem of instruction-level error detection is that it increases the code size since it generates redundant and checking code. This extra code can be executed either on the same processor as the original code (*thread-local or single-core* technique) or on a separate core (*dual-core technique*). Each scheme is suited for different use scenarios. On one hand, if there are spare cores and the communication latency is low, then the best option is to place the original code on a different core than the replicated code. On the other hand, if the communication latency is high and the cores have high issue-width, then it is preferable to keep the original and the replicated code on one core. This problem is particularly important for tightly-coupled cores [23][90][102]. CASTED [3] deals with this dilemma (single-core or dual-core) for this architecture. CASTED generates code that adjusts to the best configuration at a fine granularity and it distributes the error detection code overhead across the cores taking into consideration the available resources and the communication latency.

## 1.1 DRIFT: Decoupled compileR-based Instruction-level Fault Tolerance

In this thesis, we propose two thread-local error detection methodologies which reduce the error detection overhead without noticeably sacrificing any fault-coverage. DRIFT (Chapter 3) is based on the observation that the frequent checking of the synchronized scheme becomes a performance bottleneck. This is a phenomenon we refer to as *basic-block fragmentation*. The checks break the code into very small basic-blocks with two exiting control edges (Figure 1.1.b). The resulting complex control flow acts as a barrier for aggressive compiler optimizations at the instruction scheduling phase, even for the most aggressive schedulers. For example, in Figure 1.1, the original basic-block BB1 (Figure 1.1.a) is split into three basic-blocks after the introduction of the replicated code and synchronized checks. The scheduler cannot easily move the instructions among basic-blocks to improve ILP because it must strictly respect the program semantics. This is an important restriction that prohibits the compiler from gener-

[3] CASTED: Core-Adaptive Software-based Transient Error Detection

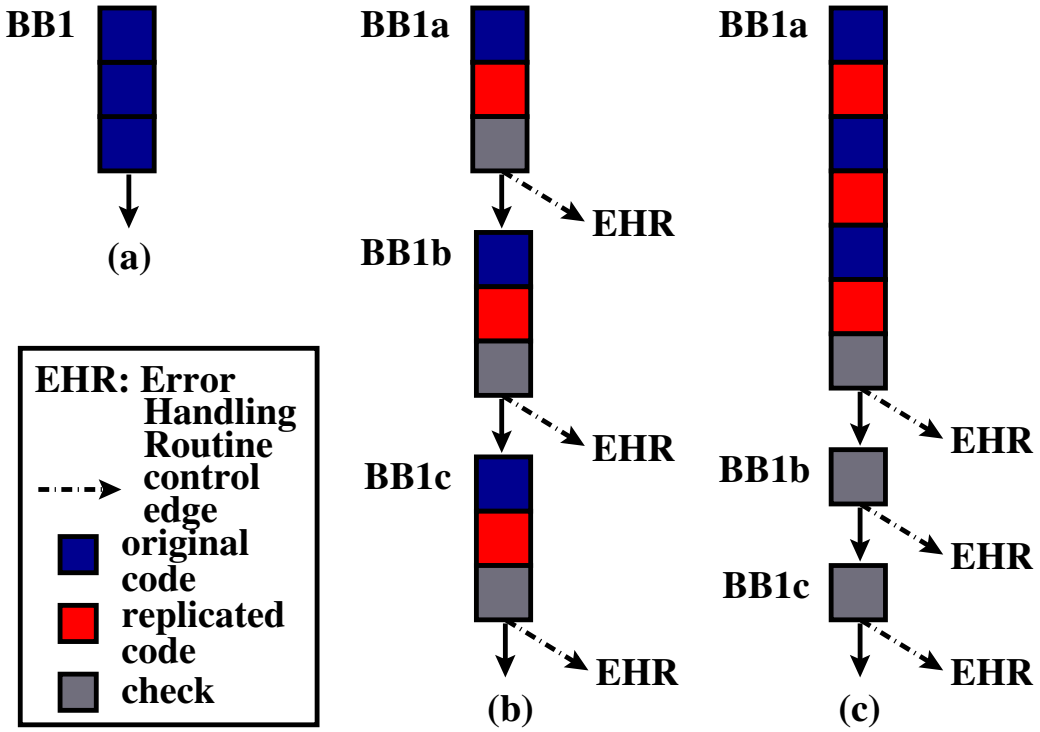


Figure 1.1: (a) Code without error detection, (b) Synchronized thread-local error detection (state-of-the-art), (c) Decoupled thread-local error detection (DRIFT).

ating high performance code for synchronized thread-local error detection. DRIFT introduces a novel decoupled single-core technique that avoids the basic-block fragmentation and improves the performance considerably by relaxing the synchronization between original code, replicated code and checks. It achieves this by clustering the checks (Figure 1.1.c) so as to merge the previously split basic-blocks. In this way, the code is not fragmented into many basic-blocks and can be scheduled more efficiently.

We note that, strictly speaking, our code generation scheme does modify the code semantics. This, however, takes advantage of knowledge of error detection semantics (which are not available to a standard compiler) and does affect the semantics of the resulting program. Therefore, the aggressive code motion that we perform in DRIFT could not have been done automatically by any compiler optimization since the compiler is restricted to always preserving the program semantics.

Our contributions with DRIFT are:

- This work is the first to point out a major performance bottleneck in synchronized error detection caused by basic-block fragmentation.
- DRIFT overcomes the basic-block fragmentation bottleneck by being the first

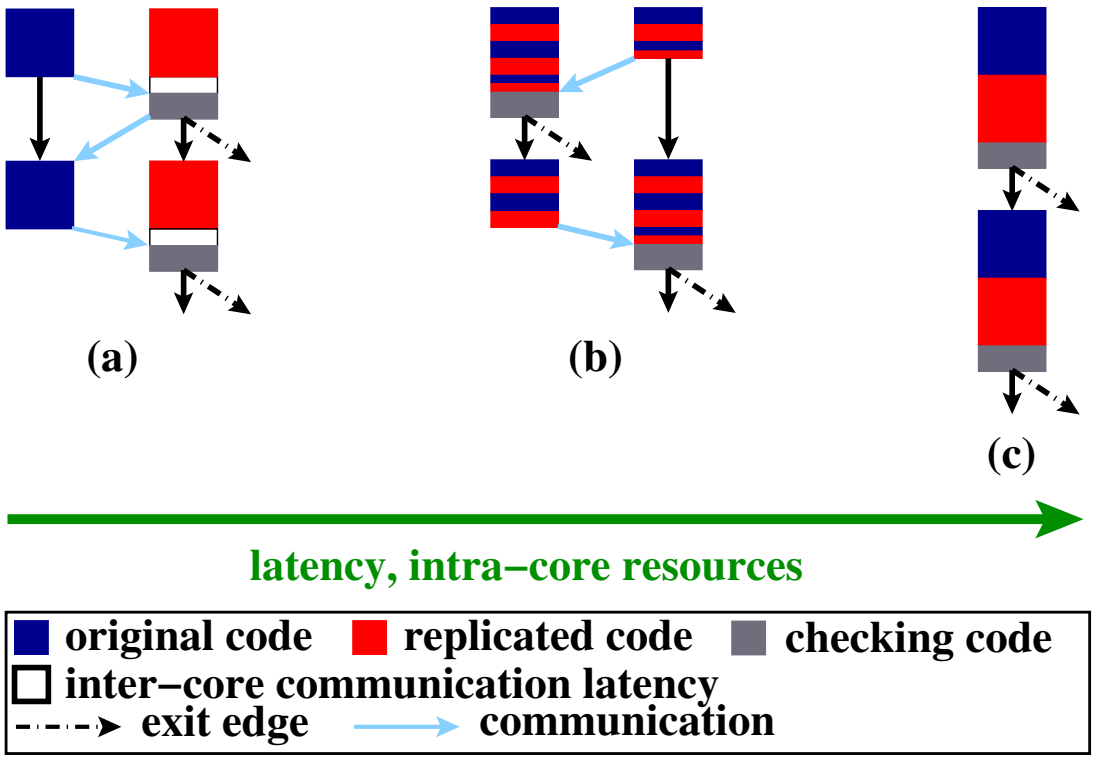


Figure 1.2: Taking into account the architecture configurations, CASTED distributes the code between the cores. If the ILP of each core is small, CASTED performs similar to dual-core technique (a). In case the inter-core communication latency is large, CASTED generates code similar to single-core technique (c). For all the other cases (b), CASTED produces near-optimal code so as to reduce the overhead of error detection.

decoupled single-core error detection scheme.

- DRIFT outperforms the state-of-the-art by up to 29.7% reducing the performance overhead down to 1.29× while retaining high fault-coverage.

## 1.2 CASTED: Core-Adaptive Software-based Transient Error Detection

Thread-local instruction-level error detection places both the original and the replicated code with the checks in the same thread. On the other hand, dual-core methodologies place the original code in one thread and the replicated code with the checks in a second thread. In either case, the resources are underutilized under certain conditions of issue-width, communication latency and application's ILP (Instruction Level Parallelism).

CASTED introduces a technique where the error detection code maps to the cores which improves the resource utilization. The proposed software-based error detection methodology takes advantage of the features (such as fast inter-core communication) of scalable tightly-coupled cores [23][90][102] to distribute the error detection overhead across cores/clusters while maintaining high reliability.

CASTED is a compiler mechanism that generates code for different architecture configurations. Figure 1.2 shows that CASTED can generate code for different architecture configurations. Taking into consideration the available resources and the inter-core communication latency, CASTED distributes the code between the cores in such a way as to minimize the impact of the error detection on performance. In case the cores have very small ILP, CASTED produces code similar to dual-core technique (Figure 1.2.a). On the other hand, if the inter-communication latency is large, then CASTED produces code similar to single-core technique (Figure 1.2.c). For all other cases, CASTED places the code considering the communication latency and the available resources (Figure 1.2.b). In this way, CASTED produces near-optimal code that reduces the error detection overhead. CASTED generates the code in two steps. Firstly, it generates the redundant code (replicated and checking code) for the detection of transient errors. Secondly, it schedules the error detection code taking into consideration the capabilities of the target architecture (issue width, inter-core communication). This suggests that depending on the conditions, the code can be: *i.* executed all within a single core, *ii.* split into sections, some of which will run on one core and the rest on the other.

Existing approaches are non-adaptive, meaning that they assign either the whole program to a single core (Figure 1.2.b), or the original code to one core and the replicated code to another (Figure 1.2.c). This is because they target commodity single-core or multi-core architectures. We show that this is sub-optimal for our target.

Our contributions with CASTED are:

- This work is the first to observe that it is possible to selectively place sections of both original and replicated code in different cores, depending on the available parallelism of each core and the inter-core communication latency.

- CASTED distributes the error detection overhead across the available resources, balancing the use of the resources and optimizing the code for a wide range of core counts, issue widths, and inter-core communication latencies.

- CASTED outperforms the state-of-the-art by up to 21.2% with no impact on the fault coverage.

## 1.3 Thesis Structure

The rest of the thesis is organized as follows: Chapter 2 gives the basic concepts of fault tolerance and the architectures we use to evaluate the proposed techniques. Chapter 3 presents and evaluates DRIFT. Chapter 4 describes and evaluates the CASTED scheme. Chapter 5 overviews the related work. Section 6 concludes this thesis and Section 7 discusses future directions.

# Chapter 2

# Background

## 2.1 Hardware Errors

Until recently the detection of hardware errors was a major design parameter only for niche markets such as military, space or other highly reliable systems. However, as recent studies suggest, the rate of hardware errors increases exponentially [79] as we move to smaller transistor technologies. Therefore, future consumer systems will have to be designed with resilience in mind.

Hardware errors are classified into three categories [84]:

- *Transient (or soft)* errors only occur once and they do not persist.
- *Intermittent* errors occur repeatedly but not continuously in the same place in the processor.
- *Permanent (or hard)* errors are those that persist and result in irreversible physical changes. The faulty component will continue to produce erroneous results every time it is used.

Depending on the category of the error, a different fault tolerance technique is needed. After the detection of a transient error, the re-execution of the program is enough to tolerate the error. This is due to the fact that transient errors just occur once. On the other hard, the permanent errors require additional repair mechanisms. These mechanisms might correct the component that is affected by the permanent error or exclude the component from the execution of the program. Finally, the intermittent errors are treated either as transient or permanent errors depending on the frequency of their occurrence. In this thesis, we focus on transient errors and their detection only.

The main sources of transient errors are:

i. cosmic radiation (neutrons) [7][38][58][104] which is related to altitude. In high altitudes, cosmic radiation is more dense increasing the frequency of transient errors.

ii. alpha particles [8][48][103].

iii. electromagnetic interference (EMI) [63].

These factors in combination with transistor and voltage scaling have contributed to the increasing rate of transient errors. Studies [12][19][30][32][79] have shown that the errors increase as we move to smaller transistor technologies. The smaller transistors operate at smaller voltages. Thus, the critical charge of the transistor is smaller. This enables neutrons and alpha particles to change the charge on a transistor and to create a bit-flip. In [30], the authors showed that the alpha particles become more of a concern than neutrons as we move to smaller transistor technologies.

A system's vulnerability to errors is measured by the Soft Error Rate (SER) metric. SER is typically expressed by two metrics: failures in time (FIT) and mean time between failures (MTBF). FIT measures the number of failures per $10^9$ hours of operation and MTBF measures the elapsed time between two failures.

The measurements of [79] show that the soft error rate increases at a faster pace for combinational logic than the memory as the transistor technology shrinks. The intersection point where the soft error rate of the combinational logic overtakes the memory is at 70nm. Moreover, Hazucha [32] estimates an 8 percent increase in soft error rate per logic state bit for each technology generation. Borkar [13] showed that the soft error rate at 19nm is 100% higher than at 180nm. The above imply that high-reliability low-overhead error detection methodologies are needed to protect the future processors from transient errors. Memory is already protected by techniques such as parity checking and ECC, but there are only few error detection and recovery mechanisms for protecting the processor against transient errors.

## 2.2 Fault Tolerance

### 2.2.1 Overview

Error detection is based on redundancy which can be implemented in space, time or both (e.g., hardware, instruction, thread or process level). This means that the original module (e.g., an ALU unit or a thread) is replicated one or more times. For the detection of transient errors two replicas are considered enough. This is due to two reasons:

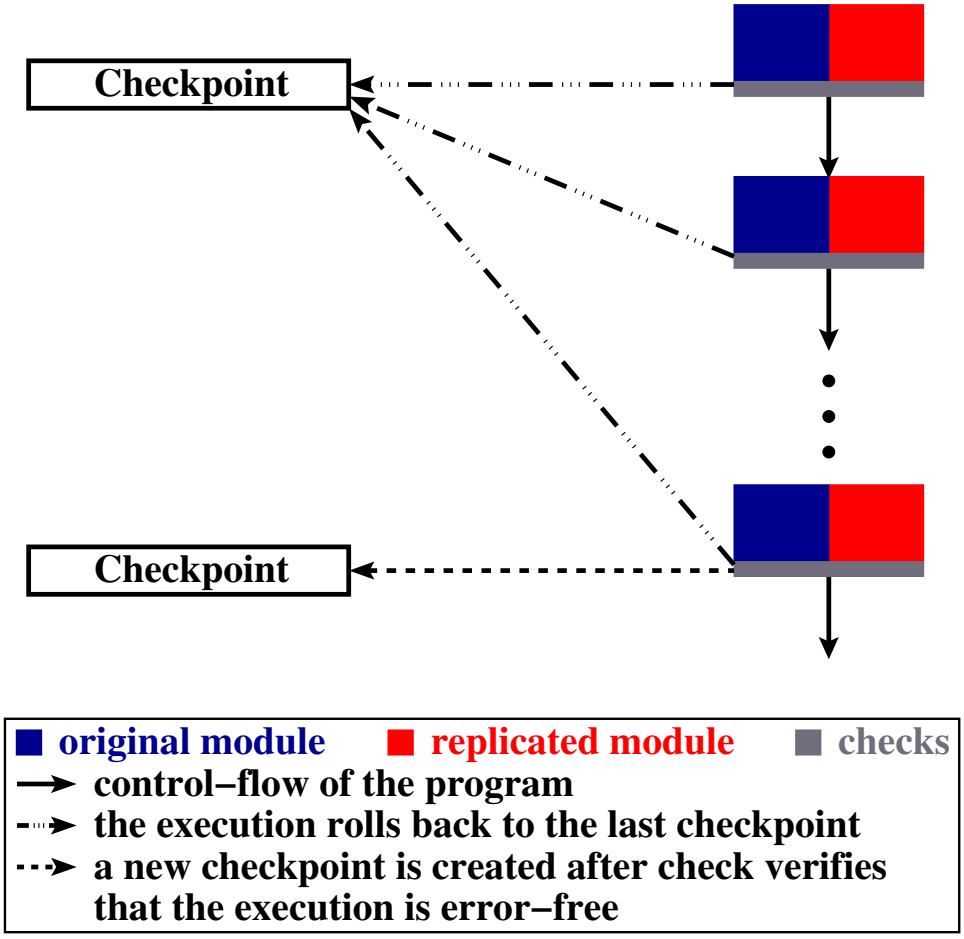


Figure 2.1: The error-free state of the program is saved at certain points (checkpoints). Upon the detection of an error, the execution rolls back to the last error-free state.

*i.* the transient errors are instant errors which occur for very small period of time and *ii.* the probability of two errors occuring at the same time in the two replicas is extremely small. In addition, the probability is even smaller that the two errors occurring on the two replicas produce the same output. Therefore, dual-modular error detection is preferred for the detection of transient errors.

In Dual-Modular Redundancy (DMR), the two replicas execute the same piece of code. The checks validate the correctness of the program's execution by comparing the outputs of the two replicas (original and replicated output). If the outputs of the two replicas are identical, then there is no error and the program continues its execution normally. In the case of different outputs, an error has been detected and the execution of the program stops. In this way, dual-modular error detection can provide fail-stop capability to the system and prevent an error from propagating to the output of the

program. The state-of-the-art DMR techniques are discussed in Section 5.1.

After the detection of an error, the recovery mechanism is activated. The error detection and the recovery mechanisms are usually independent. The most common recovery mechanism is checkpointing which can be implemented at different granularity. For instance, IBM's G4 [86] checkpoints each instruction. In this architecture, the execution units are replicated and the outcome of the two units are compared. If there is no error, the new state of the CPU is saved. In case of an error, the execution returns to the last state of the CPU. Figure 2.1 shows checkpointing at coarser granularity (e.g., checkpointing every one thousand instructions [85]). In this case, both memory and architecture states of the program are periodically saved. The checks verify the execution of the program and checkpointing mechanism saves the error-free state. In the presence of an error, the execution rolls back to the last checkpoint (last error-free state).

In Triple-Modular Redundancy (TMR), the error detection and recovery is done at the same time. The three outputs of the program are checked using a majority voting process. In the presence of an error in one of the three replicas, the erroneous value is the minority (one out of three) and it is discarded. The correct value is the only one that propagates to the rest of the execution. TMR is very demanding on hardware resources. For this reason, it is less appealing than DMR with checkpointing which can be performed less frequently.

### 2.2.2 Instruction-level Error Detection

In this thesis, we study instruction-level error detection where instructions of the program are replicated and additional check instructions are introduced. All original, replicated and checking code is generated by the compiler and is placed on the same thread. Every check compares the outcome of the original and the replicated code using a compare (CMP) instruction. If the check succeeds, then the code continues executing (no jump), otherwise the control jumps (JMP) to the appropriate error handling routine. Figure 2.2 shows how the original code is transformed to the error detection code. More details about the error detection algorithm are given in Section 2.4.1.

In Figure 2.2, it is shown that the replicated code and the checks significantly increase the code size of the program. The replicated code has smaller impact on performance than the checks. The replicated code and the original code can be executed in parallel since they do not have any dependence. Wide-issue machines can run some

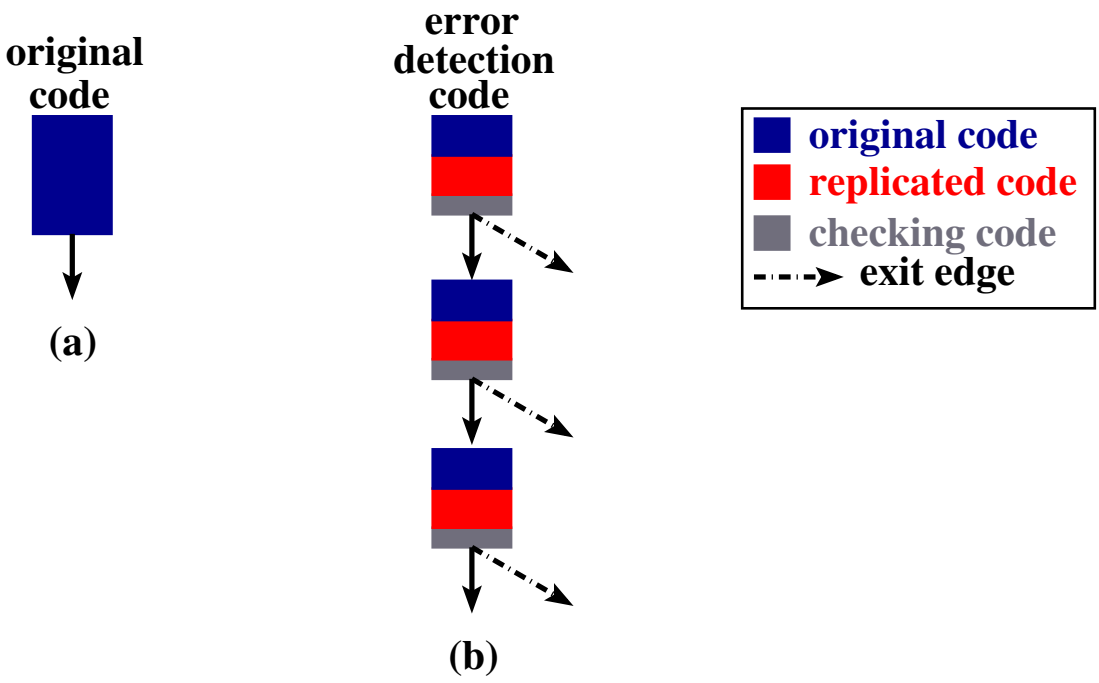


Figure 2.2: Code before (a) and after (b) instruction-level error detection (state-of-the-art methodology).

of the replicated instructions in parallel with the original ones. Hence, the overhead of the replicated code can be partially reduced. However, the overhead of checks cannot be hidden. The checks are compare and jump instructions. The latter ones are very expensive instructions since they limit the efficiency of compiler and hardware optimizations. The jump instructions prohibit the compiler from applying aggressive code motion optimizations. In addition, the effectiveness of a dynamic scheduler is also affected by jump instructions, since the number of outstanding branches is limited. The state-of-the-art instruction-level error detection methodology inserts branches every few instructions (more details in Section 3.1.1). The speculation window is not large enough to handle so many branches. Consequently, the jump instructions tend to prevent optimizations from extracting enough ILP and make the execution sequential. The reduction of the overhead of the replicated and checking code is the main target of the current state-of-the-art instruction-level error detection techniques.

The existing instruction-level error detection techniques are summarized in Table 2.1. EDDI [60] introduced instruction-level error detection. In EDDI, all the instructions of the program, apart from the branches, are replicated. Both the original and the replicated code read/write from/to memory. As a result, the memory traffic becomes dominant and is the main slowdown factor of this technique. The overall performance

| Approach | Technique | Protect Processor | Protect Memory | Performance Overhead | Main Memory Usage |
|---|---|---|---|---|---|
| Thread-local Replication | EDDI [60] | Most | All | 62% | 2x |
| | SWIFT [70] | Most | None | 41% | 1x |
| | Shoestring [25] | Most | None | 15.8%-30.4% | 1x |
| Redundant Multi-threading | SRMT [91] | Most | None | 400% | 1x |
| | DAFT [101] | Most | None | 38% | 1x |

Table 2.1: Comparison of the state-of-the-art instruction-level error detection techniques.

overhead is 1.62x compared to the code without error detection. The technique was evaluated on media and integer workloads. SWIFT [70] decreases this overhead by reducing the memory traffic. It achieves this by allowing only the original code to store data in memory. In this way, SWIFT reduces the error detection overhead down to 1.41x.

Another way to reduce the error detection overhead is to reduce the size of replicated code. This is the main target of symptom-based error detection (more details in Section 5.2). Shoestring [25] implements symptom-based error detection at instruction-level. The main idea is that some errors manifest as exceptions. For example, the operating system will raise an exception if a store instruction tries to write to an invalid memory address. Therefore, a transient error in some instructions of the program will result in an exception which can be captured by a specialized exception handler. As a result, any corruption on these instructions can be detected by the operating system. Hence, it is not necessary to replicate these instructions. Shoestring presents a technique that is based on static analysis to classify the instructions in three categories: i. safe instructions that do not need replication, ii. symptom-generating instructions (they do not need replication as well) and iii. vulnerable instructions that corrupt the output of the program and need replication. In this way, Shoestring reduces the error detection overhead which ranges from 1.15x to 1.30x (on a simulator). This performance gain comes with a small degradation on fault-coverage.

The techniques that we described so far replicate the instructions of the program locally in each thread (thread-local error detection). In redundant multi-threading, the original code is executed on the main thread and the replicated code and the checks

are executed in the checker thread. The main thread sends the checking values to the checker thread which compares these values with the ones that it produces itself. In the presence of an error, the execution of both threads is stopped. In addition, the checker thread does not have access to memory. Hence, the main thread passes the values that it loads to the checker thread. SRMT [91] introduced redundant multi-threading at instruction level. The SRMT scheme was tested on both SMT (simultaneous multi-threading) and CMP (chip multi-processor) architectures. It was shown that the SMT was slower since the main and checker thread share the same resources. The frequent communication and synchronization between the original and the checker thread becomes a performance bottleneck for SRMT (4x slowdown). In DAFT [101], the error detection overhead is significantly reduced (1.38x) by decoupling the execution of the original thread from the checker thread.

## 2.3 Fault Coverage

### 2.3.1 Sphere of Replication

According to the definition in [68], the sphere of replication indicates the components which are protected by each error detection scheme. Components inside the sphere of replication are protected against transient errors by redundant execution which can be either in time or space. On the other hand, components outside the sphere of replication must be protected by other techniques, such as ECC, parity checking etc. The values that leave the sphere should be checked. This is important to guarantee that erroneous values do not escape the sphere of replication.

In instruction-level error detection, the sphere of replication is often limited to the processor only as it is shown in Figure 2.3. The memory hierarchy is outside the sphere of replication because it is assumed that it has its own mechanisms (e.g., ECC, parity checking) to tolerate transient errors. Therefore, the data that are sent outside of the sphere of replication (to the memory) should be checked. In addition, all the data that enter the sphere of replication should be replicated. Therefore, the data that are loaded from memory are replicated. This is done in one of two ways:

i. The load instructions are replicated. In thread-local error detection techniques, this option is preferred even though the duplication of load instruction slightly increases the memory traffic and the overall error detection overhead. On the other hand, redundant multi-threading techniques might face coherency problems.

The checker thread might load a value that has been changed by the main thread. This will create a false-positive error detection.

ii. The original code (or the main thread) only loads data from the memory. Thread-local error detection methodologies emit a copy instruction that replicates the value that is loaded from memory by the original code. In redundant multi-threading, the main thread sends the value that it loads to the checker thread. This approach is usually preferred by redundant multi-threading in order to avoid the false positives.

Depending on the design specifications, the load instructions might be replicated or not. In our schemes, the load instructions are replicated. The following types of instructions are normally not replicated in order to reduce the overhead of the error detection code:

i. *Store instructions*. As we mentioned in Section 2.2.2, SWIFT [70] does not replicate the store instructions in order to reduce the error detection overhead of EDDI [60]. The replication of store instructions increases memory usage. For each memory location of the original code, a corresponding shadow memory location is needed for the replicated code. This replication increases hardware cost and slows down the program since the cache misses are increased and the memory traffic is more intense. For this reason, it is common practice [25][60][70][91][101] not to replicate the store instructions.

ii. *Control-flow instructions* (e.g., branches, function calls). The replication of control-flow instructions complicates the control-flow graph and makes optimizations harder to implement. In addition, the proposed schemes in [59][70] to protect the control-flow against transient errors, have very small impact on performance. For this reason, it is common practice [25][60][70] not to replicate the control-flow instructions. The control-flow is only followed by the original code and a signature-based error detection technique (which will be described later in this section) checks its correctness.

At this point, it has to be mentioned that the source code of pre-compiled libraries is not available when the error detection code is emitted. For this reason, library calls are outside of the sphere of replication. If the source code of the libraries is available, then it can be compiled with an instruction-level error detection mechanism so as to be protected against transient errors.

The checks are emitted before non-replicated instructions. Hence, checks are emitted before store and control-flow instructions. In the first case, it is guaranteed that error-free data leave the sphere of replication. In other words, we make sure that we send correct data to the memory and the output of the program. In the second case, the checks validate the correctness of the values of conditional jumps. In this way, it is guaranteed that the jump will take the correct path.

However, this check cannot guarantee that the correct path will be actually followed. A transient error might occur on the conditional jump after the check. As a result, the control-flow might be transferred to a wrong basic-block. Therefore, a mechanism that checks the transfers between the basic-blocks is needed. A signature-based mechanism has been proposed to fully protect control-flow [59]. A signature is assigned to each basic-block. In addition, a special register keeps the signature of the currently executing block. In the beginning of each basic-block, the value of the special register is XOR'ed with a constant. In this way, the signature of the previous basic-block is transformed to the signature of the current basic-block. Now, the special register (that contains a new signature) is compared with the current basic-block's signature. In case the comparison fails, the control transfer is invalid. Still, this is not enough since two basic-blocks that jump to the same basic-block, will have the same signature. To avoid this, a run-time signature is used. Thus, in the beginning of each basic-block, the run-time signature, the static signature and the special register are XOR'ed.

In [70], they showed that the overhead of control-flow error detection is orders of magnitude smaller than the overhead of the rest of the error detection. For simplicity, the proposed techniques do not implement control-flow error detection. This could be added as an extra feature without largely affecting the performance results since it is orthogonal to the proposed techniques.

### 2.3.2 Undetected Errors

Instruction-level error detection has some limitations which allow some errors to escape the sphere of replication undetected (as discussed in [70][69]). The following errors corrupt the output of the program:

- *Errors between validation and use:* The check validates the values of the non-replicated instructions. For this reason, they are emitted before the non-replicated instructions. The instruction scheduler shuffles the instructions in order to op-

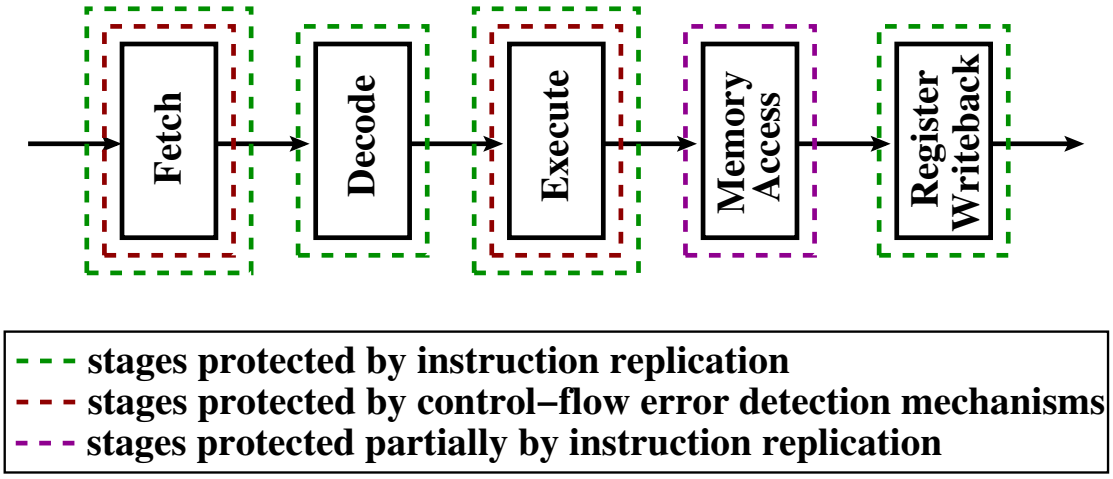


Figure 2.3: The stages of the processor pipeline that are protected by the sphere of replication of instruction-level error detection.

timally use the hardware resources (but with the restriction of not moving the check to after the non-replicated instruction). As a result, a check might end up being executed much earlier than the non-replicated instruction. Meanwhile, a transient error might occur that changes the stored value (e.g., in the register-file) before it is consumed. Therefore, there is a chance a store instruction will write wrong data to a memory address or write the data to a wrong address. In the latter case, the operating system might capture the error and prevent corrupted data from propagating to the output of the program. Unfortunately, incorrect values or addresses used in a store instruction will often not be detected.

- *Errors in the opcode of instructions:* A transient error can alter the opcode of an instruction. In most cases, this will be detected by the checks. However, if an instruction is changed to a store instruction, then the checks might not be executed before the store takes place, thus not being able to prevent the erroneous execution. The invalid store instruction will corrupt the memory unless the operating system raises an exception.

- *Control-flow errors:* For simplicity, we do not implement the control-flow error detection. We only add checks before control-flow instructions. Therefore, the correctness of the data are checked, but not the actual transfer between the basic-blocks. The proposed techniques (Chapters 3 and 4) are partially protected against control-flow errors. The control-flow error detection (as it is previously described in Section 2.3.1) is orthogonal to instruction replication and it can be

added easily with minimal overhead. The signature-based mechanism adds few extra checks whose overhead can be reduced by the DRIFT mechanism (Chapter 3).

- *Micro-architectural state:* Instruction-level error detection cannot capture all the errors that occur at micro-architectural level. In [11], the authors show such an example. An error in the control logic can mark an instruction that is ready to be executed as stalled. Since the instruction has initially been marked as ready for execution, there is no way to recover from stall. Consequently, the instruction will never be executed and the application will never finish.

To summarize, Figure 2.3 shows the pipeline stages that are protected by the sphere of replication of instruction-level error detection. For the full protection of the fetch stage both instruction replication and control-flow error detection are needed. For example, if an error results in fetching the wrong instruction, then the check that will compare the output of this instruction with the output of the corresponding replicated instruction will detect the error. On the other hand, if an error diverts the execution of the program to the wrong basic-block, then this error can only be detected by a control-flow error detection mechanism (that was described in Section 2.3.1).

Instruction replication is enough to protect the decode stage from transient errors. For example, if an addition is decoded as a multiplication, then the two replicas will produce different outputs. The checks will eventually detect the mismatch of the two outputs. As we mentioned earlier, there is a small chance of a non-store instruction being decoded erroneously as a store instruction. In the best case, the error will be captured by the check before the execution of the store instruction. In addition, the register read of the decode stage is also protected by instruction replication. Each replica performs register read and only one of them can have the erroneous value (due to either reading the wrong register or reading a wrong value from the correct register). The wrong value is detected by a check later in the program.

The execution stage is protected by both instruction replication and control-flow error detection. The replicated instructions can run either on the same functional unit or a different one. However, the jumps are not replicated and the branch unit can only be protected by control-flow error detection. In the memory access stage, the load instructions are protected if they are replicated. If an erroneous value is loaded, then this error will propagate to the code and it will be detected by a check. On the other hand, store instructions are not fully protected by recent, high-performance instruction-

level error detection. The write-back stage is protected by instruction replication for the majority of transient errors. If a value is erroneously written to the register-file, then this value will propagate to the code and it will be captured by a check.

## 2.4 Instruction-level Error Detection Algorithm

### 2.4.1 SWIFT Algorithm

The proposed mechanisms (Chapters 3 and 4) are compared against the state-of-the art instruction-level error detection technique which is SWIFT [70]. The main steps of the SWIFT algorithm are described in the flow-chart of Figure 2.4. During the instruction replication phase, the algorithm traverses all the instructions of each function of the program and it finds the instructions that can be replicated. For each replicable instruction, an exact copy is emitted before it. After the replication of all the replicable instructions, the checks are injected. In SWIFT, the checks are emitted before the non-replicated instructions. For this reason, the algorithm is now looking for the non-replicated instructions. For each one of them, it emits a check before them.

### 2.4.2 SWIFT Algorithm Example

Figure 2.5 shows how the original code is transformed by the SWIFT algorithm. In Figure 2.5.a, the original code is given. Instructions 1 and 4 calculate the data that are used by store (instruction 3) and function foo (instruction 5), respectively. Instruction 2 calculates the address of the store instruction. The store instruction and the function call are the non-replicated instructions of this piece of code (as we mentioned in Section 2.3.1).

Firstly, instruction replication takes place. The SWIFT algorithm emits a copy of the original instructions that can be replicated (instructions 2, 4 and 11 in Figure 2.5.b). The replicated instructions (1, 3 and 10 in Figure 2.5.b) are placed before their original instructions (2, 4 and 11 in Figure 2.5.a). Next, the checks are emitted. The algorithm finds the non-replicated instructions (9 and 14 in Figure 2.5.b). For each of the data that are read by a non-replicated instruction, the algorithm emits a check (compare and jump instructions) before the non-replicated instruction. For a store (instruction 9 in Figure 2.5.b), two checks should be emitted: the first one checks the data (instructions 5 and 6 in Figure 2.5.b) and the other one validates the address (instructions 7 and 8 in Figure 2.5.b).

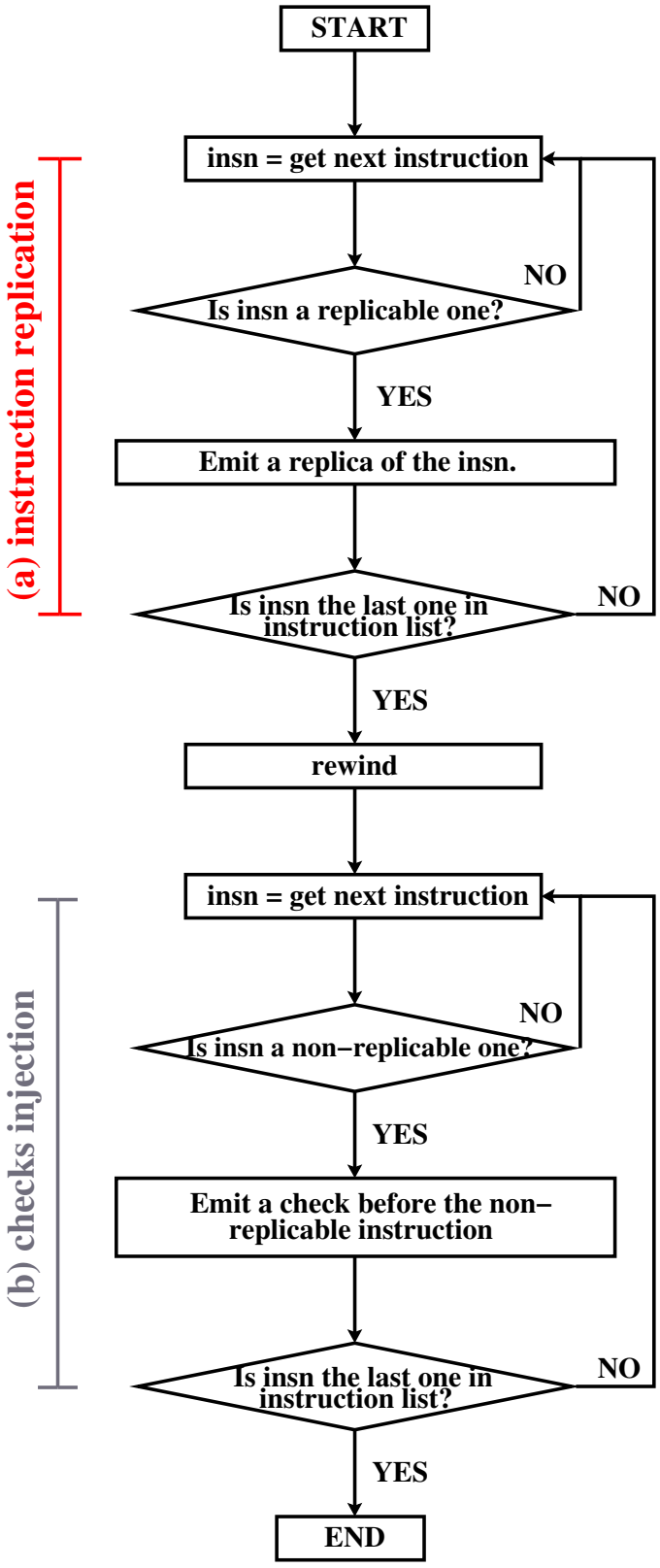


Figure 2.4: SWIFT [70] algorithm flow-chart consists of two phases: (a) the first one describes the replication of the instructions, (b) the second shows the injection of the checks before non-replicated instructions.

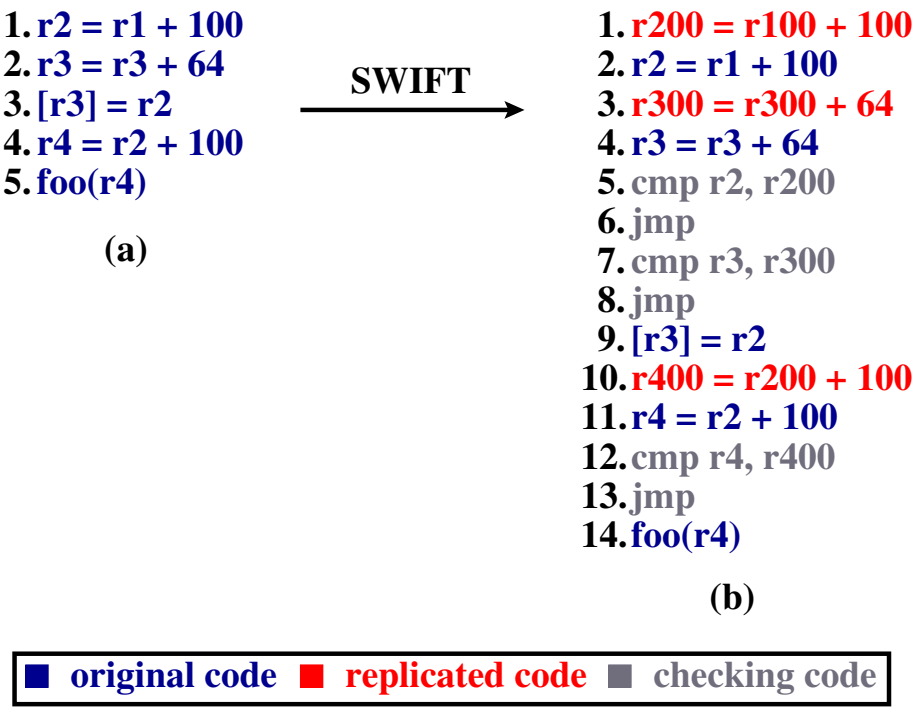


Figure 2.5: The code before (a) and after SWIFT algorithm (b).

For simplicity, in SWIFT [70], the arguments of each function are checked before the function call. This guarantees the correctness of the data that are sent to the function. In the function, the data are replicated and checked. In this way, the scheme makes sure that correct data are returned to the rest of the program. In Figure 2.5.b, function foo (instruction 14) has only one argument. Thus, one check (instructions 12 and 13) is emitted before the function call. A transient error that occurs between the check and the function call, can alter the value of one of the arguments of the function call. Since the data are replicated in the beginning of the function, both the original and the replicated instructions will have the same faulty data. As a result, the transient error cannot be detected. This could be fixed by changing the calling convention of each function. In this way, the original and replicated arguments will be passed in a function call. In case of an error, the check can detect the error because the original and the replicated argument will have different values. Similarly, the function can return multiple values if we change the calling convention. Instead, the SWIFT algorithm inserts checks before returning the data. Similar to SWIFT, the proposed techniques do not change the calling convention of the functions, but they insert checks before function calls and return instructions.

## 2.5 Fault Coverage Evaluation

### 2.5.1 Fault Model

As it was mentioned in Section 2.1, the proposed techniques, –DRIFT (Chapter 3) and CASTED (Chapter 4)– focus on the detection of transient errors. Single Event Upset (SEU) fault model is widely used for testing the efficiency of the error detection methodologies. This model assumes that only one error event occurs during one execution of the program. The event is temporal and it does not permanently damage any transistor's functionality. This event results in a bit flip. Thus, the SEU model simulates the occurrence of most types of transient errors.

### 2.5.2 Fault Injection

Similar to SWIFT [70], the fault-coverage evaluation is done by using Monte Carlo simulations. The SKI IA-64 simulator [2] was modified to inject errors at the output registers of instructions. It is common practice ([17],[25],[70],[91],[101]) to assume that an error in any functional unit (e.g., ALU) or any structure will eventually propagate to the output register of the instruction. Therefore, it is enough to bit flip the output register of an instruction in order to simulate a transient error. In our evaluation, the general purpose and predicate registers are injected with errors.

The Monte Carlo simulations run in the following steps:

i. The procedure starts with each binary being profiled in order to count the number of dynamic instructions.

ii. Next, a dynamic instruction is randomly selected.

iii. Then, a random bit of the register output is picked and is flipped.

iv. Finally, the modified binary is executed.

v. Steps ii to iv are repeated 300 times[1] for each configuration and each benchmark.

The binaries that support error detection are much larger (2.3x larger on average) than the original ones (Figure 3.12). Therefore, the probability of an error is bigger and the soft error rate increases. A fair comparison between the original code and the

[1] This number of repetitions is suggested in SWIFT [70] and it is considered adequate for the fault-coverage.

error detection code requires keeping the error rate fixed [70]. Thus, the error detection codes are injected with one error per the number of dynamic instructions of the original binary.

Finally, it has to be mentioned that we also inject errors in instructions which are out of the sphere of replication. Therefore, we also inject errors on store instructions, libraries etc. This approach gives a realistic indication of the fault-coverage of the proposed techniques (DRIFT (Chapter 3) and CASTED (Chapter 4)).

### 2.5.3 Error Classification

The output of each Monte Carlo trial is classified into one of the following five categories:

- *Benign Errors* (aka masked errors) are the errors that do not affect program's output and the program produces the same output and exit code as the error-free execution. For example, if the result of an ALU unit is used by a shift operation, the corrupted bit might be shifted away. Therefore, the corrupted value will never reach the output of the program. The branch predictor is a structure that always produces benign errors. An error in the branch predictor will result in a mis-prediction and the re-execution of the branch.

- The errors that the error detection algorithms successfully detect, are classified as *Detected*.

- Some errors manifest themselves as *Exceptions*. For example, a division by zero will raise an exception. In addition, the operating system raises an exception in case a store instruction tries to write data to an invalid address of the program. In [101], the authors propose a customized exception handler which can catch these errors. For this reason, for our purposes, exceptions are also considered detected errors.

- The errors that propagate to the output of the program, are called *data corruption*. As it was mentioned in Section 2.3.2, there are some errors that cannot be detected by the proposed techniques and they corrupt the output of the program.

- Finally, some errors result in infinite execution of the program. These errors are detected by our simulator and we name them *Time-out Errors* since we use a time-out script to stop the program.

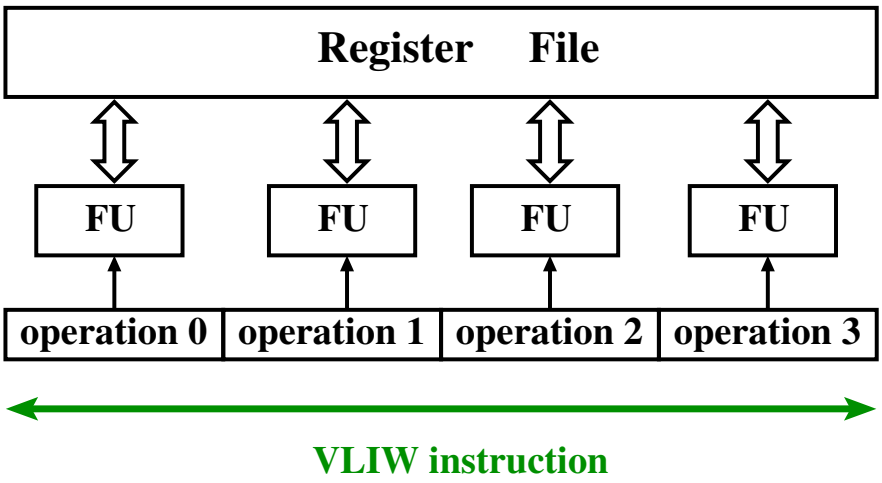


Figure 2.6: The VLIW architecture.

The first three categories do not corrupt the output of the program since they are either detected or swept away. Thus, data corruption and time-out errors are the critical ones. In Sections 3.3.3 and 4.4.3, we will show the resilience and the vulnerability of the original code on bit-flips. We will also present the efficiency of the proposed methodologies to detect these errors and minimize the probability of data corruption.

## 2.6 Target Architectures

### 2.6.1 VLIW Machine Model

Very Long Instruction Word (VLIW) processors are statically scheduled processors with RISC-like instruction sets [26][33]. In comparison to dynamically scheduled superscalar processors, the VLIW uses less hardware components since some of the optimizations including instruction scheduling are done at compile-time in software. For this reason, the compiler becomes critical for VLIW processors. Some of the advantages of VLIW processors over superscalar processors are summarized in [27]:

i. They need less hardware on the chip (thus, a lower silicon cost) since scheduling and register renaming are done at compile-time. On the other hand, superscalar processors use special hardware structures (e.g., reservation tables, reorder buffer etc.).

ii. VLIWs can achieve faster clock since there is less to do at each cycle.

iii. In terms of power, VLIWs are more efficient since scheduling is done statically

and it does not require any hardware support.

iv. VLIWs can achieve higher ILP with appropriate compiler-generated schedule. On the contrary, large ILP comes with big hardware cost for dynamic scheduled processors. The additional hardware needed to rearrange computations tends to grow exponentially with the amount of ILP available.

Figure 2.6 shows the VLIW architecture from the compiler's viewpoint. All the functional units share the same register file with uniform access latency. Each functional unit can execute instructions of several types. The scheduler groups the individual instructions (operations) that can run in parallel into a large instruction (VLIW instruction). Each operation of the long instruction executes at a different functional unit. All the functional units execute the operations of the VLIW instruction in parallel and in lockstep. The more instructions are scheduled in parallel (more ILP), the faster the code runs. As a result, VLIW systems rely on an efficient compiler scheduler for performance.

VLIW architectures are widely used in embedded systems because they offer very good trade-off between performance and power consumption. Examples of these embedded processors are Qualcomm Hexagon [18], STMicroelectronics ST200 family [24], Texas Instruments TMS320 C6000 series [37] and Fujitsu FR-v [89]. A VLIW-like architecture with many dynamic hardware additions for run-time optimizations is also used in Itanium 2 servers [49][78]. In 2013, Intel announced Kittson which is the latest successor of Itanium 2. Finally, AMD's GPUs are based on VLIW architecture [15].

### 2.6.2 Tightly-coupled Cores

Tightly-coupled cores look like Figure 2.7. These cores communicate efficiently with very small latency (typically only a few cycles). The design is scalable to large number of cores as the communication latencies are exposed to the programmers. Examples of such architecture are the wide-issue scalable clustered architectures such as VLIW clusters [23], RAW [90] and VOLTRON [102]. In this work, we use a clustered VLIW architecture. It differs from a traditional monolithic VLIW design in that critical resources (e.g., the register-file) are partitioned into small parts. Each part along with other resources (e.g., functional units) are tightly connected together and form a cluster. Within a cluster the data transfers are fast and energy efficient. Clusters are tightly-coupled and each cluster can access the other cluster's register file but this has

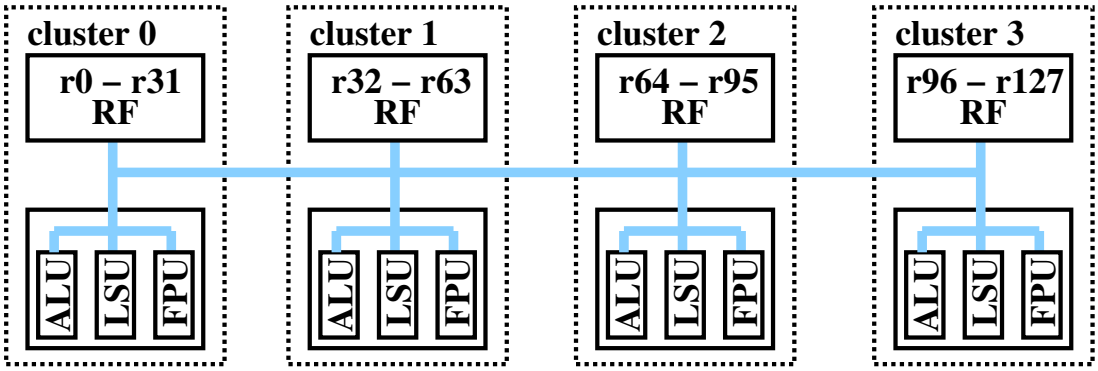


Figure 2.7: Clustered VLIW architecture with 4 clusters where cores in the same cluster share the same register file (RF).

an increased latency (*inter-core communication latency*) since it has to go through the interconnect.

Clustered architectures are proposed to improve the scaling in large issue widths without incurring clock frequency reductions. The complexity of the hardware design does not scale well to large issue widths [16]. The larger the issue width is, the more ports should be added to the register-file. But, the number of ports are limited and more ports lead to the increase in the clock cycle. To make the VLIW more scalable, the solution is to partition the register-file and the functional units in smaller private groups.

# Chapter 3

# DRIFT: Decoupled compileR-based Instruction-level Fault Tolerance

This chapter presents the DRIFT technique which reduces the error detection overhead of the state-of-the-art (SWIFT) without degrading the fault-coverage. DRIFT achieves this by *decoupling* the execution of the original and replicated code from the checking code. Recall that the checks are compare and jump instructions. The latter ones become a performance bottleneck because they tend to make the code sequential and prohibit the compiler from performing aggressive instruction scheduling optimizations. DRIFT solves this problem by relaxing the execution of the checks. In this way, DRIFT generates scheduler-friendly code with more ILP. As a result, DRIFT outperforms SWIFT by up to 29.7% and its performance overhead on the original code is reduced down to 1.29× (on average).

## 3.1 Motivation

### 3.1.1 Limitations of Instruction-level Error Detection

Figure 3.1 summarizes the main concept of instruction-level error detection as it is described in Sections 2.2.2 and 2.4. The frequent checks (compare and jump instructions) break the original code into a sequence of small basic-blocks with two outgoing edges each. In Figure 3.1, the original basic-block BB1 is split into three smaller ones: BB1a, BB1b and BB1c (Figure 3.1.b). The performance bottleneck of this scheme is shown up as what we call **basic-block fragmentation**.

This problem has two main factors:

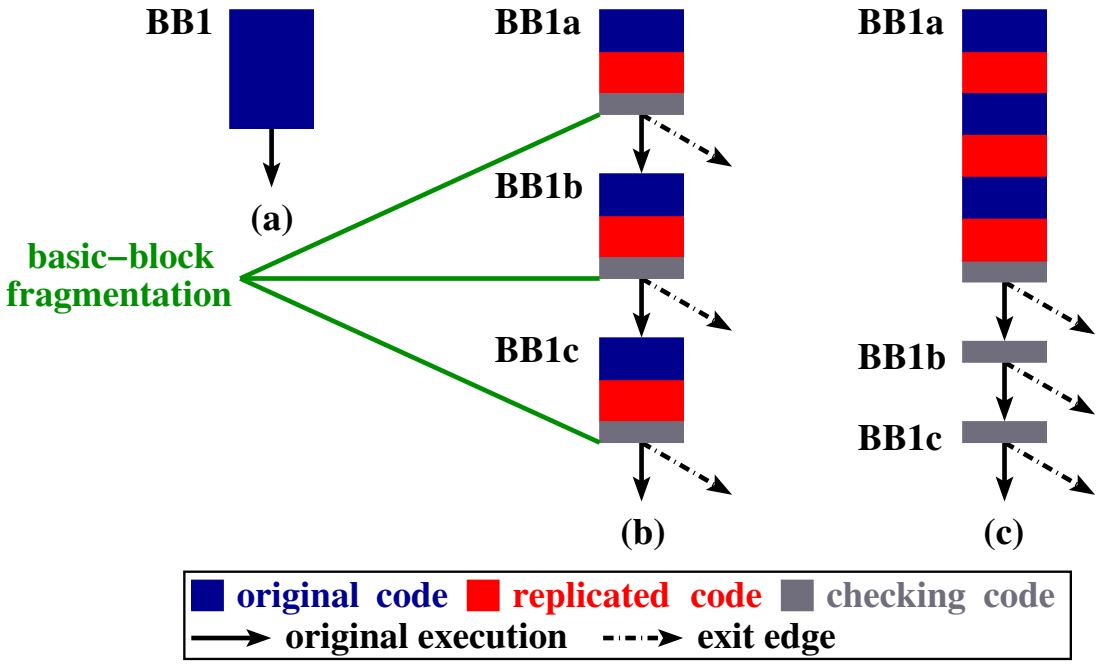


Figure 3.1: The basic-block fragmentation of the synchronized scheme. (a) Code without error detection (b) Synchronized thread-local instruction-level error detection (b) Decoupled thread-local instruction-level error detection.

- The new basic-blocks (BB1a, BB1b and BB1c in Figure 3.1.b) usually have a small number of instructions. Therefore, a basic-block scheduler does not have enough instructions to improve instruction-level parallelism (ILP).

- The checks create new edges in the control-flow. For each check, an exit edge is added. The new edge connects the current basic-block with the basic-block that invokes the error handling routine. The complex control-flow due to the checks acts as a scheduling barrier for the instruction scheduling optimization (e.g., trace scheduling). Even with a speculative scheduler that schedules regions of multiple basic-blocks [31][36][44][46][54][53], the control edges limit the scheduler's ability to hoist instructions across the basic-blocks. As a result, the scheduler cannot extract adequate amounts of ILP. Any state-of-the-art region-based instruction scheduler has some *limitations* in hoisting instructions across basic-blocks:

  - It cannot hoist instructions with side-effects over branches since this can break the program semantics. This restricts the hoisting of system calls, and store instructions [36, 44].

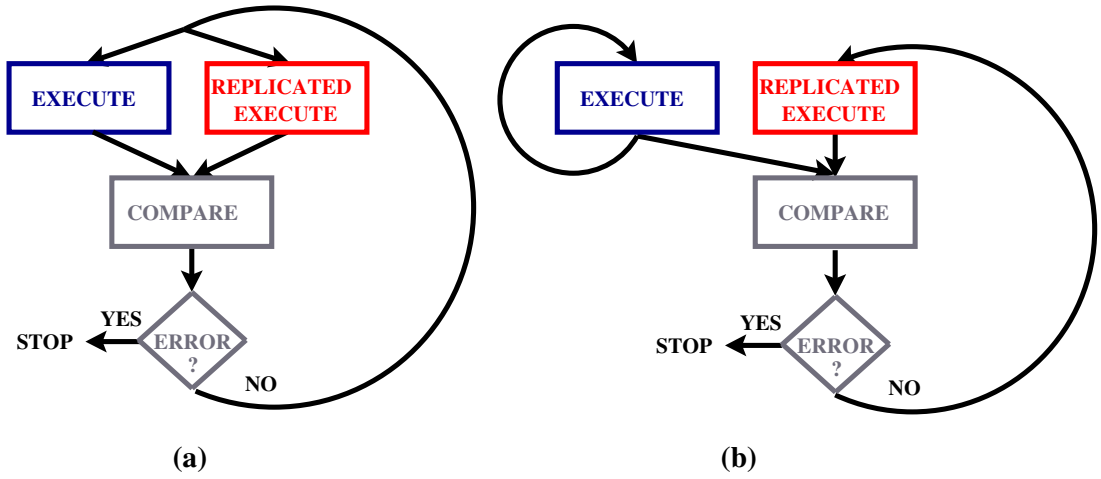


Figure 3.2: The code execution of synchronized (a) and decoupled (b) error detection schemes.

– If there is no hardware support for deferring exceptions then potentially faulting instructions such as loads and divisions cannot be hoisted either [45].

As a result, the scheduler generates poorly performing code with low ILP. For this reason, basic-block fragmentation becomes a performance bottleneck for instruction-level error detection. We will quantify this effect in Section 3.3.2.

### 3.1.2 Synchronized versus Decoupled Error Detection

In instruction-level error detection, the checks are synchronization points where the execution of the code is checked for errors. For this reason, this technique is called *synchronized error detection*. In the extreme case, the checks occur after the execution of every instruction. However, as we have already discussed, it suffices if they occur before every non-replicated instruction (i.e., an instruction whose effect escapes the sphere of replication). In Figure 3.2.a, it is shown that the original and the replicated code are executed synchronously. The execution of the program is interrupted by the checks. After confirming that the code is not affected by any error, the execution resumes. Therefore, the execution of the synchronized error detection follows the cycle *execute-check-confirm-execute*. In this way, the synchronized error detection guarantees fail-stop capability to the program.

In thread-local instruction-level fault tolerance, the execute-check-confirm-execute

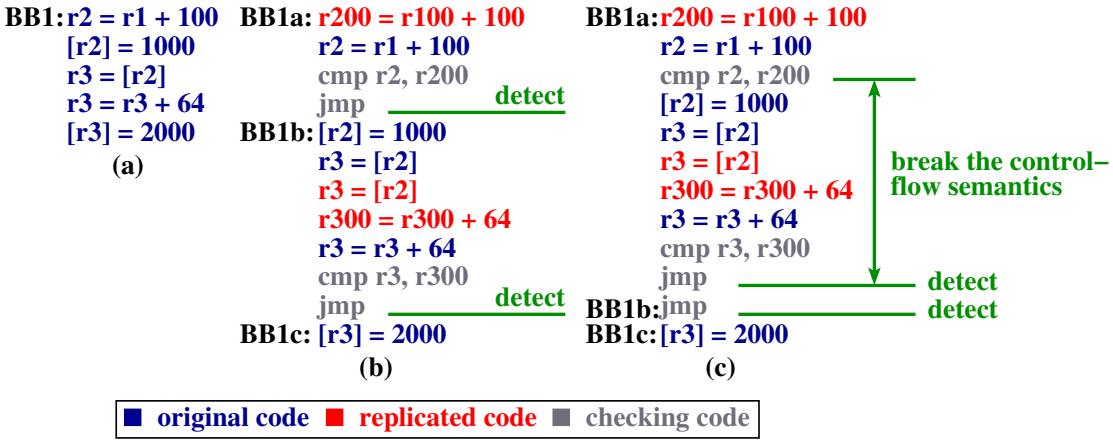


Figure 3.3: The original code (a) is transformed by synchronized (b) and decoupled (c) error detection schemes. The synchronized scheme cannot be optimized further because the compiler should respect the program semantics. The decoupled scheme breaks the control-flow semantics and optimizes the code.

cycle manifests itself as the basic-block fragmentation problem. The solution is to break this cycle by decoupling the execution of the original and replicated code from the checks (Figure 3.2.b). Decoupling reduces the impact of basic-block fragmentation as the checks are pushed off the critical path, leading to longer stretches of instructions to execute. This reduces the error detection overhead and boosts the performance of the program. Such performance improvement may come at the expense of reduced fault coverage since the program loses its fail-stop capability. However, as shown previously (e.g., [101]), and as our experiments will demonstrate, the impact on fault-coverage is often negligible.

## 3.2 DRIFT

In this thesis we propose DRIFT, an error detection scheme that addresses the shortcomings of the synchronized error detection scheme, as described earlier. DRIFT introduces decoupling in instruction-level error detection by relaxing the execution of the checks. Instead of synchronizing before every non-replicated instruction through a check operation, DRIFT groups the checks and executes them later. In Figure 3.1.c, it is shown that the checks are grouped together and they are pushed to the end of the basic-block. In this way, DRIFT does not fragment the original basic-block (Figure 3.1.a) into smaller ones (Figure 3.1.b) and it enables the scheduler to do its job better.

DRIFT is based on three ideas:

i. *Optimized control-flow:* Modifying the control-flow of the application can enhance the ability of the instruction scheduler to optimize the code and reduce the impact of basic-block fragmentation. Since instruction schedulers are not as effective across basic-blocks as within basic-blocks, larger basic-blocks are better (higher ILP). This can be done by decoupling the execution of checks and by executing them later together as a group. By contrasting Figure 3.1.b versus Figure 3.1.c, we observe that DRIFT generates a much more instruction-scheduler friendly code than the synchronized scheme.

ii. *It is acceptable to break the semantics of the combined original and replicated code, as long as the semantics of the original code are respected.* The jump instructions due to checks force the compiler to apply conservative code motion optimizations (Section 3.1.1) because the scheduler should respect the program semantics. The compiler does not know that transient errors are exceptional events. Thus, it does not understand that these branches are not usually taken and the program mostly follows the error-free execution. This unawareness of normal compilers to the semantics of error detection code is the main reason why the compiler cannot automatically generate decoupled code (like the one DRIFT generates) out of the synchronized code. Therefore the code of Figure 3.1.c cannot have been generated by any compiler optimization. Breaking the semantics in a controlled way is required for modifying the code in such an aggressive way. Figure 3.3 shows how our scheme breaks the control-flow semantics of the code. In synchronized thread-local error detection, the errors are detected on time (Figure 3.3.b). However, the original execution is interrupted by the checks which have negative impact on execution time since the compiler cannot deal with the frequent checking code (the compiler generates code with poor ILP, Section 3.1.1). On the other hand, if we relax the execution of the checks (detection of the errors) by breaking the strict control-flow semantics, then the original execution can be decoupled from checking code. In this way, the compiler can extract more ILP and the code can run faster (Figure 3.3.c).

iii. *DRIFT's decoupled semantics have little to no effect on fault-coverage.* As shown in [101], modifying the semantics of the application with error detection support, such that the checks are decoupled from the execution, has a minimal impact on the effectiveness of error detection. This is because in the usual case, the increased delay between the error and its detection is not great enough to let the error propa-

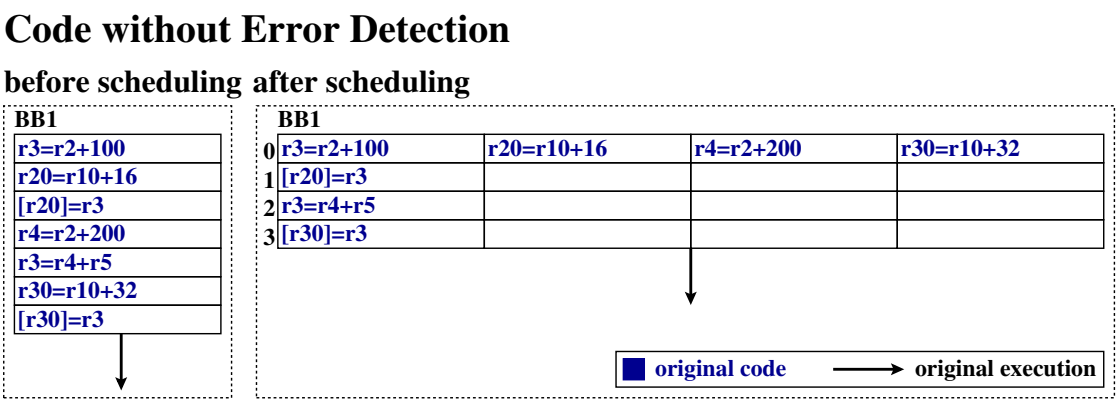


Figure 3.4: The code before and after instruction scheduling for the original code without error detection.

gate to the output (Figure 3.3.c). Moreover, it has been shown in [25][41][93] that a significant number of errors such as ISA-defined exceptions can be detected by the operating system. This is a fundamental feature of DRIFT, which guarantees its high fault-coverage despite the modified semantics that allow better performance.

### 3.2.1 DRIFT Motivating Example

The following example presents the original code in Figure 3.4 and the transformation of the code after the insertion of the error detection code from SWIFT and DRIFT (Figures 3.5 and 3.6 respectively). Each figure shows the code before instruction scheduling (left) and the scheduling table (right) of a hypothetical 4-issue machine which supports predication. The outcome of compare instructions is kept in predicate registers.

The following observations can be made:

- *The overhead of the replicated code is less significant than the jump instructions.* In Figure 3.4, it is shown how the original code is scheduled. The dependences between the instructions do not allow the compiler to fully benefit from the available hardware parallelism in the architecture. The empty slots are filled with the replicated and checking code as it is shown in Figure 3.5. In this way, the overhead of redundant code is partially hidden. Applications with low ILP can efficiently hide the error detection overhead in processors with even moderate amounts of parallelism.

- *Basic-Block Fragmentation:* The checks act as fragmentation points for the

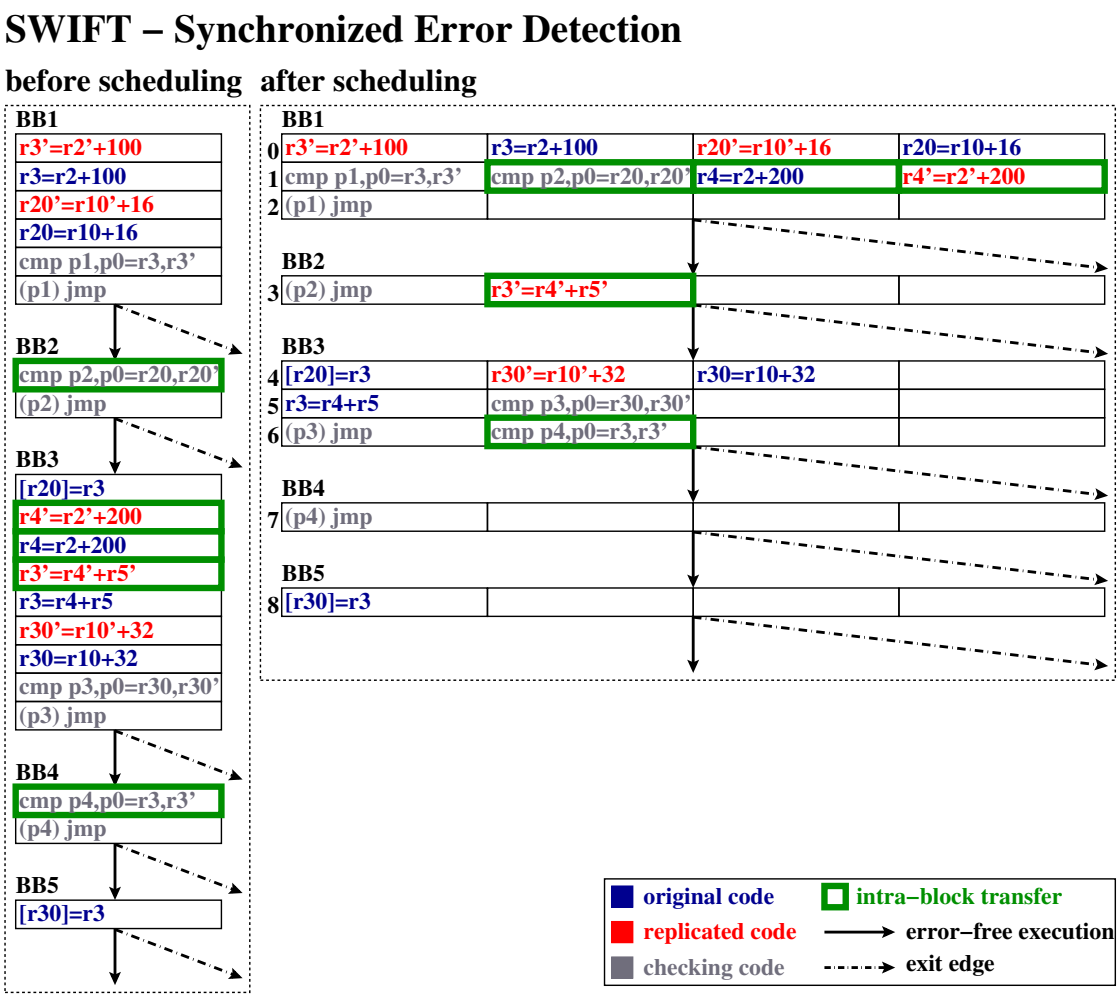


Figure 3.5: The code before and after instruction scheduling for the synchronized scheme (SWIFT).

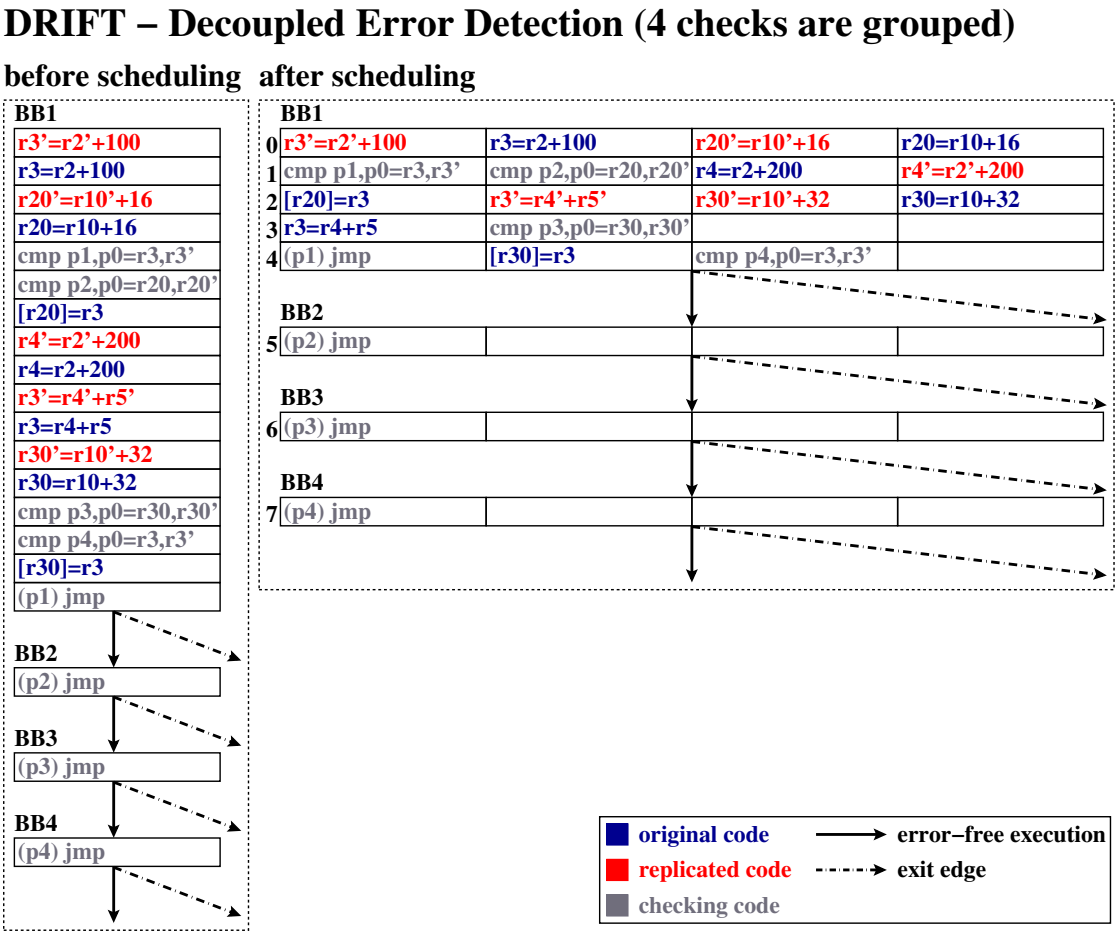


Figure 3.6: The code before and after instruction scheduling for the DRIFT scheme. In this example four checks are clustered together.

control-flow graph and they split the code into numerous basic-blocks. For example, the original code of Figure 3.4 is a single basic-block, but the error detection code of Figure 3.5 spans over 5 basic-blocks (BB1-BB5).

The key difference between the synchronized scheme (Figure 3.5) and the DRIFT scheme (Figure 3.6) is the amount of fragmentation of the basic-blocks. The synchronized case is the most fragmented one, as checks are regularly injected into the code (see Figure 3.5 left). On the other hand, DRIFT groups together multiple checks. In the example of Figure 3.6, it groups 4 checks together which are placed at the end of the basic-block.

- *Performance and Schedule:* The impact of basic-block fragmentation is shown at the left in Figures 3.5 and 3.6. In the synchronized case, the checks fragment the code (see Figure 3.5 left) and the instructions are isolated in small basic-blocks. DRIFT clusters the checks at the end of the basic-block. In this way, all the instructions remain in the original basic-block (see Figure 3.6 left).

  An intra-block scheduler schedules the instructions of each basic-block. Then, an inter-block scheduler hoists as many instructions as possible across the basic-blocks in order to improve the ILP. The inter-block transfers are marked with green. The synchronized scheme is fragmented as jump instructions introduce edges into the control-flow. These edges prohibit aggressive code hoisting in several cases. For example, “[r20]=r3” of BB3 cannot be hoisted into BB2 or BB1 as it has side-effects (writes to memory) and the compiler cannot guarantee that this instruction will be executed (the execution might follow another control path). For this reason, the compiler will break the program semantics if it hoists the store instruction in another basic-block. For the same reason, “r[30]=r3” of BB5 cannot be hoisted either. These factors prohibit the scheduler from taking advantage of the available ILP. The produced schedule is full of NOP instructions.

  Removing the code motion restrictions of the synchronized scheme, the schedule improves considerably. For example, in Figure 3.6, all instructions are within a single basic-block (BB1) which makes it straight-forward for any scheduler to parallelize. In this way, DRIFT produces scheduler-friendly code which can be more easily optimized. As it is shown in Figure 3.6, the scheduler exploits all the available ILP and it produces a compact schedule.

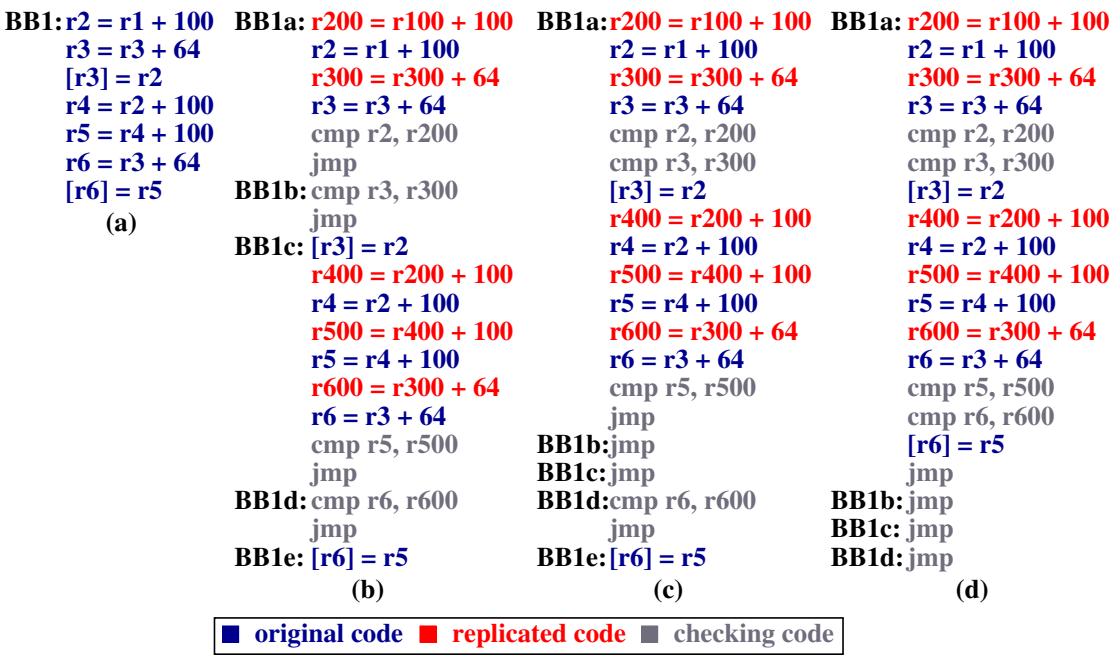


Figure 3.7: In this example, decouple factor 3 (c) and 4 (d) have similar impact on the original code (a). In this way, the decoupled scheme prevents basic-block fragmentation and creates large basic-blocks. On the contrary, the synchronized scheme (b) fragments the code and creates small basic-blocks with few instructions.

### 3.2.2 Decouple Factor

To better study the impact of decoupling on performance, we introduce the concept of *decouple factor*. This metric describes the number of checks that are clustered together. As explained in Section 3.1, DRIFT relaxes the execution of the checks by grouping them together. Each group of checks contains up to $N$ number of checks. We refer to this as decoupling $N$ checks or setting the *decouple factor* to $N$. Therefore, the decouple factor is a knob that controls the number of checks that are executed together in a group. As the decouple factor increases, more checks are grouped together, and the effect of basic-block fragmentation is reduced.

Figure 3.7 shows how decouple factor works. In Figure 3.7.c, the decouple factor is three. Thus, three checks are clustered together. Similar to synchronized error detection (Figure 3.7.b), the checks split the basic-block in five smaller basic-blocks. But, in Figure 3.7.c, the original basic-block has more instructions. In this example, decouple factor three and four (Figures 3.7.c and 3.7.d) have similar impact on basic-block fragmentation. Both configurations keep the majority (or all) the instructions in the original basic-block. Moreover, in this example, a decouple factor of two produces

almost the same code as the synchronized error detection technique. In general, the synchronized error detection is represented by a decouple factor of one.

Increasing the decouple factor has three side-effects:

i. For small values of the decouple factor, the program has similar (though slightly better) behavior to the synchronized error detection and suffers from basic-block fragmentation. As the decouple factor increases, more checks are clustered together giving the scheduler the freedom to schedule the instructions more efficiently and improve the ILP.

ii. Performance is not always improved as the decouple factor increases. On the contrary, there is a chance to degrade performance due to hardware congestion. For large values of the decouple factor, many checks are executed later in the code. Therefore, the distance between the definition and the use of a value increases as the decouple factor increases. For large values of the decouple factor, more values are kept in the register file for a longer period of time. This might lead to predicate register pressure which can cause performance degradation if it results in register spilling. Another factor that increases the probability of hardware congestion is the large number of consecutive compare instructions. In this case, there might not be enough units to deal with them in a timely manner. As a result, the compare instructions might stall until a unit is available.

iii. According to the value of the decouple factor, the checks might not be emitted before the non-replicated instructions. In Figure 3.7.d, the checks are emitted after all the instructions of the basic-block. Thus, if the check executes much later (large values of decouple factor), we slightly increase the risk of allowing erroneous data to propagate to memory and corrupt the output of the program.

As a result, it is not easy to predict which value of the decouple factor is the best. There is a trade-off between the number of checks that are decoupled, the hardware capacity and the fault-coverage. We explore the effect of the decouple factor on both performance and reliability in Section 3.3.

### 3.2.3 DRIFT Algorithm

The DRIFT algorithm operates in four steps:

i. *Code Replication:* For each basic-block (BB) of the function, the algorithm finds the instructions that should be replicated (DRIFT Algorithm line 16). For each

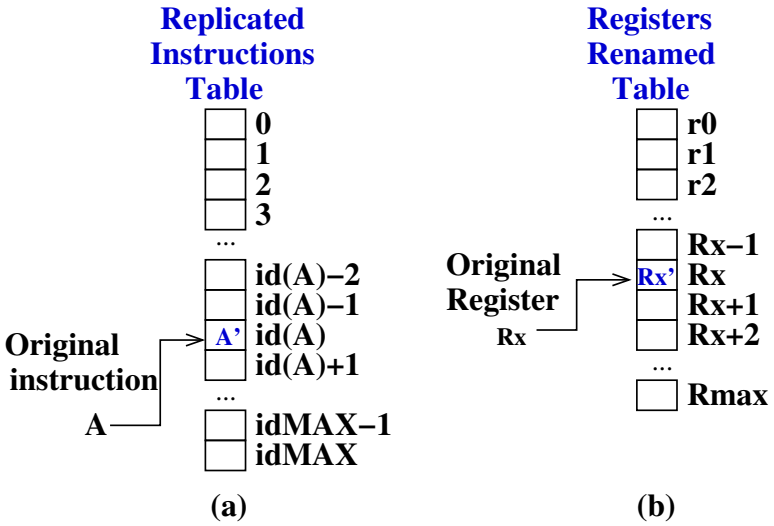


Figure 3.8: The unique ID of the original instruction is the element of Replicated Instructions Table (a) that keeps the unique ID of the corresponding replicated instruction. Similarly, the number of the original register is the element of Register Renamed Table (b) that keeps the corresponding renamed register.

one of them, it creates an exact copy of the original instruction (DRIFT Algorithm line 18). The new instruction (replicated) is emitted just before the original one (DRIFT Algorithm line 19). The replicated instruction is kept into the Replicated Instructions Table (Figure 3.8.a, DRIFT Algorithm line 20). Each original and replicated instruction has a unique ID [1]. The Replicated Instruction Table is a one-column array whose size is equal to the number of the original instructions of a function (each function is compiled separately). Each element of the array is indexed by the unique ID of the original instruction and each element keeps the RTL [2] of the corresponding replicated instruction. For example, the replicated instruction of the original instruction with unique ID 13 is the 13th element of the Replicated Instructions Table. The Replicated Instructions Table (Figure 3.8.a) is used by the code isolation step to recall the replicated instruction of the corresponding original one.

ii. *Code Isolation:* At the end of the previous phase, the two replicas share the same source and destination registers. To prevent the replicated instructions from writing to the registers of the original instructions, code isolation is needed. This is

[1] GCC assigns a unique ID (an integer number) to each instruction of the program.

[2] In the back-end of GCC, the instructions of the program are represented using the RTL format. RTL representation is similar to the instruction representation of the assembly.

done by register renaming (DRIFT Algorithm line 25). The algorithm iterates over all original instructions in the program (DRIFT Algorithm lines 27). For each original instruction that has a duplicate (DRIFT Algorithm lines 29-31), the algorithm retrieves the corresponding replicated instruction from the Replicated Instructions Table (Figure 3.8.a, DRIFT Algorithm line 33). Then, for each replicated instruction, the algorithm renames the register that is written (DRIFT Algorithm lines 36). This is done by generating a new pseudo register to the replicated instruction. DRIFT's pass is implemented before the instruction scheduler pass. At this stage, the code can use as many as registers as it needs (pseudo registers). Later, the register allocation pass maps pseudo registers to hard registers (architecture registers). In the next step of the algorithm, the uses of the renamed register are updated (DRIFT Algorithm line 37). The original and the replicated registers are added in Registers Renamed Table (Figure 3.8.b, DRIFT Algorithm line 38). Similarly to Replicated Instructions Table, the Registers Renamed Table is a single column array where each element of the array (indexed by the number of the original register) keeps the replicated register. For example, the 13th element of the Registers Renamed Table is the renamed register number corresponding to the original register 13. The Registers Renamed Table is used by next step to emit the checks.

iii. *Emit checks:* Next, the algorithm finds all the non-replicated instructions (DRIFT Algorithm line 49). For each non-replicated instruction, the algorithm finds the registers that the non-replicated instruction reads (DRIFT Algorithm line 51). For each one of these registers, the algorithm traverses the Registers Renamed Table (Figure 3.8.b) and finds the renamed register (DRIFT Algorithm line 53). For each pair of registers (original and renamed), a compare instruction is created (DRIFT Algorithm line 54). The compare instruction is emitted right before the non-replicated instruction (DRIFT Algorithm line 55). Afterwards, for the synchronized error detection technique, a jump instruction is emitted after the compare instruction and the control-flow is updated. The latter step is a very crucial one because it guarantees the correct execution for both the error-free and the erroneous execution. In the first case, the control-flow follows the original execution. Thus, a new edge between the current basic-block (where the check is) and the next basic-block is added. In the second case where an error occurs, the jump instruction diverts the execution to the basic-block (exit block) that invokes

the exception handler that manifests the error. As a result, another edge is added between the current basic-block and the exit block. This is the final step for the synchronized error detection scheme. On the other hand, DRIFT collects all the compare instructions of a basic-block into the vector CMP_VEC which is used in step 4 to perform the grouping of checks (DRIFT Algorithm line 56).

iv. *Decouple Checks:* Finally, decoupling takes place (DRIFT Algorithm line 62). In more details, the algorithm pops from the head of the CMP_VEC vector as many compare instructions as the value of the decouple factor (DRIFT Algorithm line 64). Next, the algorithm emits as many jump instructions as the number of decouple factor after the compare instruction which is popped last from the CMP_VEC vector (DRIFT Algorithm line 68 - 71). Then, the control-flow is updated (DRIFT Algorithm line 72). For example, in Figure 3.7.c, the decouple factor is three. Thus, three compare instructions are popped from CMP_VEC vector. The three jumps are emitted after the third compare instruction (Figure 3.7, cmp r5, r500). If the number of checks in a basic-block is smaller than or equal to the decouple factor, then the algorithm emits all the jump instructions at the end of the basic-block. Moreover, if the decouple factor is 1, then the algorithm emits all the jump instructions after each compare instruction as the synchronized scheme does.

DRIFT Algorithm

```
1 drift_main ()
2 {
3   for each BB in function
4       {
5        replicate_insns (BB)
6        register_rename (BB)
7        CMP_VEC = emit_compare_insns (CMP_VEC, BB)
8        emit_jump_insns (CMP_VEC, DECOUPLE_FACTOR, BB)
9       }
10 }
11 /*Emit replicated instructions*/
12 replicate_insns (BB)
13 {
14   for each INSN in BB
15       {
16        if INSN is not control-flow or store instruction
17           {
18            INSN_DUP = copy INSN
```

```
19              emit INSN_DUP before INSN
20              add INSN_DUP in the Replicated Instructions Table
21             }
22       }
23 }
24 /*Code isolation.*/
25 register_rename (BB)
26 {
27   for each INSN in BB instructions:
28       {
29        if INSN is an original instruction
30           {
31            if INSN has a replica
32               {
33                INSN_DUP = get the replica of INSN from the
                     ↪Replicated Instructions Table
34                if REG of INSN_DUP is written
35                   {
36                    RENAMED_REG = rename REG of INSN_DUP
37                    update the uses of RENAMED_REG
38                    add RENAMED_REG in the Registers Renamed
                         ↪Table
39                   }
40               }
41           }
42       }
43 }
44 /* Emit compare and jump instructions. */
45 emit_compare_insns (CMP_VEC, BB)
46 {
47   for each INSN in BB
48       {
49        if INSN is a non-replicated instruction
50           {
51            for each REG read by INSN
52                {
53                 RENAMED_REG = get renamed register from the
                       ↪Registers Renamed Table
54                 CMP_INSN = create compare instruction that
                       ↪compares REG and RENAMED_REG
55                 emit CMP_INSN before INSN
56                 push CMP_INSN in CMP_VEC vector
```

```
57                  }
58              }
59          }
60 }
61 /*Decouple checks.*/
62 emit_jump_insns (CMP_VEC, DECOUPLE_FACTOR, BB)
63 {
64   for i in DECOUPLE_FACTOR
65   {
66     CMP_INSN = pop compare instruction from CMP_VEC vector
67   }
68   for i in DECOUPLE_FACTOR
69   {
70     JMP_INSN = create a jump instruction
71     emit JMP_INSN after CMP_INSN
72     update control flow for JMP_INSN
73   }
74 }
```

## 3.3 Results and Analysis

### 3.3.1 Experimental Setup

We implemented our error detection scheme in a compiler pass in GCC-4.5.0 [1]. The DRIFT pass was placed just before the first instruction scheduling pass (Figure 3.9).

We evaluated our instruction-level error detection scheme using 9 benchmarks from the Mediabench II video [28] and the SPEC CINT2000 [34] benchmarks. These are the benchmarks that we managed to compile with our heavily modified compiler.

All benchmarks were compiled with -O2 optimizations enabled. To prevent optimizations such as Common Sub-expression Elimination (CSE) and Dead Code Elimination (DCE) from removing the replicated code, we disabled them at the *late* back-end stages of compilation, only for the error detection schemes (they are enabled in the code without error detection). This is common practice in instruction-level error detection schemes (e.g., SWIFT [70]). These optimizations are called several times in the intermediate representation and the back-end of GCC. The last stages of CSE and DCE do not affect the overall performance. They are mostly called to clean the code after instruction scheduling and register allocation. Our measurements show that the impact of the last stages of CSE and DCE on performance is negligible (1.5% in the

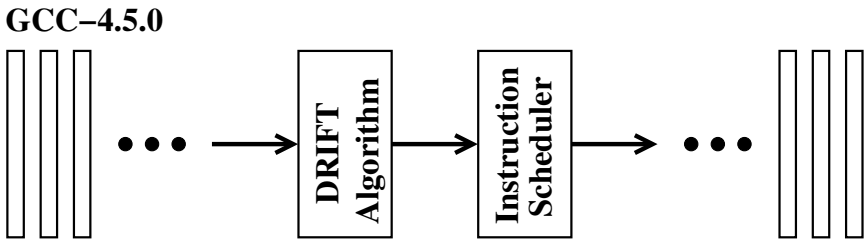


Figure 3.9: DRIFT algorithm pass is placed at the back-end of GCC-4.5.0 just before the instruction scheduler.

| Processor: Itanium2 | | | | |
|---|---|---|---|---|
| Issue width: | 6 | | | |
| Instruction Latencies: | same as Itanium2 [49] | | | |
| Register File: | 128GP, 128FL, 64PR | | | |
| Branch Prediction: | Perfect | | | |
| Cache: Levels 3 (same as Itanium2 [49]) | | | | |
| Levels : | L1 | L2 | L3 | Main |
| Size (Bytes): | 16K | 256K | 3M | ∞ |
| Block size (Bytes): | 64 | 128 | 128 | - |
| Associativity: | 4-way | 8-way | 12-way | - |
| Latency (cycles): | 1 | 5 | 12 | 150 |

Table 3.1: SKI IA64 simulator configuration.

worst case and 0.3% on average).

The performance evaluation was done on a DELL PowerEdge 3250 server with 2x1.4GHz Intel Itanium 2 processors. For the fault coverage evaluation, we used a modified SKI IA-64 simulator [2] (Table 3.1). The simulator is a cycle-accurate Itanium 2 simulator, modified to allow fault injection (Section 2.5.2).

### 3.3.2 Performance Evaluation

We evaluated our scheme by measuring:

i. *NOED* which is the code with no error detection,

ii. *SWIFT* which is the state-of-the-art synchronized thread-local error detection methodology [70]. For simplicity, SWIFT is usually implemented with branch checking

instead of control-flow checking [17][25]. These techniques have the same overhead. The only difference is that control-flow checking verifies the execution of a jump instruction. It should be noticed that data checking is orthogonal to control-flow checking. This means that control-flow checking can be plugged in the proposed technique as well without any performance degradation.

iii. *DRIFT* was implemented with various decouple factors (DEC-2, DEC-4, DEC-8, DEC-16, DEC-INF). For example, DEC-4 implies a decouple factor of four. DEC-INF implies an infinite decouple factor which means that all checks are placed at the end of the basic-block. A decouple factor of 1 is not measured because it is equivalent to SWIFT.

The results are shown in Figure 3.10 and Figure 3.11. Each row shows the results of a given benchmark. The results are the aggregate of several runs. We did not notice any significant variation. The first column shows the normalized execution time of all schemes. The execution time is normalized to NOED. The second column presents the percentage of basic-blocks that have a given number of checks. For example, in cjpeg, over 30% of the basic-blocks have 2 checks (checks2). This measurement is based on run-time information (we take into account the number of times each basic-block is executed at run-time). The number of checks usually relates to the basic-block size.

The results of the first column in Figure 3.10 and Figure 3.11 verify our assumption that basic-block fragmentation is a significant slow-down factor of the synchronized single-core error detection scheme (SWIFT). Both SWIFT and DRIFT were scheduled with the same state-of-the-art GCC region-based speculative scheduler. In the case of SWIFT, it is shown that the compiler cannot produce efficient code since the complicated control-flow acts as a barrier to code motion optimizations. On the other hand, DRIFT creates a scheduler-friendly code. As a result, the performance improvement of DRIFT over SWIFT is up to 29.7% (h263enc, DEC-4) and DRIFT manages to decrease its overhead over NOED down to 1.29×.

DRIFT's performance varies across benchmarks and it is largely affected by the check distribution. Benchmarks like cjpeg, h263dec, mpeg2dec, 175.vpr and 300.twolf have small number of checks per basic-block. Therefore, a decouple factor of 2 is enough to improve their performance. On the other hand, a larger decouple factor benefits the applications that contain many checks per basic-block (e.g., djpeg, h263enc and mpeg2enc).

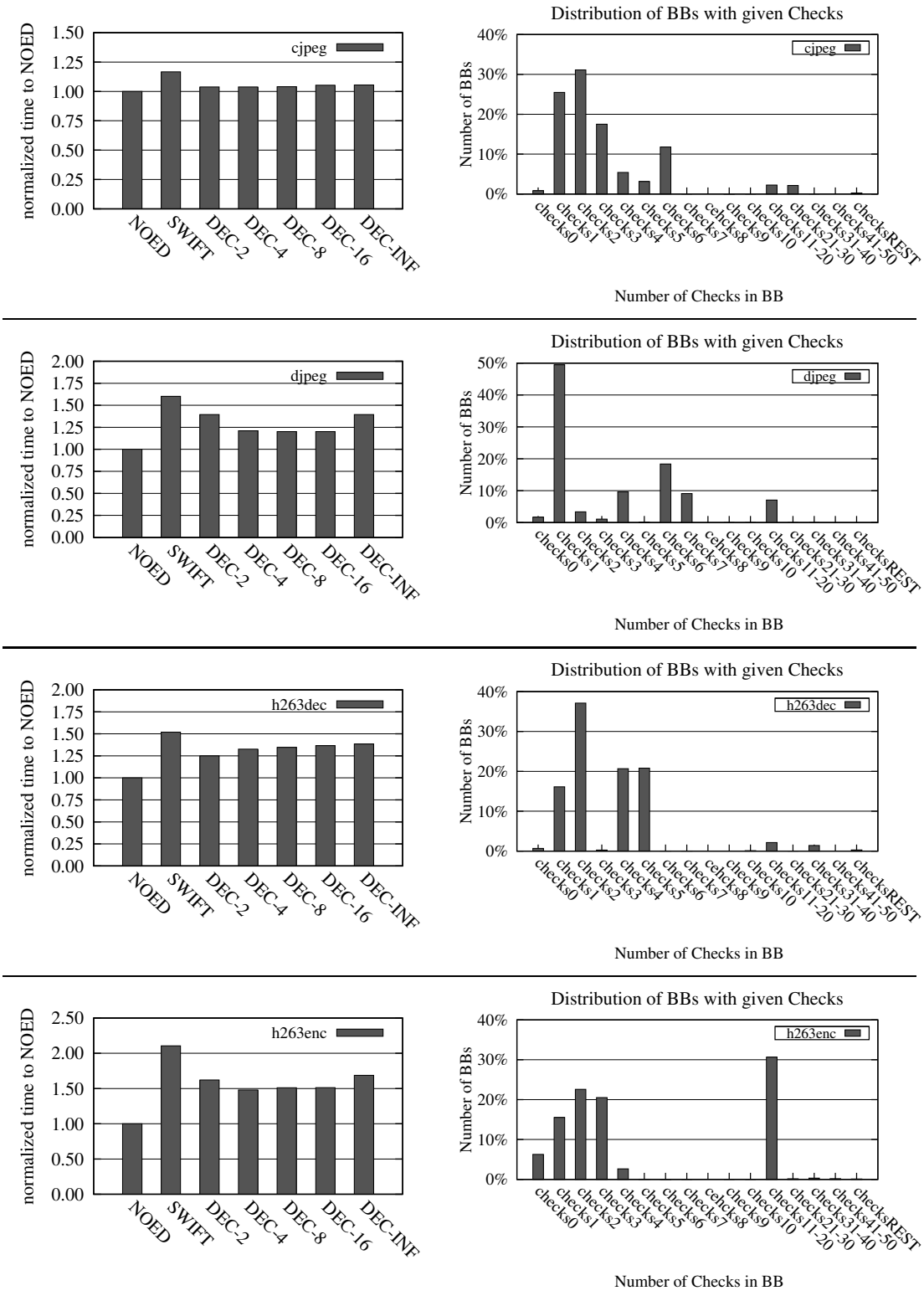


Figure 3.10: Results Part 1: The first column shows the performance improvement of DRIFT over SWIFT and NOED and the second one presents the percentage of basic-blocks that have a given number of checks.

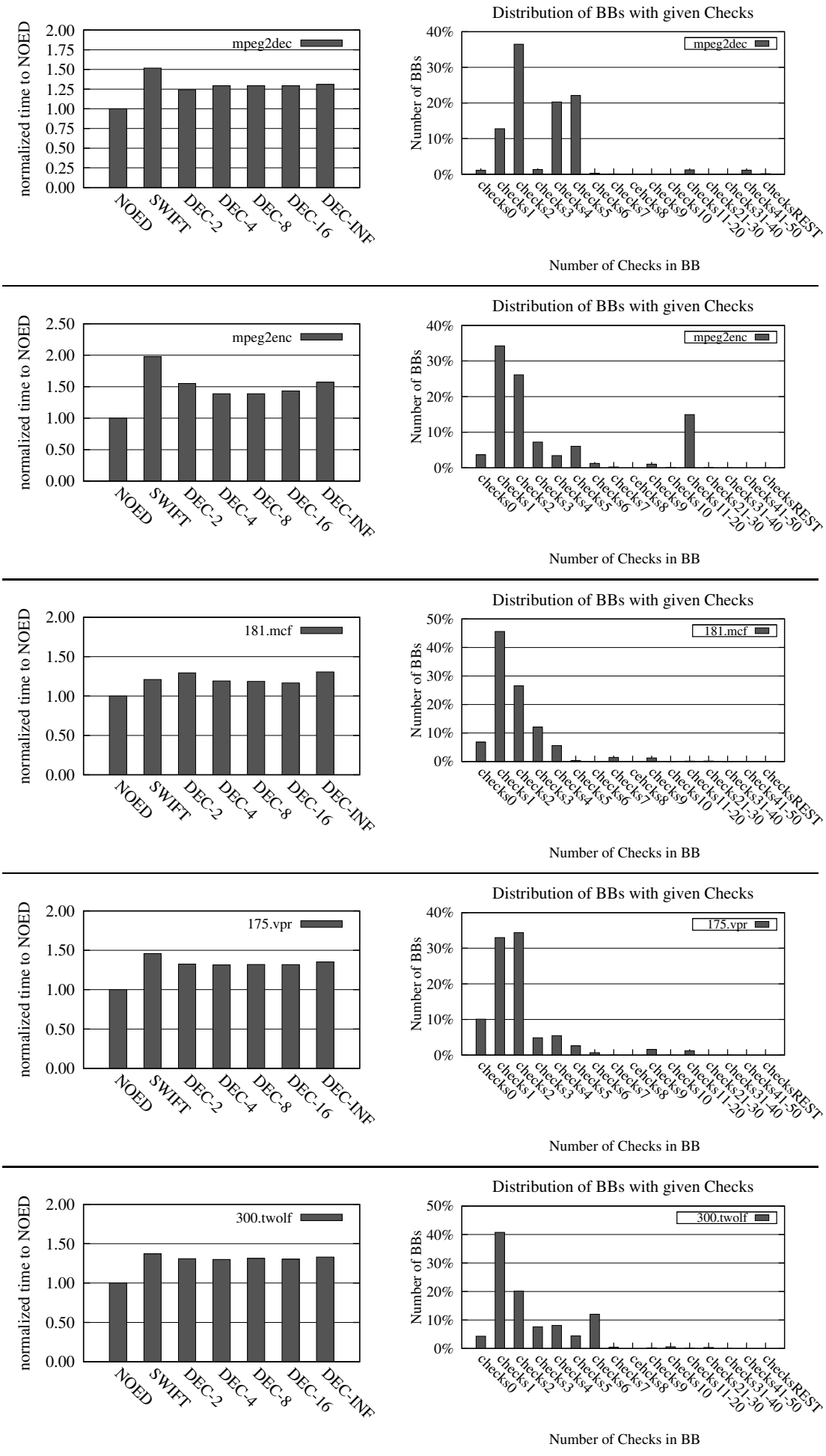


Figure 3.11: Results Part 2: Same as Part 1.

| Benchmark | Performance gain over SWIFT | Slowdown over NOED | Decouple Factor |
|---|---|---|---|
| cjpeg | 11.1% | 1.04x | 2,4 |
| djpeg | 25% | 1.2x | 8,16 |
| h263dec | 17.7% | 1.25x | 2 |
| h263enc | 29.7% | 1.48x | 4 |
| mpeg2dec | 18.2% | 1.24x | 2 |
| mpeg2enc | 28% | 1.39x | 4,8 |
| 181.mcf | 2% | 1.18x | 8 |
| 175.vpr | 10.5% | 1.31x | 4 |
| 300.twolf | 5.1% | 1.37x | 4 |

Table 3.2: DRIFT's best performance for each benchmark over SWIFT and NOED.

The performance of some benchmarks, however, degrades as the decouple factor reaches very high values (close to DEC-INF). This is the case for djpeg, h263enc and mpeg2enc. These benchmarks have many basic-blocks with a high number of checks (as shown in the second column). A high value of the decouple factor in these cases can lead to high predicate register pressure. In addition, at the end of each basic-block, we have a tree of compare instructions that slows down the code. For this reason, DEC-4 performs best for h263enc and mpeg2enc (29.7% and 28% respectively) and DEC-INF is much worse.

Table 3.2 shows the decouple factor for which DRIFT achieves the best speedup over SWIFT. From the above discussion, we can see that the best decouple factor is a trade-off between basic-block fragmentation and register pressure. The results show that DEC-4 is a good compromise between the two; DEC-4 is large enough to reduce the impact of basic-block fragmentation and small enough to avoid hardware congestion.

Figure 3.12 shows that the binary size of SWIFT is about 2.5× greater than NOED. This is expected due to the additional error detection code injected into the code stream. DRIFT generates slightly smaller binaries (2.3× larger than NOED), which is further evidence that DRIFT improves the resulting schedule, because the instructions are packed into fewer instruction bundles. As the decouple factor increases the binary size remains almost the same. Increasing the decouple factor in benchmarks with small

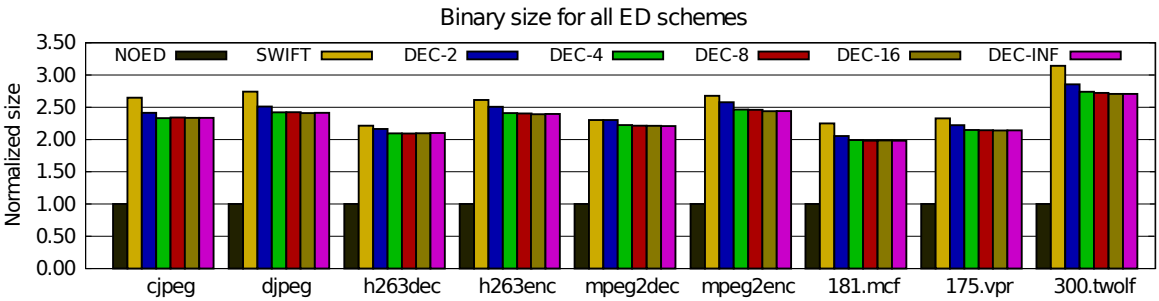


Figure 3.12: Binary code size for all benchmarks, normalized to NOED.

number of checks per basic-block does not change the code any further. In benchmarks (e.g., djpeg, h263enc and mpeg2enc) with large number of checks per basic-block, the ILP might increase as the decouple factor increases, leading to more compact code, but the register spilling adds extra code which counterbalances the code reduction.

### 3.3.3 Fault Coverage Evaluation

As described in Section 2.5, the fault coverage experiments are performed using the SKI IA-64 simulator [2]. The simulator was modified to inject errors at the output registers of instructions, which is common practice in the literature [17][25][70][91][101].

Figure 3.13 shows that DRIFT and SWIFT are almost identical in fault-coverage. In a few cases (h263enc and 181.mcf), some of the detected errors in SWIFT are transformed into exceptions in DRIFT. As we explained in Section 2.3.1, both SWIFT's and DRIFT's Sphere of Replication do not include store instructions. Therefore, store instructions are not replicated. In SWIFT, a check is inserted before every non-replicated instruction in order to prohibit corrupted data to propagate to memory. DRIFT delays the execution of some of the checks. Thus, some stores might be executed before verification takes place, leading to exceptions raised by the system. These exceptions are detected by our exception handler (as done in DAFT [101]). As in all high fault-coverage techniques, Data-corruption and Time-out errors are very rare. Therefore, DRIFT has practically the same fault-coverage as SWIFT even for high values of the decouple factor.

In the performance evaluation (Section 3.3.2), we showed that a decouple factor of 4 always improves system performance. The fault-coverage results show that it has very good fault-coverage as well.

Finally, we observe that the *computational nature of the benchmark* plays an im-

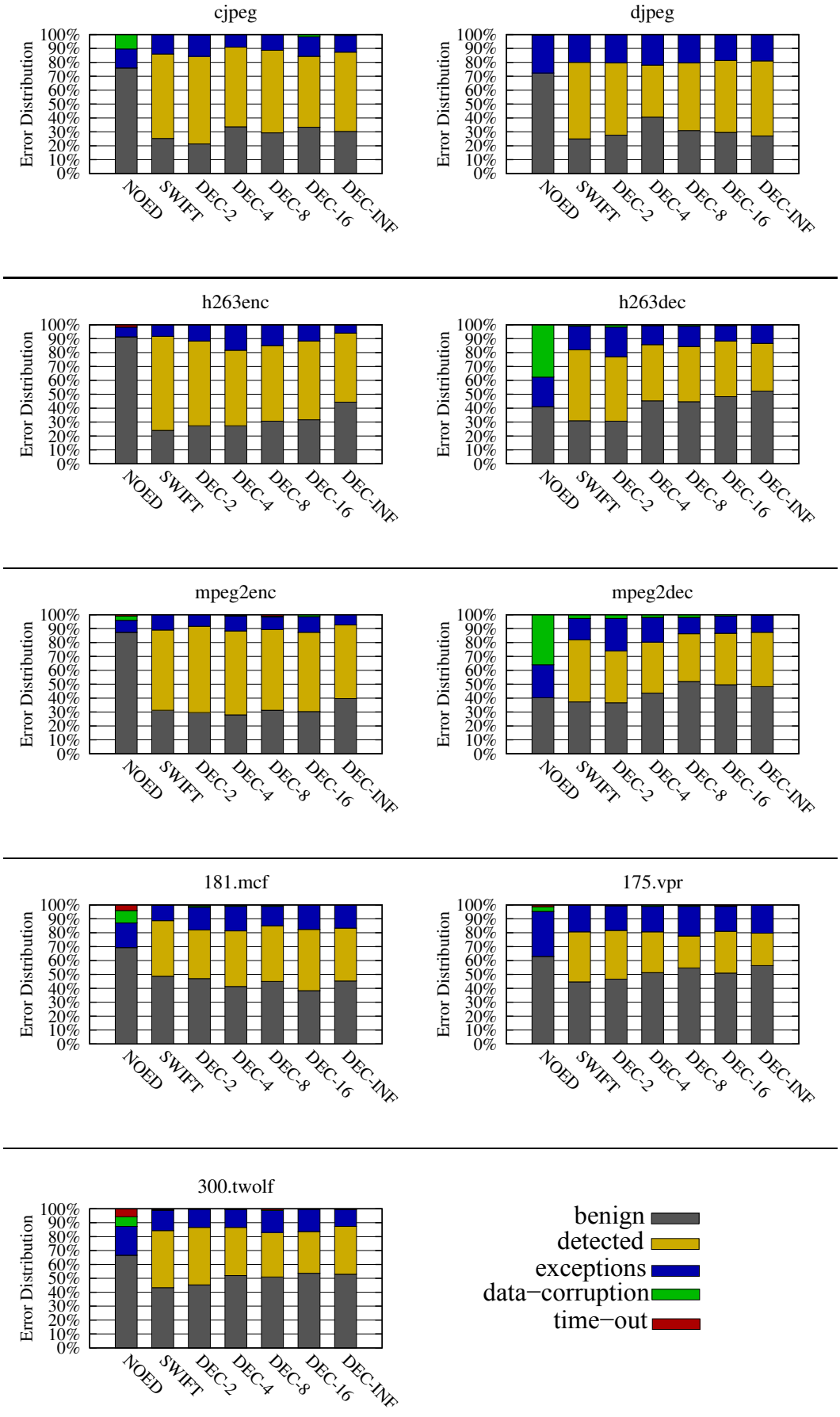


Figure 3.13: Fault coverage results for NOED, SWIFT and different decouple factors of DRIFT.

portant role on fault-coverage. For example, mpeg2enc, cjpeg and h263enc are media encoding benchmarks. The encoding process usually involves data compression or loss of input information (e.g., sampling) which by definition ignores parts of the input. If an error occurs on data that gets compressed, then it may not propagate at all and it will not appear in the output of the program. For this reason, NOED has almost 90% benign errors. In this type of application, decoupling is less risky.

## 3.4 Summary

In this chapter, we presented DRIFT, the first work that explores and solves a significant performance limitation in thread-local instruction-level error detection methodologies, namely, basic-block fragmentation. DRIFT is based on the idea of decoupling which breaks the execute-check-confirm-execute synchronization cycle existing in synchronized schemes. DRIFT decouples the execution of the code from the checks, resulting in code that the scheduler can better optimize as it is no longer constrained by the complex control flow caused by the frequent checking. Our evaluation on a real machine shows significant performance improvements up to 29.7% and average performance overhead of 1.29× compared to native, non-fault tolerant, code. The performance gains have no impact on the fault-coverage compared to synchronized schemes.

# Chapter 4

# CASTED: Core-Adaptive Software-based Transient Error Detection

In this chapter, we present CASTED, an instruction-level error detection technique for architectures with tightly-coupled cores. Current state-of-the-art error detection techniques are not adaptive. As a result, they fail to map the code in different architecture configurations like the issue-width and the communication latency between the cores. Hence, the code is misplaced on the cores and the overhead of error detection becomes a performance bottleneck. CASTED proposes a technique which takes into consideration the architecture configurations and it generates code for each configuration in such a way as to decrease the impact of error detection on performance. Consequently, CASTED improves the placement of the code for each architecture configuration and the performance. In some cases, it outperforms, by up to 21.2%, the best fixed approaches (single-core or dual-core).

## 4.1 Motivation

### 4.1.1 Limitations of Single-core and Dual-core Error Detection

Figure 4.1 summarizes the main instruction-level error detection methodologies as they were described in Section 2.2.2. Figure 4.1.a shows single-core error detection that is executed on one core and it is described in Section 2.4. Figure 4.1.b presents the dual-core technique where the original code runs on one core and the replicated code with

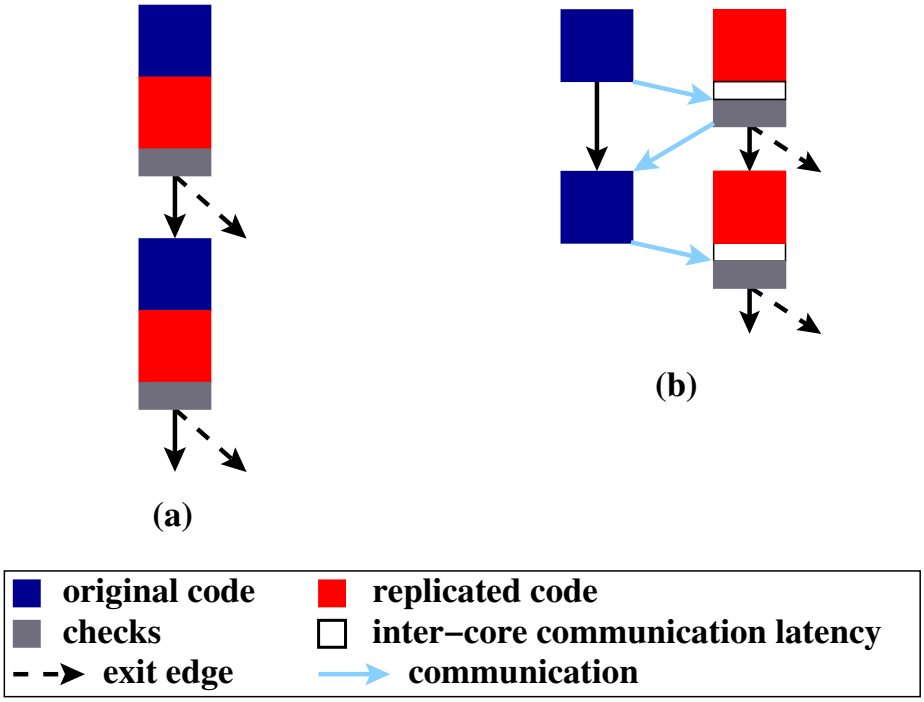


Figure 4.1: Summary of the existing instruction-level error detection methodologies: (a) Single-core instruction-level error detection, (b) Dual-core instruction-level error detection.

the checks on another one.

Both techniques have limitations. The first one requires many resources within a core (such as functional units, registers) to accommodate the overhead of redundant code and it suffers from basic-block fragmentation as it was mentioned in Chapter 3. The second one requires an extra core and has a large communication overhead since the original code has to frequently send data to the checker code. The original code sends the values that are about to be checked in the checker code. These two factors make communication critical in dual-core error detection technique.

We show that these techniques are sub-optimal for our target. Tightly-coupled architectures (VLIW clusters [23], RAW [90], VOLTRON [102]) differ from traditional multi-core architectures in the inter-core communication latency. Contrary to the multi-core systems, data can be communicated across cores relatively fast. The communication latency between the cores is just a few cycles since the communication occurs between the register file of one core and the register file of the other (Figure 4.2). Therefore, the single-core technique does not fully benefit from the available resources of tightly-coupled cores. On the other hand, the dual-core technique does not take into consideration the communication latency in the placement of the code. As

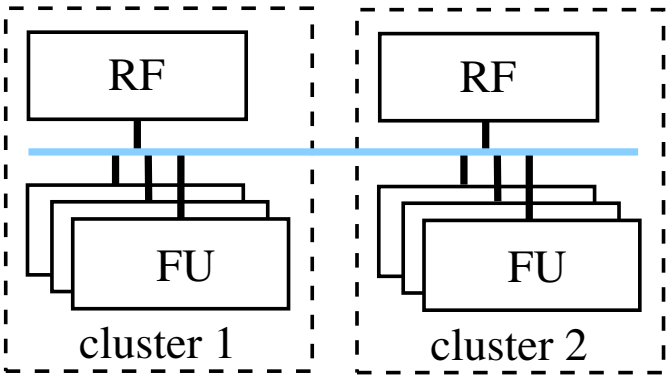


Figure 4.2: Clustered architecture with 2 clusters (RF = Register File, FU = Functional Unit). The light blue line indicates the inter-core interconnect.

a result, the frequent communication becomes a performance bottleneck for the dual-core technique.

## 4.2 CASTED

CASTED tries to ”hide” the error detection overhead by fully exploiting the available ILP of architectures with tightly-coupled cores. As was mentioned earlier, the communication latency between the cores in such architectures is very small. This feature is exploited by CASTED to distribute the error detection overhead across cores in a fine-grain fashion (that is, distributing the workload at an instruction-level granularity). This can boost performance since the original and replicated code have no true dependency and thus can run in parallel.

Depending on the system setup, the architecture might be configured in many ways. Parameters such as the issue-width, the inter-core delay and the number of available cores can change across designs. The challenge for CASTED is to fully take advantage of the available resources and to effectively distribute the error detection overhead no matter what configuration is used. CASTED uses these parameters to decide whether it is preferable to assign the whole error detection code to one core or it is more efficient to split the code onto different cores. The adjustment of the generated code to each and every architecture configuration is the main feature of CASTED.

Figure 4.3 shows how CASTED adjusts to different architecture configurations. Given architecture parameters such as the issue-width and the communication latency, CASTED generates code for each architecture. For example, if the resources within a core are not many and the communication latency is small, then scheduling the original

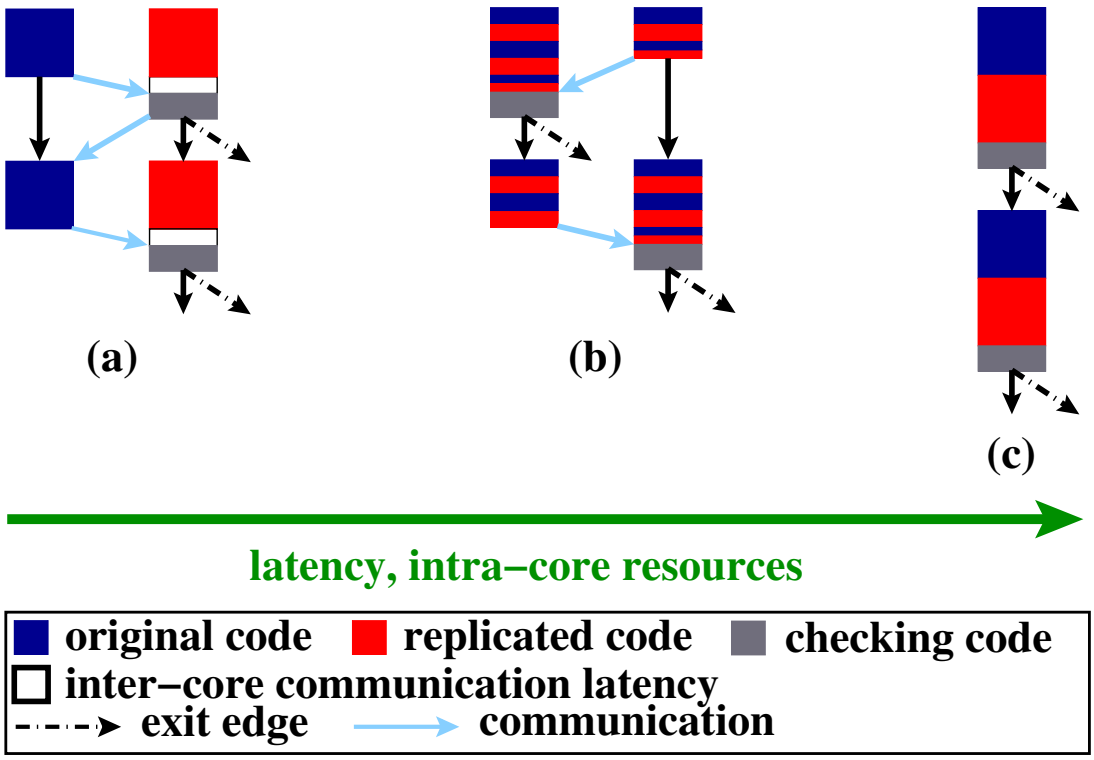


Figure 4.3: CASTED behaves similar to the best technique for each architecture configuration: if the inter-core communication latency and the ILP are small, then CASTED generates similar code as the dual-core scheme (a). On the other hand, if the inter-core communication latency and the ILP are large, then CASTED tends to put all the code on one core as the single-core scheme does (c). For all the other cases, CASTED distributes the error detection code across the cores trying to reduce the overhead of error detection.

and the replicated code in separated dedicated cores performs better. Thus, CASTED generates similar code to the dual-core technique. On the other hand, if many resources are available within a core and the communication latency is large, then scheduling all code in the same core is preferable. As a result, CASTED puts all the code on one cluster. For all the other cases, CASTED places the code across all clusters trying to reduce the error detection overhead. The difference with respect to the dual-core technique is that CASTED is not restricted to run the original instructions on one core and the redundant ones on another one. As it is shown in Figure 4.3.b, CASTED places any instruction on any cluster producing near-optimal code.

In this way, CASTED distributes the error detection overhead across all available resources. It achieves this by placing the code between the cores considering the different architecture configurations like the issue-width and the communication la-

tency. Compared to single-core technique, CASTED uses all the available hardware resources. Contrary to dual-core technique, it optimizes the code so as to hide the communication overhead. All the above enable CASTED to achieve performance at least as high as the best performing of the existing techniques on any configuration. Occasionally, it also outperforms the best fixed technique (Section 4.4).

### 4.2.1 Motivating Examples

The following examples show the lack of adaptivity of the existing error detection techniques and they demonstrate how CASTED works. All the examples (Figures 4.4 - 4.6) are based on some sample code with the Data Flow Graph (DFG) shown on the left of each figure. This code is referred to as the *original* code. Figures 4.4.c, 4.5.c and 4.6.c show the DFG of the error detection code. The error detection DFG shows some important attributes of the error-detection code:

i. The error detection DFG is much larger (in node count) than the original DFG. This is because of i. the numerous replicated instructions (in red) and ii. the check instructions (in grey) just before each non-replicated (N.R.) node.

ii. Its critical path is longer because of the check instructions.

iii. Its ILP is higher compared to the original code. This is because the replicated instructions can be executed in parallel with their respective original instructions.

To quantify the performance of each scheme, we show the corresponding instruction schedule after applying the error detection algorithms (SCED, DCED and CASTED) on our target (Figures 4.4.d,e,f, 4.5.d,e,f and 4.6.d,e,f).

The examples show the schedules of:

i. The original code (Figures 4.4.b, 4.5.b and 4.6.b) with no error detection (NOED). The empty schedule slots are NOPs.

ii. The Single-Core Error Detection (SCED) approach (Figures 4.4.d, 4.5.d and 4.6.d) where the original and the replicated codes are interleaved.

iii. The Dual-Core Error Detection (DCED) approach (Figures 4.4.e, 4.5.e and 4.6.e) where the original code always runs on one core and the replicated and checking code (redundant code) always runs on the second one. The non-replicated (N.R.) instructions (store and control-flow instructions) are only executed in the main

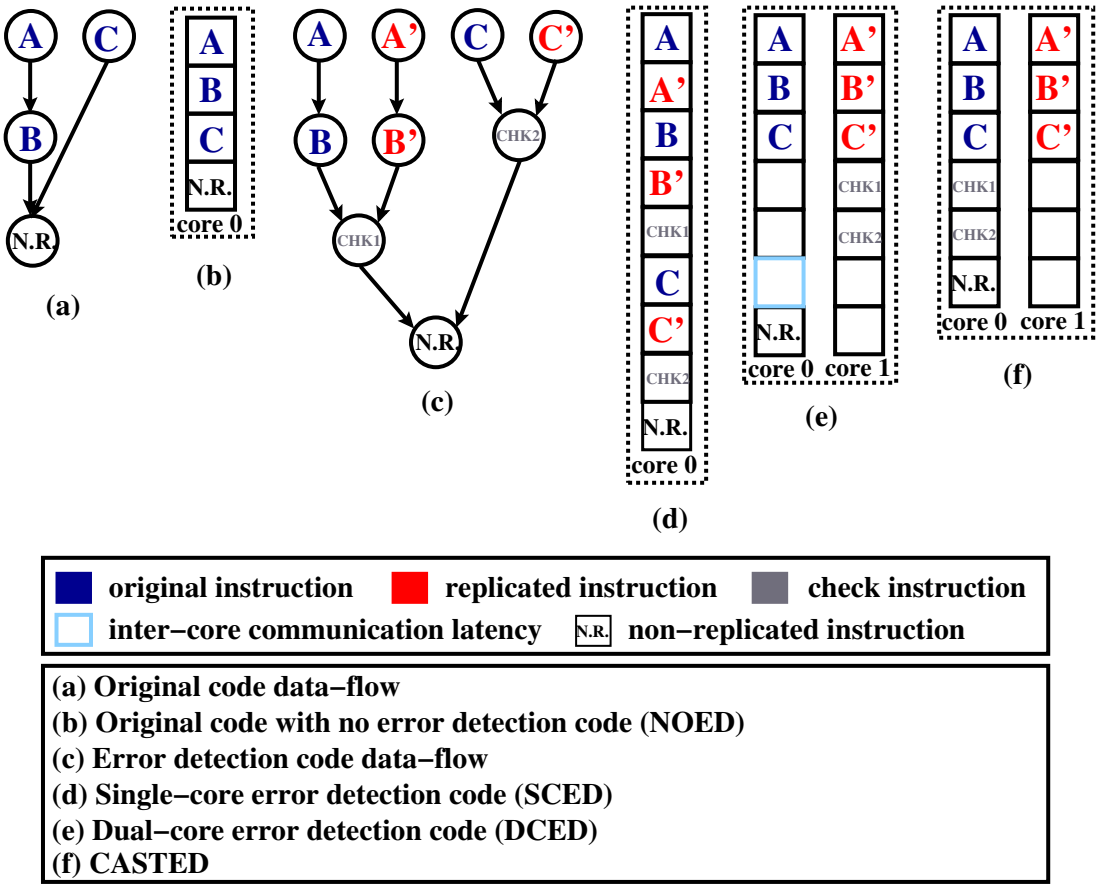


Figure 4.4: Resource constrained example. It is assumed that the machine has one issue-slot and the communication latency is one cycle. DCED outperforms SCED since it uses more resources. As a result, CASTED behaves similar to DCED and optimizes DCED by scheduling better the code.

code. The verification is only done by the checker code. The communication between the cores is done implicitly (without explicit copy instructions) and the micro-architecture handles the data transfer across register files. The NOPs that are attributed to the communication latency are shown in light blue.

iv. In CASTED (Figures 4.4.f, 4.5.f and 4.6.f), the instructions of the error detection code (original code, replicated code and checks) are assigned to the cores taking into consideration the underlying architecture configurations.

#### 4.2.1.1 Resource Constrained Example

In the example shown in Figure 4.4, each core is single-issue and the communication latency for this example is set to 1 cycle. This setup might be simplistic but helps us

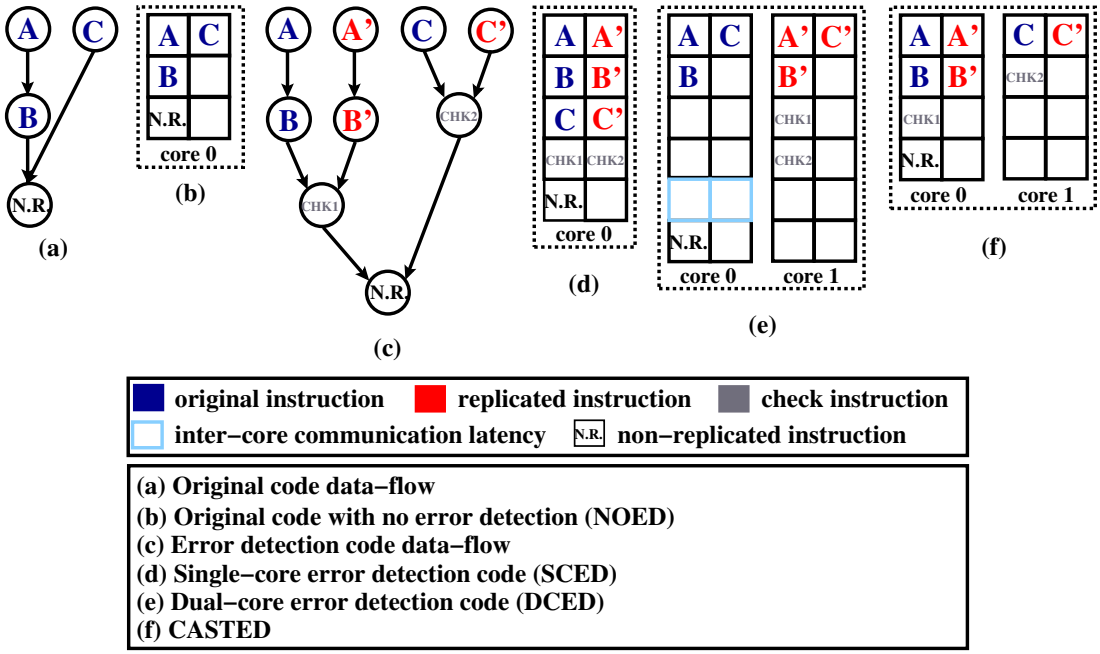


Figure 4.5: Resource rich example. The machine has two issue-slots and the communication latency is one cycle. DCED does not benefit as much as SCED from the extra resources. CASTED schedules the code so as to use all hardware resources and hide the inter-core communication latency.

point out the shortcomings of the existing approaches. We observe that the dual-core case (Figure 4.4.e) outperforms the single-core one (Figure 4.4.d). This is due to the fact that the single-issue single core is *resource constrained* and as such it can not effectively execute both the original code an the replicated code. On the other hand, the dual-core technique uses more resources and performs better. CASTED (Figure 4.4.f) makes better use of the resources. It assigns the instructions of the original and the replicated code to the first available core. For example, the checks and a part of the replicated code are executed in the main cluster as this will speed up the algorithm. In this way, CASTED schedules the instruction in such a way so as to hide the communication penalty.

#### 4.2.1.2 Resource Rich Example

The example of Figure 4.5 shows that the dual-core technique does not benefit from the extra resources within each core as much as the single-core technique. In more detail, in Figure 4.5, each core is two-wide issue and the communication latency is 1 cycle. We observe that the single-core case (Figure 4.5.d) outperforms the dual-core

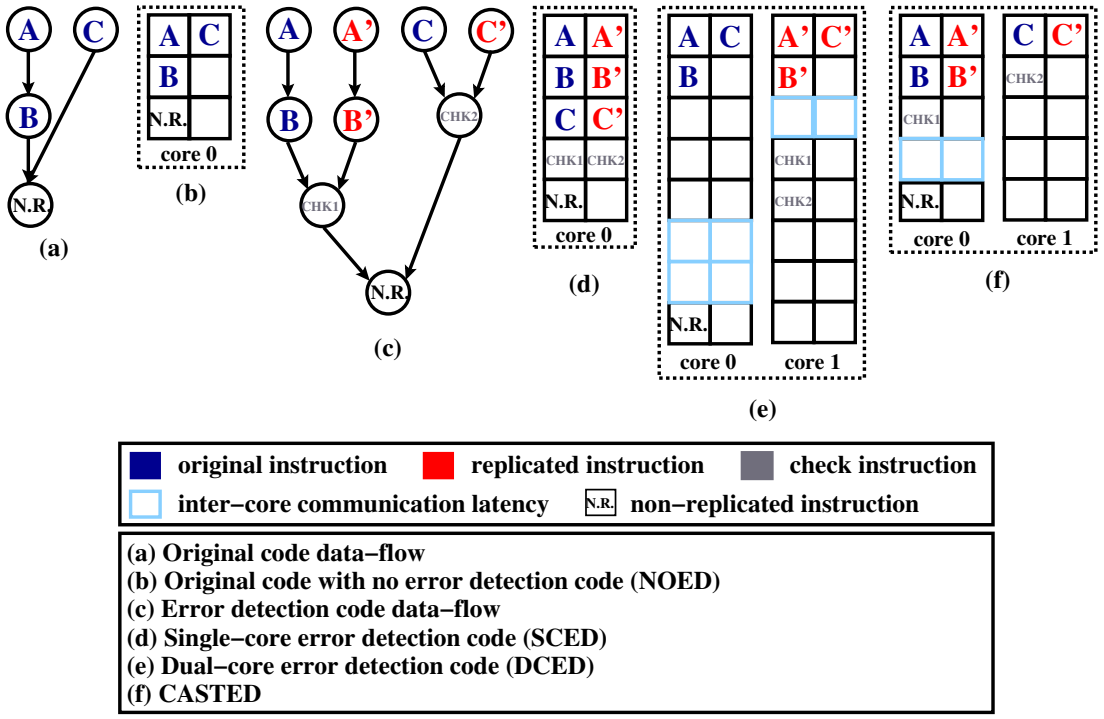


Figure 4.6: Latency constrained example. The machine has two issue-slots and the communication latency is two cycles. DCED suffers from the inter-core communication latency. CASTED is also affected by the overhead of the inter-core communication latency, but it reduces communication's impact by better placing the code at the cores.

(Figure 4.5.e) for two reasons: *i.* Each core is wide-issue enough to accommodate the ILP of the error detection code with just a few cycles of overhead compared to NOED. *ii.* The dual-core case has more resources, but the sub-optimal fixed placement of the original and checker code instructions on the first and second core hurts the performance. CASTED (Figure 4.5.f) outperforms both approaches. The state-of-the-art approaches fail to generate code for different architecture configurations. On the other hand, CASTED produces near-optimal code by adjusting the placement of the instructions to the cores. This is because it is delay-aware and assigns the instruction to the cores in such a way that the delay does not become the bottleneck. It is worth noting that the replicated instructions and the check instructions are moved across cores in an attempt to minimize the cycle count.

#### 4.2.1.3 Latency Constrained Example

The example of Figure 4.6 shows a case where the *inter-core delay* can become the performance bottleneck for the dual-core case. In this example, the communication

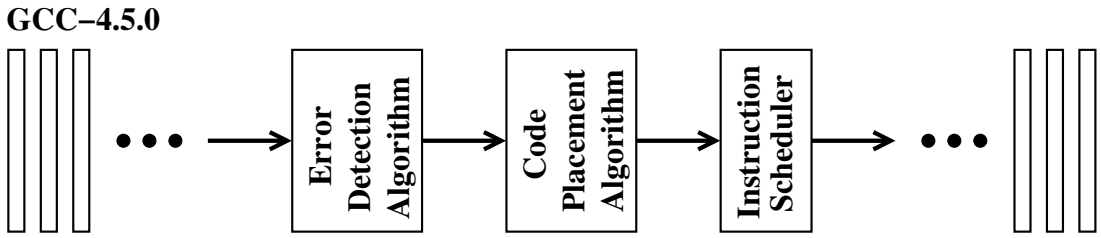


Figure 4.7: The two passes of CASTED (error detection and code placement) are placed at the back-end of GCC.

latency between the cores is 2 cycles. Both the dual-core technique and CASTED are affected by the increase in communication latency. However, CASTED manages to perform as well as the single-core technique which is not affected by the communication latency. CASTED fully exploits the available ILP and schedules four instructions in parallel. In addition, it hides the penalty of the communication latency by executing one check on the main core and the other on the checker core.

All the above examples present the limitations of current techniques to adjust to different architecture configurations. In contrast, CASTED maps the error detection code on each architecture configuration. The produced code is sometimes better than the schedule of the best non-adaptive technique (Figures 4.4 and 4.5).

## 4.3 CASTED Algorithm

CASTED comprises of two algorithms which are implemented in two separate passes in the back-end of GCC (Figure 4.7). CASTED uses the following algorithms:

- The first algorithm generates the error detection code.
- The second algorithm is the one responsible for the placement of the code. It is based on [21] and is the one that assigns the instructions to the cores taking into consideration the issue-width and the communication latency.

### 4.3.1 Error Detection

CASTED uses the standard thread-local instruction-level error detection algorithm (SWIFT) to generate the error detection code since this is the standard baseline assumed by previous work in the literature [25][71]. The SWIFT algorithm was described in Section 2.4.2. In short, these steps do the following tasks:

- *Replicate* the instructions of the program that belong to the sphere of replication (Section 2.3.1).
- As it was mentioned in Section 3.2.3, the original code should not read/write the register values from/to the same registers as the replicated code. Therefore, the original code is *isolated* from redundant code using register renaming.
- Finally, the checks are *emitted* before the non-replicated instructions.

### 4.3.2 Code Placement

After the generation of the error detection code, CASTED assigns the code to the available cores. This is done using the Bottom-Up-Greedy (BUG) clustering algorithm [21] (BUG Algorithm). As its name suggests, it is a greedy algorithm that makes the clustering decision based on the completion cycle of the instruction into consideration; each instruction gets assigned to the core where it will execute the earliest. The completion cycle heuristic is aware of the inter-core delays and can therefore adjust its behavior on any architecture configuration.

BUG Algorithm

```
1 /* The main function of BUG algorithm '*/
2 bug (node)
3 {
4   if node is leaf OR node is assigned
5      return;
6   /* Visit the instructions in topological order giving
       ↪preference to the critical path */
7   for node's predecessor sorted by critical path
8       bug (predecessor)
9
10  /* Calculate the completion cycle heuristic */
11  sorted cores,sorted cycles=compl_cycle(node)
12  /* Assign NODE to CORE and CYCLE */
13  node.core = FIRST (best cores)
14  node.cycle = FIRST (best cycles)
15
16  /* Reserve issue slots in reservation table */
17  reservation set(core, cycle)
18 }
```

In more detail, the algorithm walks through the Data Flow Graph in a topological order, by giving preference to the instructions in the critical path (BUG Algorithm line 7). For each instruction, it calculates the value of the completion cycle (BUG Algorithm line 11) and selects the core that corresponds to the lowest cycle (BUG Algorithm line 13). The completion cycle is resource aware. After the core assignment decision has been made, that specific resource (that is the cycle and the chosen core) is marked as used in the reservation table (BUG Algorithm line 17).

In Figure 4.4.f, CASTED identifies that the execution of the replicated instructions (A', B' and C') in the second cluster is beneficial for performance as the communication latency overlaps with the execution of the checks. Similarly, executing both checks in the second cluster is expensive because of the communication latency. Therefore, CASTED places the checks in the first cluster. Contrary to existing schemes, checks can migrate from one cluster to the other when appropriate (Figure 4.4). In addition, the non-replicated instructions are executed on both cores. In the case of memory instructions, this improves memory level parallelism (MLP). In this way, CASTED balances the use of hardware resources, unlike the other approaches.

## 4.4 Results and Analysis

### 4.4.1 Experimental Setup

The CASTED system is implemented as two back-end passes in GCC-4.5.0 [1] compiler infrastructure. We implemented both the error detection and the core-assignment algorithms in separate passes placed just before the first instruction scheduling pass, as illustrated in Figure 4.7.

CASTED works on tightly-coupled cores such as VLIW clusters [23], RAW[90] and VOLTRON[102]. In this work, we use a clustered VLIW architecture with the Itanium 2 [78] instruction set. Both clusters operate in lockstep execution. Each cluster can access the other cluster's register file but this has an increased latency (the inter-core communication latency) since it has to go through the interconnect.

The processor configuration is listed in Table 4.1. We simulate the execution on a modified SKI IA-64 simulator [2]. The modified simulator is a cycle-accurate Itanium 2 simulator with a full cache memory hierarchy, the same as the one of Itanium 2. The register file per cluster is half of that assumed in the single cluster configuration.

We evaluated our software error detection scheme on 7 benchmarks, 4 from the

| Processor: IA64 based clustered VLIW | | | | |
|---|---|---|---|---|
| Clusters: | 2 | | | |
| Issue width: | configurable | | | |
| Instruction Latencies: | configurable | | | |
| Register File: | (64GP, 64FL, 32PR) per cluster | | | |
| Branch Prediction: | Perfect | | | |
| Cache: Levels 3 (same as Itanium2 [49]) | | | | |
| Levels : | L1 | L2 | L3 | Main |
| Size (Bytes): | 16K | 256K | 3M | ∞ |
| Block size (Bytes): | 64 | 128 | 128 | - |
| Associativity: | 4-way | 8-way | 12-way | - |
| Latency (cycles): | 1 | 5 | 12 | 150 |
| Non-Blocking: | YES | YES | YES | - |

Table 4.1: SKI IA64 simulator configuration.

Mediabench II video [28] and 3 from the SPEC CINT2000 [34] benchmarks. We ran the benchmarks to completion. All benchmarks were compiled with optimizations enabled (-O1 flag) and with instruction scheduling enabled. Similar to the DRIFT implementation (Section 3.3.1), we turned off the late stages of the Common Subexpression Elimination (CSE) and Dead Code Elimination (DCE) optimizations that get called after the CASTED passes. This is important to prevent the compiler from removing the replicated code and the checks.

### 4.4.2 Performance Evaluation

We evaluate the performance of CASTED by comparing it against the Single-Core Error Detection (SCED), the Dual-Core Error Detection (DCED) and the single-core No Error Detection (NOED)(this is the unmodified code). The performance results for all benchmarks for various issue widths and inter-core delays are shown in Figures 4.8 - 4.11. These results are normalized to NOED for each issue width (that is all issue 1 results are normalized to NOED-issue 1, all issue 2 to NOED-issue 2, etc.).

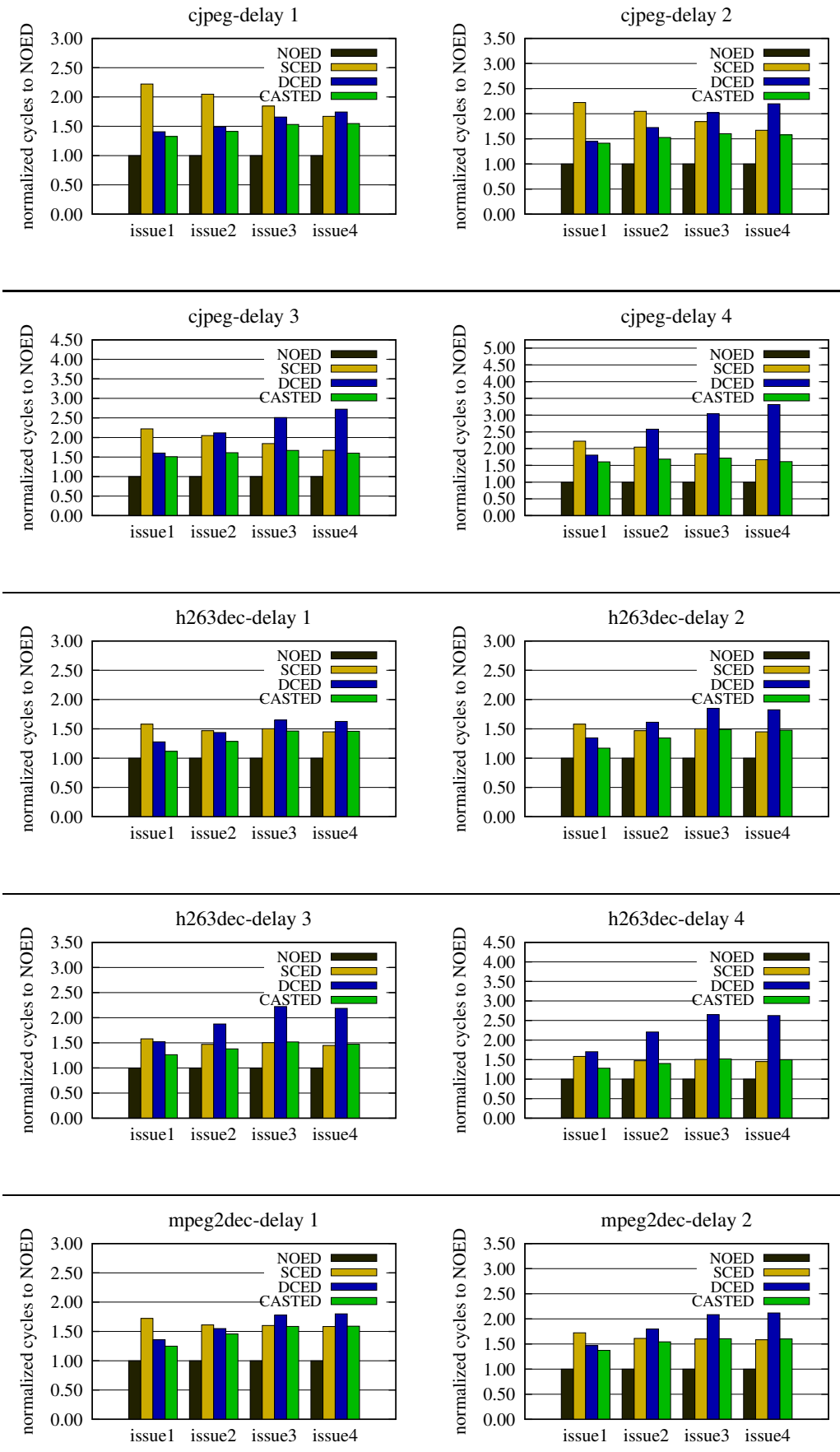


Figure 4.8: Performance for delays of 1 to 4 and issue-width per cluster in the range of 1 to 4, normalized to NOED for each issue-width (part 1).

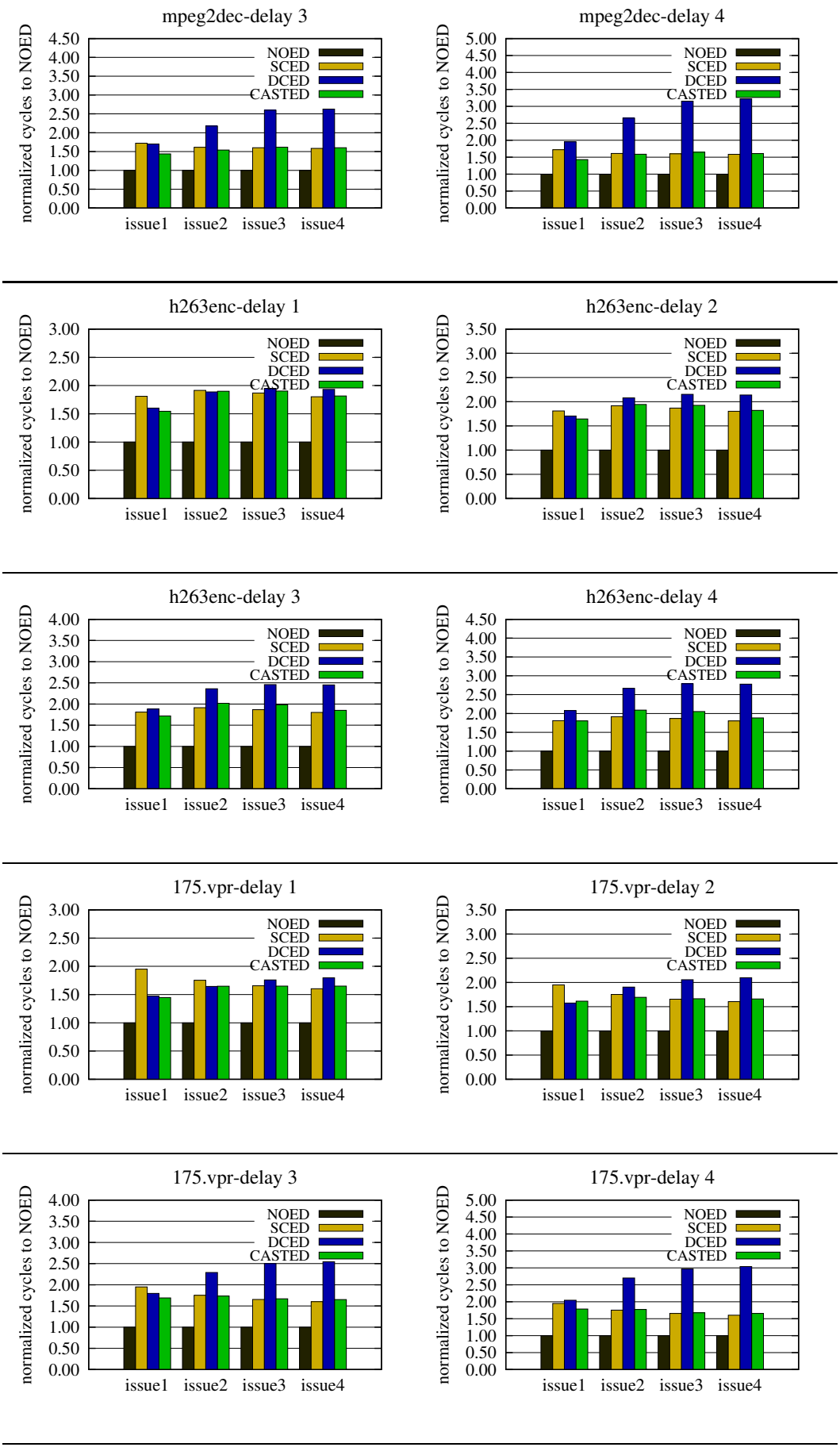


Figure 4.9: Performance for delays of 1 to 4 and issue-width per cluster in the range of 1 to 4, normalized to NOED for each issue-width (part 2).

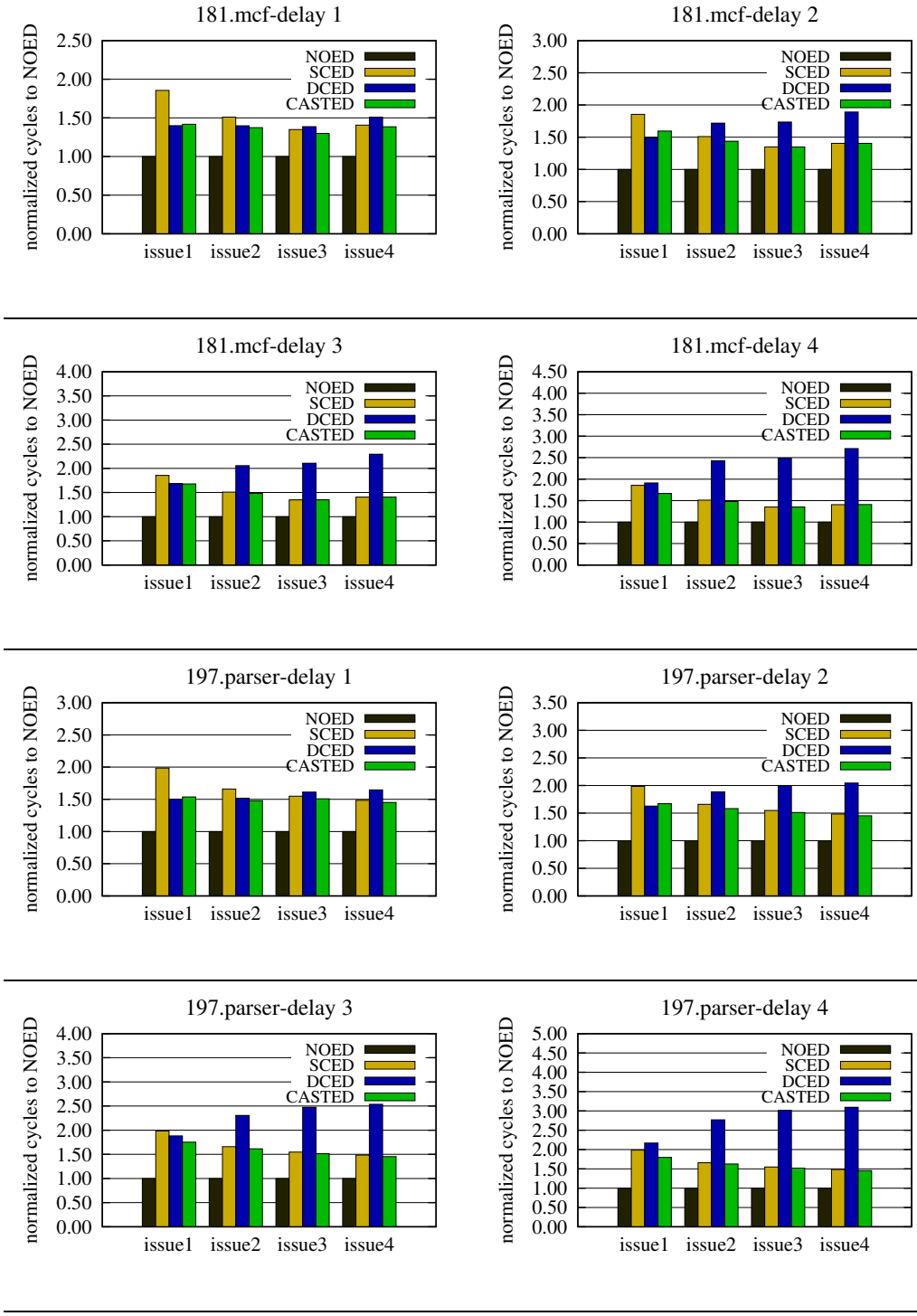


Figure 4.10: Performance for delays of 1 to 4 and issue-width per cluster in the range of 1 to 4, normalized to NOED for each issue-width (part 3).

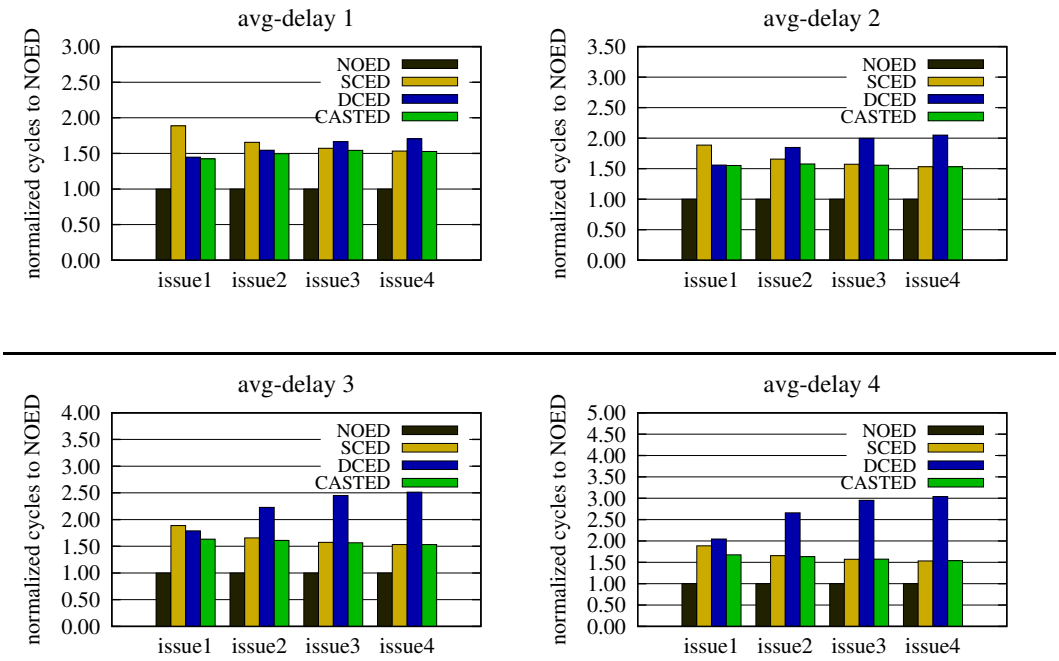


Figure 4.11: Average Performance for delays of 1 to 4 and issue-width per cluster in the range of 1 to 4, normalized to NOED for each issue-width.

#### 4.4.2.1 SCED Slowdown

The first observation to be made is the variation in the slowdown of SCED compared to NOED across benchmarks and configurations. It varies from 1.34x to 2.22x, and is 1.7x on average. Such variation can be attributed to the variation in the quantity of the error checking code and the variation of register spilling it causes. For example, the more non-replicated instructions (e.g., store instructions, function calls) the code has, the more checks the error detection algorithm adds. In SCED, both the original and the error detection code run in one core. Therefore, the performance is only affected by the issue-width. In general, SCED's performance improves dramatically as the issue width increases. As we explained in Section 4.2 and in the motivating examples of Figures 4.4 - 4.6, the redundant code has no dependencies with the original code and can run in parallel in an ILP fashion. Once the resource constraints are no longer the bottleneck, the execution speeds up. In other words, the more available resources we have within a core, the better performance SCED achieves. The exception is h263enc which will be discussed next.

Finally, it is worth noting that the overhead of SCED over NOED follows the same

pattern as those seen in Section 3.3 for SWIFT and NOED. The differences can be attributed to the different evaluation of the two schemes. SWIFT's performance was measured on a real Itanium 2 system. On the other hand, SCED's evaluation was done on a simulator since there is no available commercial system with tightly-coupled cores. On top of that, the issue-width of the simulator is different from the one of the real system (Itanium 2 is six issue wide).

#### 4.4.2.2 SCED Scalability

Figure 4.12 shows the scaling of NOED, SCED, DCED and CASTED performance as the issue-width increases. This is a metric of the ILP, the steeper the curve, the more the ILP. In most cases, SCED scales better than NOED (Figure 4.12) which results in a decrease in the SCED-NOED performance difference as the issue-width increases. This can be clearly observed in the majority of benchmarks in Figures 4.8 - 4.10. This difference in scaling between SCED and NOED is a measure of the additional ILP of the redundant code. In applications with low ILP (e.g., 181.mcf), the original code (NOED) scales poorly with the issue-width (as there is low ILP). However, SCED scales better than NOED because of the extra ILP.

On the other hand, h263enc (Figure 4.8) is a benchmark where SCED does not scale as expected (Figure 4.12). This is because the redundant code has low ILP due to the frequent checking (basic-block fragmentation as it was shown in Section 3.1.1). The checking code consists of compare and jump instructions. Therefore, the more checks the code has, the more sequential the code becomes and according to Amdahl's law the error detection code should scale worse than NOED. The opposite can be observed in benchmarks with low ILP (such as 181.mcf). In these benchmarks, the original code scales poorly with the issue-width as there is low ILP. The error detection code has more ILP compared to NOED. Therefore, it scales better than NOED and the overhead of error detection code decreases compared to NOED (Figure 4.12).

#### 4.4.2.3 DCED Slowdown

The baseline dual-core performance (DCED) (Figure 4.8 - 4.11) also varies compared to the performance of NOED across benchmarks and configurations. The slowdown is between 1.31x and 3.32x (2.1x on average). The two factors that degrade performance were mentioned in Sections 4.2.1.2 and 4.2.1.3. The first one is the issue width. As it is shown in Figure 4.5, DCED benefits less from the increase of the issue-width.

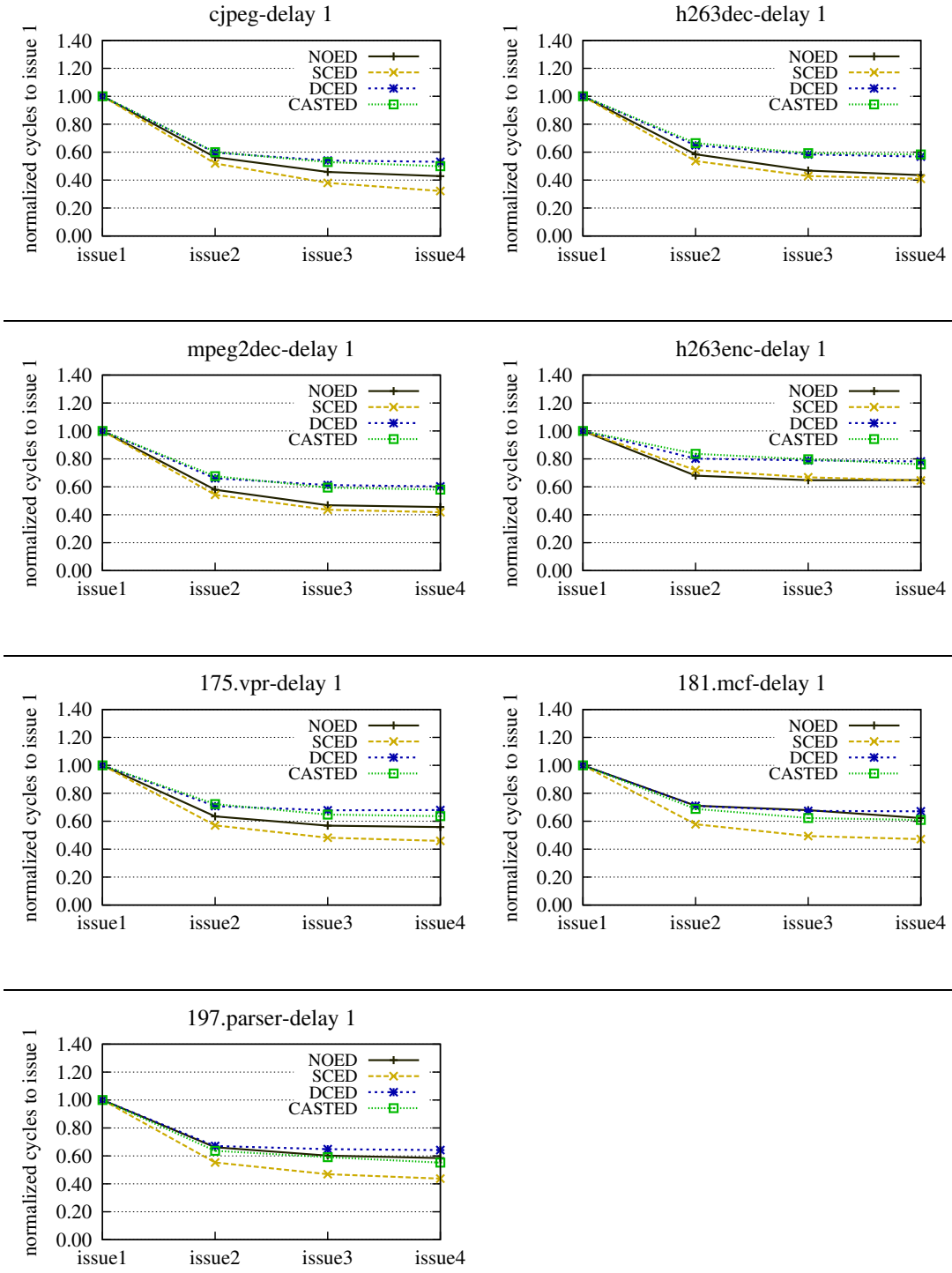


Figure 4.12: Benchmark ILP scaling.

The second and most important factor is the inter-core delay. The bigger the delay, the worse the performance. This is due to the fact that DCED performs regular inter-core communication which becomes a performance bottleneck as the inter-core delay increases.

#### 4.4.2.4 DCED Scalability

The scalability of DCED according to Figure 4.12 is worse than that of SCED. As explained previously, SCED performs better as the issue-width increases because it spreads instructions across more issue-slots. DCED has a head start. Even at issue 1, it has exploited a large part of the ILP of the redundant code as it executes it in a different core. From that point on, there is little room for improvement. This explains the strange phenomenon where the overhead of DCED against NOED increases as the issue-width increases in Figures 4.8 - 4.11.

#### 4.4.2.5 SCED vs DCED

A more interesting comparison is DCED against SCED. SCED performs better in wider-issue configurations because the high-ILP SCED code expands effectively to the available issue-slots. DCED however, cannot reach these levels of performance, as it always suffers from the inter-core latency upon checks. Things become worse for DCED when the delay is greater than or equal to three. In these cases, the communication cost between the two cores is so large that DCED performs poorly. On the other hand, when the issue-width and the inter-core latency remains low, DCED easily outperforms the resource constrained SCED.

#### 4.4.2.6 CASTED

In the majority of cases, CASTED can at least match the performance of the best performing approach (SCED or DCED) and in some cases it can even outperform the best. For instance, in Figure 4.8 h263dec-d1 for issue-width 1, the best non-adaptive is DCED and CASTED behaves similar to this technique. The ability of CASTED to adjust the error detection code in every configuration has a positive impact on its slowdown against NOED. The slowdown varies from 1.19x to 2.1x (1.58x on average). Upon low issue widths, CASTED behaves similar to DCED which is less resource constrained than SCED.

| Technique | Average Performance Overhead | Best Configuration | Lowest Average Performance Overhead |
|---|---|---|---|
| SCED | 70% | issue-width 4 | 53% |
| DCED | 110% | issue-width 1 & delay 1 | 44% |
| CASTED | 58% | issue-width 1 & delay 1 | 42% |

Table 4.2: The average performance overhead for each technique and the configuration with the lowest average performance overhead for each technique.

Furthermore, in some cases CASTED outperforms the best non-adaptive approach. This is because CASTED not only distributes the error detection code across cores (as DCED does) but it also distributes the original code if profitable. This leads to performance improvements of up to 11.4% (in cjpeg for issue 2 delay 2). As the issue-widths and delays increase, DCED is no longer the preferable approach. Instead SCED becomes the most efficient approach. As we can see, at that point, CASTED does not behave as DCED anymore, but instead it tries to generate similar code as SCED. In this case too, CASTED can outperform SCED due to the exploitation of the available resources on the distant core. The performance improvements are up to 21.2% (in cjpeg issue 2 delay 3). It has to be noticed that we compare CASTED against the baseline technique (SCED, DCED) that performs better for each configuration.

Finally, Table 4.2 summarizes the average performance overhead for each technique. On average, CASTED performs better than the other two techniques. The big slowdown of DCED is mainly due to the huge overhead of the communication when the interconnect delay increases.

The third and fourth columns of Table 4.2 present the configuration that has the lowest average performance overhead. In case of SCED, this happens when the issue-width is 4 and the average performance overhead is 53%. The average performance overhead of CASTED for issue-width 4 and delay 1 is 52%, for issue-width 4 and delay 2 is 53%, for issue-width 4 and delay 3 is 53% and for issue-width 4 and delay 4 is 54%. Once more, we see that CASTED behaves similar to SCED when the issue-width increases. In case of DCED, the lowest average performance overhead is 44% for issue-width 1 and delay 1. For this configuration, CASTED has also the lowest

average performance overhead which is 42%.

### 4.4.3 Fault Coverage Evaluation

Figure 4.13 verifies that CASTED is as good as the other high reliability methodologies. In most of the cases, there are no data-corruption or time-out errors. The presence of data corruption errors after applying CASTED, SCED or DCED is mainly attributed to the fact that these techniques cannot detect errors that occur in the system's library functions since the compiler does not have access to the library source codes to protect them. On the contrary, in some related work ([25][70][91][101]) system libraries are excluded from fault injection, which is somewhat unrealistic. If the source code of the system libraries is available, they can also be compiled with CASTED and be protected against transient errors.

Another interesting point extracted from Figure 4.13 is that encoding benchmarks (cjpeg, h263enc) are less prone to errors. This is intuitive as there is some data compression or sampling involved. Finally, we observe that most of the errors are exceptions. This is acceptable since exceptions can be easily detected by an exception handler.

In Figures 4.14 and 4.15, it is shown how CASTED error detection algorithm behaves under different architecture configurations for the h263dec benchmark. The fault coverage, as expected, is not affected by the underlying architecture configuration and CASTED retains the same level of reliability. The variation in fault-coverage results is mainly attributed to statistical deviation. Overall, Figures 4.8 - 4.11, 4.13 and 4.14 - 4.15 validate our previous claim that CASTED can adjust to different architecture configurations without any impact on reliability.

## 4.5 Conclusion

We presented CASTED, a novel software-based error detection scheme for architectures with tightly-coupled cores. CASTED effectively distributes the impact of the error detection overhead across the available resources and generates near-optimal code for each configuration. This improves performance without affecting the fault coverage across the architecture configurations. It reduces the overall slowdown by 7.5% against the single-core error detection and 24.7% against the dual-core case.

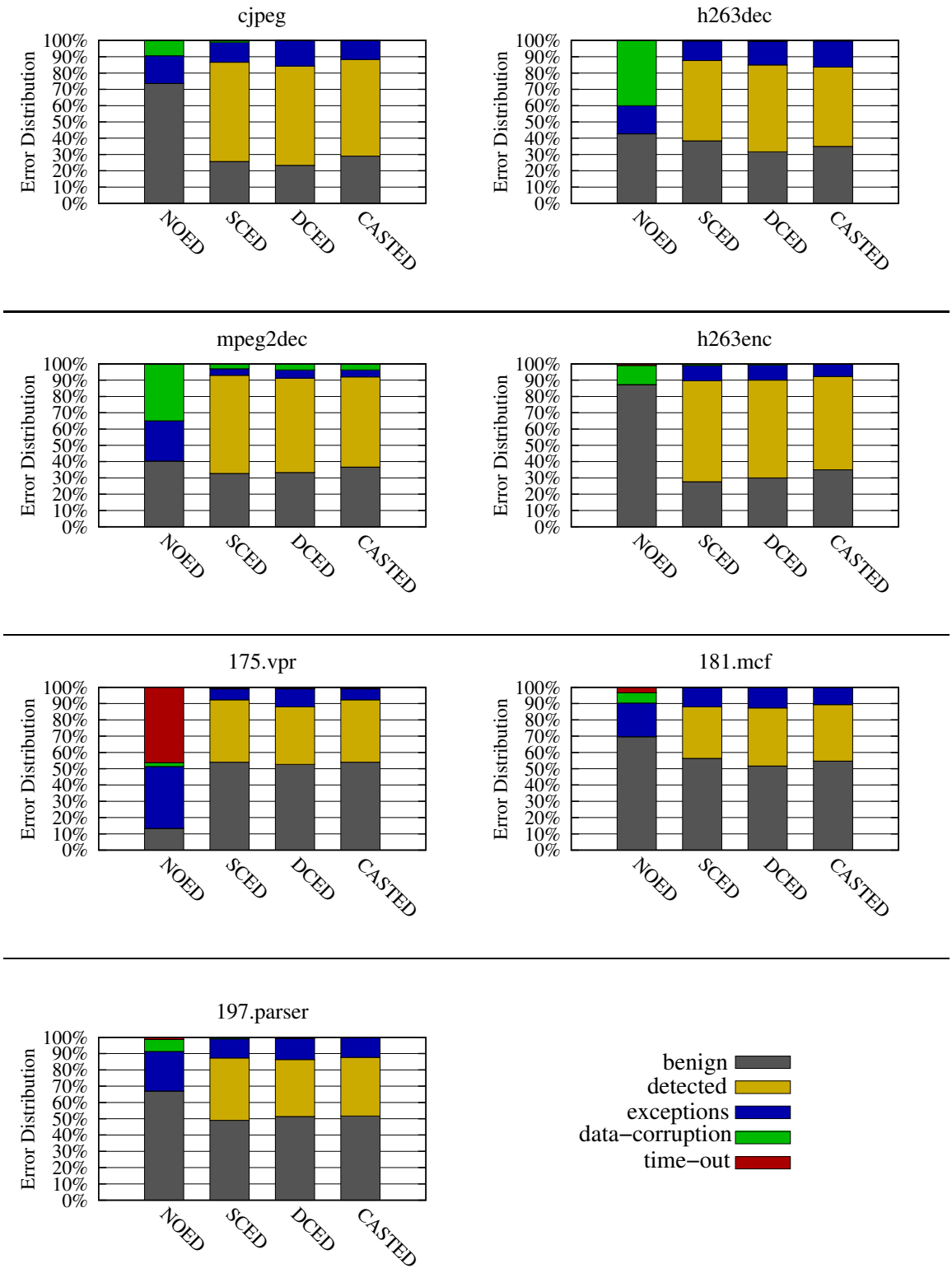


Figure 4.13: Fault-coverage results for NOED, SCED, DCED and CASTED for issue-width=2 and delay=2.

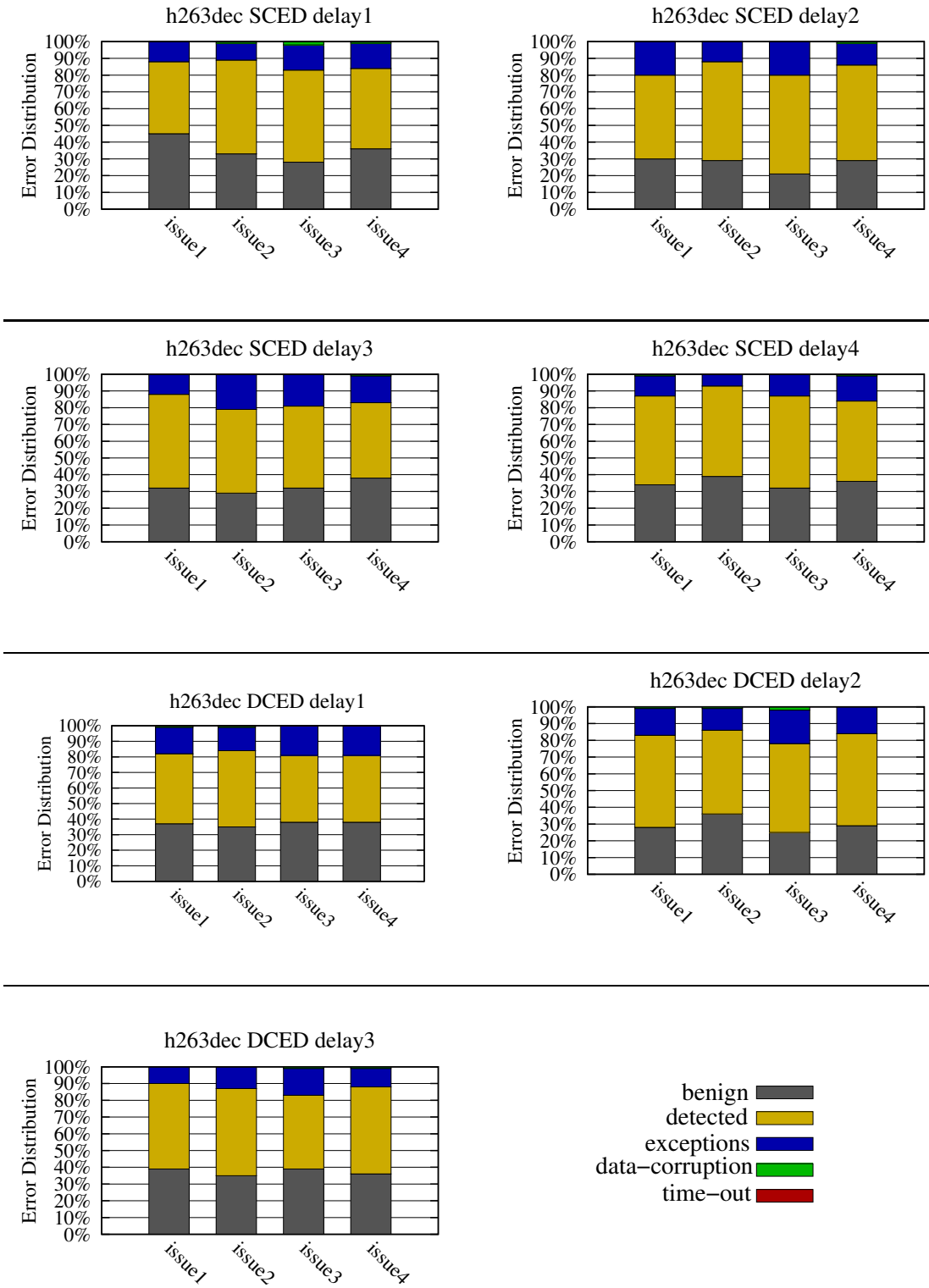


Figure 4.14: The fault-coverage of h263dec benchmark for NOED,SCED,DCED and CASTED for issue 1 to 4 and delay 1 to 4 (part 1).

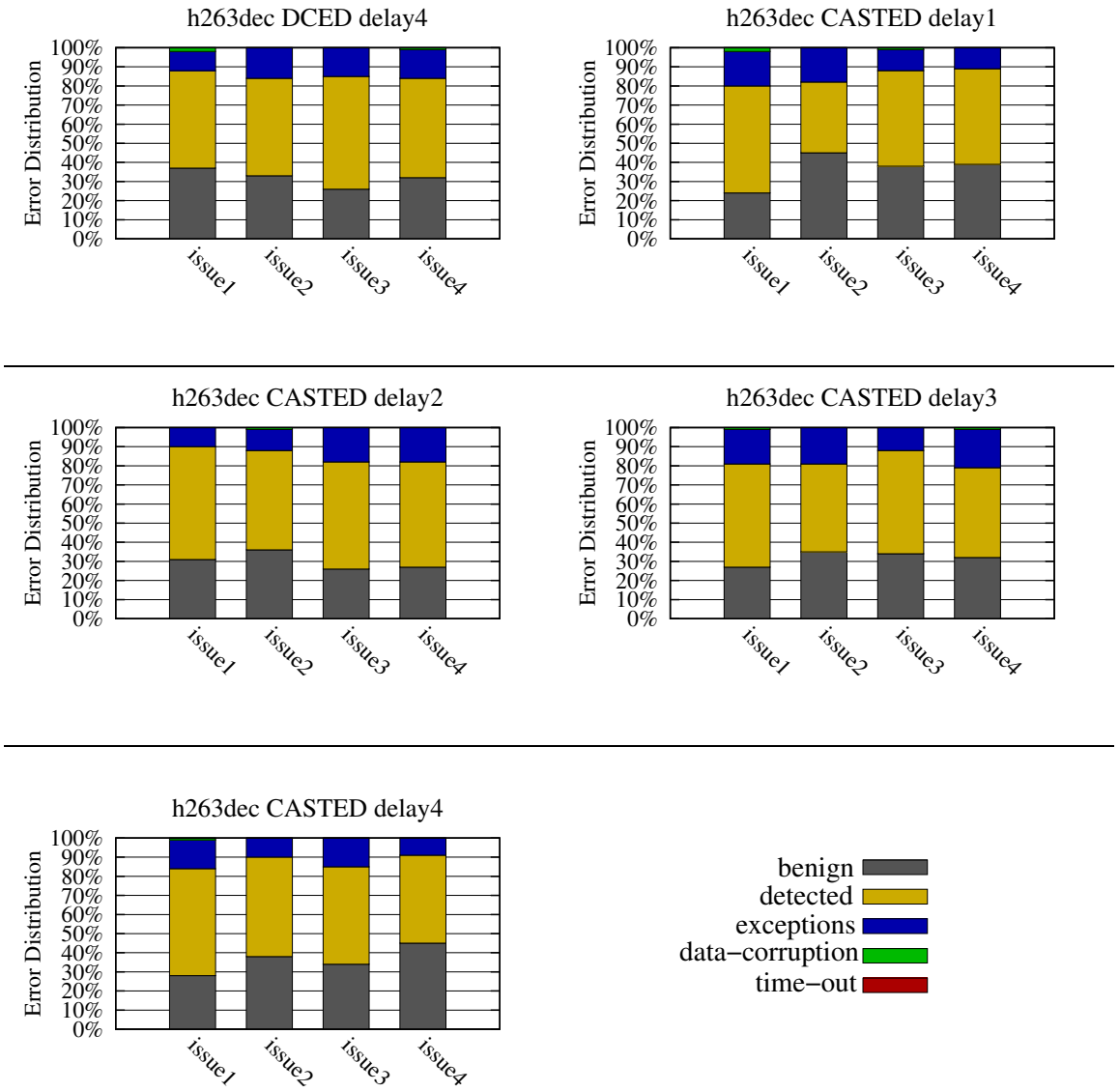


Figure 4.15: The fault-coverage of h263dec benchmark for NOED,SCED,DCED and CASTED for issue 1 to 4 and delay 1 to 4 (part 2).

# Chapter 5

# Related Work

## 5.1 Redundancy-based Error Detection

Code redundancy can take various forms: instruction, thread and process redundancy. The main **instruction-level** error detection methodologies were described in Section 2.2.2. In brief, EDDI [60] was the first to introduce thread-local redundancy. Next, SWIFT [70] improved performance by reducing the memory overhead. SRMT [91] inspired by redundant multi-threading error detection proposes a multi-threading technique that uses software checks instead of hardware ones. DAFT [101] improves further this technique by decoupling the execution of the original and the checker thread. In [17], the authors present triple-modular redundancy at instruction-level.

The proposed techniques of this work focus on improving the performance of instruction-level error detection with error-detection-aware instruction scheduling optimizations. On one hand, DRIFT reduces the performance overhead of instruction-level error detection by reducing the impact of checks on the control-flow. In this way, the compiler can optimize the code better during scheduling and generate code with more ILP. On the other hand, CASTED explores the mapping of instruction-level error detection on tightly-coupled cores. CASTED proposes a technique that improves the placement of the code for any architecture configuration (e.g. issue-width, communication latency). CASTED succeeds this with an improved error-detection-aware scheduling algorithm that considers both issue-width and inter-core latency. Table 5.1 summarizes the proposed techniques and compares them against the state-of-the-art.

**Hybrid** techniques produce the checker code in the same way as instruction-level error detection does. In addition, they use hardware support to do the checking and improve its efficiency further. In [71], extra hardware structures are used to improve

| Technique | Performance Overhead | Fault Coverage | Target Architecture | Performance Evaluation |
|---|---|---|---|---|
| EDDI [60] | 62% | Processor & Memory | MIPS | SGI Octane |
| SWIFT [70] | 41% | Processor | VLIW | Itanium 2 |
| Shoestring [25] | 15.8%-30.4% | Processor | x86 | Simulator |
| SRMT [91] | 400% | Processor | x86 | Simulator |
| DAFT [101] | 38% | Processor | x86 | Xeon X7460 |
| DRIFT | 29% | Processor | VLIW | Itanium 2 |
| CASTED | 58% | Processor | tightly-coupled cores | Simulator |

Table 5.1: The proposed techniques and the state-of-the-art instruction-level error detection techniques.

further the fault coverage. The two structures (Checking Store Buffer, Checking Load Buffer) increase the system's resilience against errors in memory instructions. In the scheme of [35], there are no software checks, and checking is done entirely in hardware. In addition, the authors propose two techniques which target to optimize the code for performance and power. The main drawback of this technique is that any performance or power gain comes by sacrificing fault coverage.

**Redundant multi-threading (RMT)** was introduced by Rotenberg in AR-SMT [73]. The main idea is that an exact replica of the original thread is created. The replicated (trailing) thread lags behind the original (leading) thread. The leading thread pushes the output of each instruction in a buffer. The leading thread checks the values of the buffer with the ones that it produces. To avoid the branch mis-predictions, the leading thread sends the branch outcomes to the trailing thread. [68] introduces the concept of sphere of replication. The sphere of replication determines the part of the system that is protected by a given technique. In [68], the authors exclude the memory subsystem from the sphere of replication and define the data that should be replicated and the data that should be compared.

Smolens [83] reduces the performance overhead of RMT by helping the two threads to efficiently share the instruction queue and the reorder buffer. In [61], the authors try to reduce the overhead of RMT by reducing the number of instructions in the trailing thread. In [29], the authors opportunistically enable redundancy when the performance

is not affected. For example, applications with low-ILP can accommodate more redundancy than the ones with high-ILP. To reduce the overhead of RMT, Mukherjee [56] proposed chip-level redundant multi-threading (CRT). In this approach, the leading and the trailing threads run on different cores. Similarly to [68][73], the leading thread sends to the trailing thread the values that are for checking. [39][62][72][82][97] present techniques where the redundant execution is diverted to idle cores.

RMT is also used for the detection of permanent errors. In CRT [56], the detection of permanent errors is possible since the two threads run on two different cores. [56] also proposes a technique to detect permanent faults in SMT processors. The authors propose preferential-space redundancy which schedules the instructions of the trailing and the leading threads for execution on different units. In a similar way, [76] shuffles the instructions of the two threads in order to make sure that they will be executed in different units.

The main disadvantage of redundant multi-threading is that it reduces the system's total throughput because it occupies more thread contexts and hardware resources. Additionally, compared to instruction-level approaches (where software queues are used for the communication between the threads), most of the redundant multi-threading schemes require custom hardware.

**Process** level redundancy (PLR) [80] replicates the processes of the application and compares their outputs to ensure correct execution. The processes synchronize to compare their outputs when the value escapes user space to the kernel. RAFT [100] improves this scheme by removing the synchronization barriers. PLR has small overhead since it checks fewer values than other approaches, but this comes at the cost of maintaining multiple memory states.

**Hardware-based redundancy** replicates hardware units. Hence, the whole system must be custom designed for fault-tolerance. Although this process is very expensive and less flexible than the ones described above, hardware-based approaches often suffer less performance degradation from fault tolerance.

Typical examples are the HP NonStop Advanced Architecture (NSAA) [9] and IBM's z series [22]. NSAA can be configured to run either dual-modular redundancy or triple-modular redundancy. The data are replicated two or three times and the replicas are executed in lockstep. The outcomes of the two or three units are compared in a checker or a voter respectively. In [22], the execution unit is replicated and the register file is protected by ECC and parity checking. After every instruction, a checkpoint is saved. In the presence of an error, the execution can be diverted to one of the eight

spare cores.

A more lightweight approach is DIVA [6]. DIVA introduced dynamic implementation verification architecture where a small simpler core (checker) executes the same instructions as the original core. The checker core does not have any performance improving structures such as branch prediction, reservation tables etc. The original core sends to the checker core the data and the opcode of the instruction that is about to be executed. In this way, the checker code verifies the execution of the original core.

The scheme in [47] introduced a watchdog processor which monitors the execution of the original processor. Contrary to DIVA, the watchdog processor does not execute the instructions of the program again, but it watches the execution of the program in the main processor and checks if some invariants (e.g., control-flow, memory accesses) are violated. Argus [50] proposes lightweight methods to check control-flow, instruction execution and memory accesses.

In [3], the authors propose an architecture which can reconfigure so as to isolate the error. In this way, the processor can continue the execution of the program without the erroneous component. In [65][66], the authors present a technique that takes advantage of the inherent time redundancy of the programs so as to protect the fetch and decode stages of the pipeline. [14] presents a technique to protect array structures (e.g., reorder buffer) against hard errors.

## 5.2 Symptom-based Error Detection

**Symptom-based** error detection tries to reduce the amount of redundancy by trading off performance and hardware resources against reliability. In [93][95], the authors observe that some transient errors result in symptoms such as exceptions. Therefore, they propose a hardware mechanism that detects these symptoms instead of using redundancy for error detection. This is enough to increase by 2x the MTBF (mean time between failures). The symptoms are classified in the following categories:

1. ISA-defined exceptions: these are the exceptions defined by the instruction set architecture (ISA) (e.g., overflow).

2. Incorrect control flow: A transient error might lead to the execution along the wrong path. In this case, the branch is mis-predicted. For this reason, mis-predicted branches are another symptom of transient errors.

3. Detecting memory instruction address differences: An error in the upper bits of the address of a store instruction might result in writing to a memory area that the program does not have access. In this case, an exception is raised. A transient error in the lower bits of the address might result in a cache miss since the requested block is not in the cache. Therefore, cache misses are also considered symptoms of transient errors.

Restore [93] proposes an architecture that detects these symptoms and recovers from them using a checkpoint mechanism. Shoestring [25] uses static analysis in order to figure out the instructions that can produce ISA-defined exceptions. The scheme excludes these instructions from replication while the program's remaining instructions are replicated (as SWIFT [70]). In [64], the authors extend the symptoms catalog by proposing to verify data value ranges and data bit invariants. In [41][74][75], the authors introduce symptom-based error detection on the diagnosis of permanent errors.

Symptom-based error detection is a low-cost alternative to redundancy, but the lower fault-coverage limits its scope to systems where reliability is not as critical. For instance, such systems are those that use approximate computing. In this design paradigm, application's correctness is sacrificed in favor of better performance and lower power consumption.

## 5.3 Error Resilient Applications

The above suggest that redundancy is expensive in terms of performance or hardware resources. For this reason, techniques that reduce the amount of redundancy are desirable. In [94], the authors observe that up to 85% of the injected errors in the memory subsystem and 88% of them in the computational logic are masked errors. Similarly, in [11], the authors show that the microarchitectural masking is 6.47% and the architectural-masking is 88.35%. For instance, a bit-flip in a speculative instruction that will not commit, will not have any impact on application's correctness. In [92], the authors show that 40% of dynamic branches and 50% of mis-predicted branches do not have any impact on program's correctness when forced down to the wrong path. For example, encoding benchmarks have different levels of compression which are implemented with a loop. In this case, an error in a loop invariant might result in a few more (or less) iterations. Most probably, this error will not affect application's correctness.

In addition to the above, more studies [4][20][40][42][43][52][98] show that there is a large number of applications that have inherent resilience to transient errors. Example of these applications are audio and video processing, Bayesian inference, cellular automata, neural networks and hyper encryption. In Sections 3.3.3 and 4.4.3, we showed that the encoding benchmarks from Mediabench suite have an increased number of masked errors.

Mukherjee [57] first introduced the *architectural vulnerability factor (AVF)*. This metric gives the probability of an error in a processor structure to corrupt the output of the program. For example, an error in the branch predictor will not affect the committed instructions. Hence, the AVF for the branch predictor is 0%. On the other hand, a bit-flip in the program counter will change the execution sequence of the instructions. Thus, the AVF for the program counter is 100%. The AVF for most of the processor structures will range between these two extremes. The sum of each structure's AVF is the processor's AVF. The error rate of each structure is the product of raw fault rate and its AVF. The raw fault rate is defined by the manufacturing processes and the environmental factors. Therefore, a processor's error rate could be calculated by summing up all these products.

*Timing vulnerability factor (TVF)* is another vulnerability factor. TVF represents the fraction of each cycle where a bit affects the correctness of the program (architecturally correct execution bit (ACE bit)). For example, the TVF of RAM cells is 100%. Latches hold the data for 50% of the time and they drive data for the rest of the time. Hence, the TVF of a latch is 50% since the first process is the vulnerable one. In [77], the authors show that the TVF of a latch might be less than 50% since a strike late in the hold phase may not have enough time to propagate. For simplicity in AVF analysis, it is assumed [57][55] that TVF is part of the raw fault rate.

Some bits that are critical for the correctness of the execution at architecture level, are critical for the program's correctness. To measure those bits, the authors of [87] define the *program vulnerability factor (PVF)*. The PVF of a bit is the fraction of time (number of instructions) where this bit is an ACE bit. For example, consider a program that has an add instruction whose outcome is the input of a shift instruction. The latter one results in discarding the upper bit of the input data. Hence, a bit-flip in the upper-bit at the output of the ALU will never affect the execution of the program. Thus, the PVF of this bit is 0%. This bit is an ACE bit for the calculation of the AVF, but is un-ACE for PVF. Therefore, PVF can be extracted from AVF by eliminating the architecture-level masking. The PVF of an architecture resource changes if the binary

or the input of the program changes.

Taking into consideration that some errors do not manifest to the output of the program, Weaver [96] presents a technique which aims to find the benign errors and reduce them. This technique is based on locating instructions such as dead instructions or instruction types that are neutral to errors (e.g., NOP instructions). In [10], the authors calculate the AVF of address-based structures.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions

As technology and voltage scales, there is an increased need for low-overhead and high reliability error detection methodologies since transistors become more vulnerable to transient events. Instruction-level error detection is flexible since it can be easily applied to any part of the program. In addition, it does not need any hardware extensions. Hence, it can be used at any system without custom hardware for error detection. In this thesis, we worked on reducing the performance overhead of instruction-level error detection without decreasing the fault-coverage. We presented DRIFT and CASTED which address the performance bottlenecks of instruction-level error detection and optimize them.

In DRIFT, we proved that the checks are the main slowdown factor of instruction-level error detection. The checks are compare and jump instructions. This sequence of instructions make the code sequential. Moreover, the jump instructions (due to checks) act as scheduling barriers prohibiting the compiler from applying aggressive code motion optimizations. We named this side-effect basic-block fragmentation. The frequent checking makes basic-block fragmentation more intense and the scheduling worse. DRIFT deals with this problem by decoupling the execution of the original and replicated code from the checks. The latter ones are grouped together. In this way, the compiler can generate code with more ILP. This optimization reduces the error detection overhead down to 1.29x (on average). DRIFT outperforms the state-of-the-art (SWIFT) by up to 29.7% without any impact on fault-coverage.

CASTED optimizes the error detection code for tightly-coupled core architectures. The main characteristic of this architecture is that the communication delay between

the cores is a few cycles. Therefore, the error detection code can be executed in any of these cores in order to exploit all the available ILP. Current state-of-the-art techniques do not adapt well to different architecture configurations such as the instruction width and the inter-core communication. The single-core technique does not fully benefit from the available resources and the dual-core technique suffers from the communication penalty. CASTED presents an algorithm that distributes the error detection overhead across the cores in such a way as to reduce the error detection overhead. The CASTED algorithm achieves this by taking into consideration the available resources, the inter-core delay and the data-flow graph. As a result, CASTED manages to map the error detection code to different architecture configurations. For each one of them, it performs as good well as the best performing state-of-the-art technique. CASTED generates better code and outperforms the state-of-the-art in some cases. It reduces the error detection overhead of the single-core technique by 7.5% and the overhead of dual-core technique by 24.7%.

## 6.2 Future Work

The proposed techniques study the overhead of instruction-level error detection on VLIW and clustered-VLIW architectures. The behavior of instruction-level error detection might be different on architectures like x86 due to two reasons: i. number of architectural registers and ii. issue-width.

1. Itanium 2 has many more architectural registers compared to x86. Itanium 2 has 128 general purpose registers while x86 has 16. The error detection duplicates the register pressure because the replicated instructions have their own registers. The checks compare the values of the registers of the original and the replicated instructions. As a result, this might become a serious performance bottleneck for x86 architectures.

2. Instruction-level error detection also increases ILP by almost a factor of 2 because of the replicated instructions. For this reason, wide-issue machines like the Itanium 2 (6-issue) can handle this workload better.

The following proposals show how the overhead of instruction-level error detection can be decreased for x86 architectures.

### 6.2.1 Redundant Multi-threading Performance Optimizations

The study of CASTED suggests that the communication latency is a bottleneck for the dual-core technique. In addition, it showed that a processor with large enough issue-width can accommodate the error detection overhead. As a next step, we will study the trade-off between thread-local and redundant multi-threading (as it was discussed in CASTED) for commodity multi-core processors using pthreads and a software communication queue. For these architectures, Thread Level Parallelism (TLP) should be considered as another dimension in the trade-off space.

Redundant multi-threading uses extra threads for the execution of the checker code. If the system has many cores, than the error detection will have small impact on the original execution. However, if the available cores are not many, than the original execution might be delayed because of the checker threads. As a result, redundant multi-threading can potentially harm performance because it might consume resources that could be used for increasing the throughput of a scalable multi-threaded application. On the other hand, thread-local error detection does not affect system's throughput, but it can only be efficient of the available cores are wide enough.

Figure 6.1 shows two examples that explain the trade-off between redundant multi-threading and thread-local error detection. We assume an architecture with four single-threaded cores. The first example (Figure 6.1.a-6.1.c) presents an application that scales to four threads and the second example (Figure 6.1.d-6.1.f) shows an application that scales to two threads.

In case of redundant multi-threading, the original threads of the first example (Figure 6.1.b) require four extra threads for the checking code. However, the given architecture does not support eight threads. Consequently, two original and two checker threads can be executed at the same time (Figure 6.1.b). As a result, the application cannot fully scale and its execution is delayed. On the other hand, redundant multi-threading is very efficient for the second example (Figure 6.1.e). The original execution does not have to share resources with the checker threads. In this example, the overhead of redundant multi-threading has only to do with the communication between the original and the checker threads.

Thread-local error detection is applied to each thread of the multi-threaded application. Thus, each thread has the overhead of thread-local error detection (Figure 6.1.c and 6.1.f). If the cores are wide enough, this overhead might not big. As it was shown in CASTED, thread-local error detection is more efficient on cores with high ILP. By

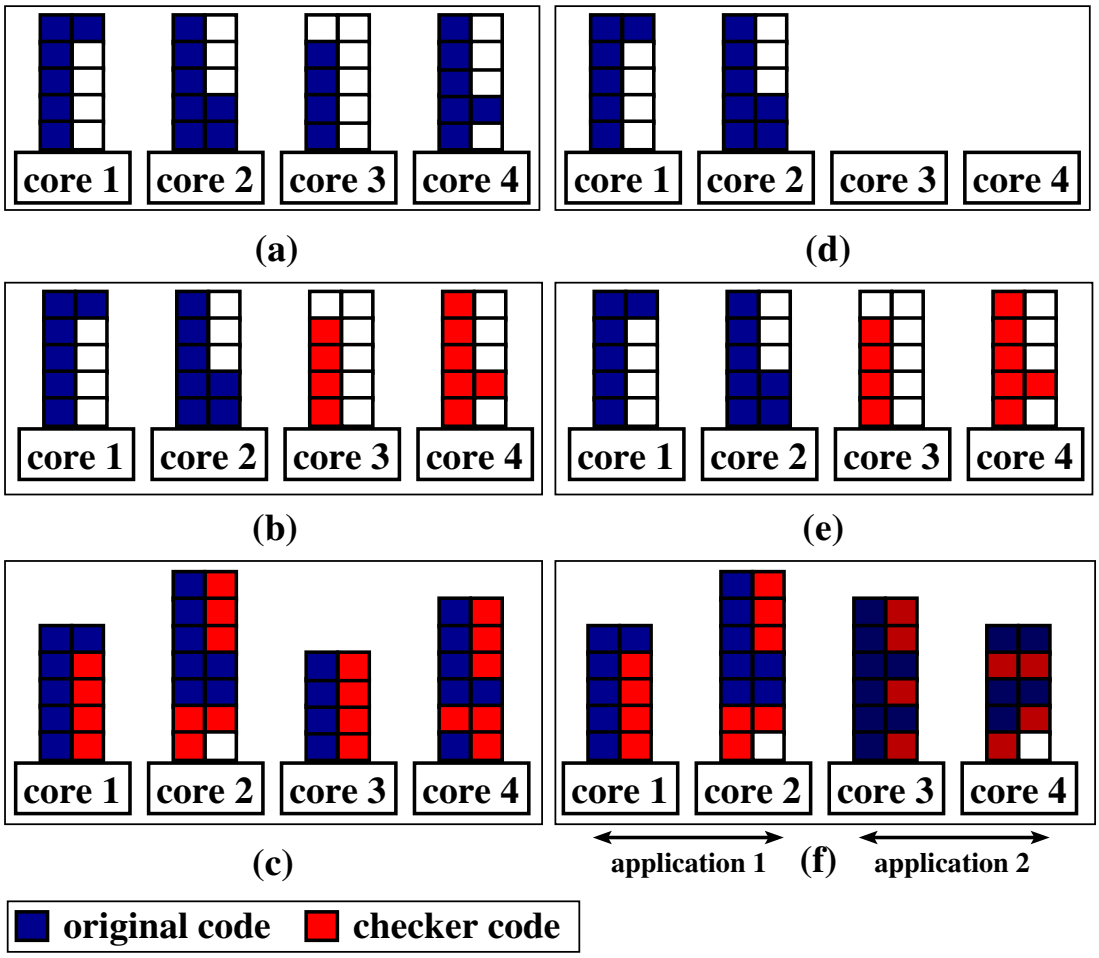


Figure 6.1: This figure shows the trade-offs between redundant multi-threading and thread-local error detection. (a)-(c) refer to an application that scales in four threads. (b) shows the impact of redundant multi-threading on the execution of the application. Due to the checker threads, the application can only use half of the resources. In (c), thread-local error detection delays the execution of each thread, but the application can fully benefit from all the resources. (d)-(f) present an application that scales in two threads. In this case, the redundant multi-threading error detection (e) does not have negative impact on performance since there are spare resources for the checker threads. On the other hand, thread-local error detection (f) increases system's throughput. The spare cores (3 and 4) can be used by another application.

comparing Figures 6.1.b and 6.1.c, we can see that each thread in 6.1.b executes faster than the threads of 6.1.c. But, the application scales to more cores in 6.1.c and gains more speedup. Finally, Figure 6.1.f shows that the thread-local scheme can increase system's throughput by letting another application use the spare cores. In Figure 6.1.f, application 1 scales to two threads and uses thread-local error detection (core 1 and 2) while the free cores (3 and 4) can be used by application 2.

From the above, we conclude that single-threaded applications may sometimes benefit from redundant multi-threading, depending on the core sizes and the communication latency, as shown in this thesis. However, it is not straightforward to identify which scheme fits best to a multi-threaded application. Applications with TLP will be slowed-down by redundant multi-threading if there are not enough cores available. In this case, a thread-local scheme might be preferred. Moreover, applications with high ILP might perform poorly using the thread-local scheme. Therefore, an adaptive mechanism that takes into consideration all of the above is needed.

### 6.2.2 Instruction-level Triple-modular Redundant Error Detection

Instruction-level error detection can be extended to triple-modular redundancy. Figure 6.2 shows how TMR is implemented at instruction-level. The original instruction is replicated two times. Next, a sequence of checks (voter) discards the erroneous value and propagates the correct value to the rest of the execution. This is done by copying the correct value to the erroneous register. In this way, TMR manages to do error detection and recovery at the same time.

In [17], the authors show that instruction-level TMR has 100% overhead. In Figure 6.2, it is shown that the replicas can be executed in parallel since there is no dependency between them. Therefore, the impact of the three replicas can be hidden by wide processors. But, the long sequence of checks results in intensive basic-block fragmentation. The voting system should be added in the code with the same frequency as the checks in dual-modular error detection (SWIFT [70]). As a result, the voter fragments the code even more than the checks in dual-modular error detection. In addition, a voter breaks the original basic-block in three smaller ones. In dual-modular error detection, a check breaks the basic-block into two pieces. Thus, the compiler's job is now even harder. DRIFT's decoupling capability would be helpful in this case.

To reduce further TMR's overhead, vectorization could be applied. TMR's replicas are the perfect candidates for vectorization. In addition, this will extend our scheme on

the detection of permanent errors. In [17], the replicas might be scheduled to execute on the same unit. However, in the case of vectorization, each replica will be forced to run on a different unit (scalar or vector).

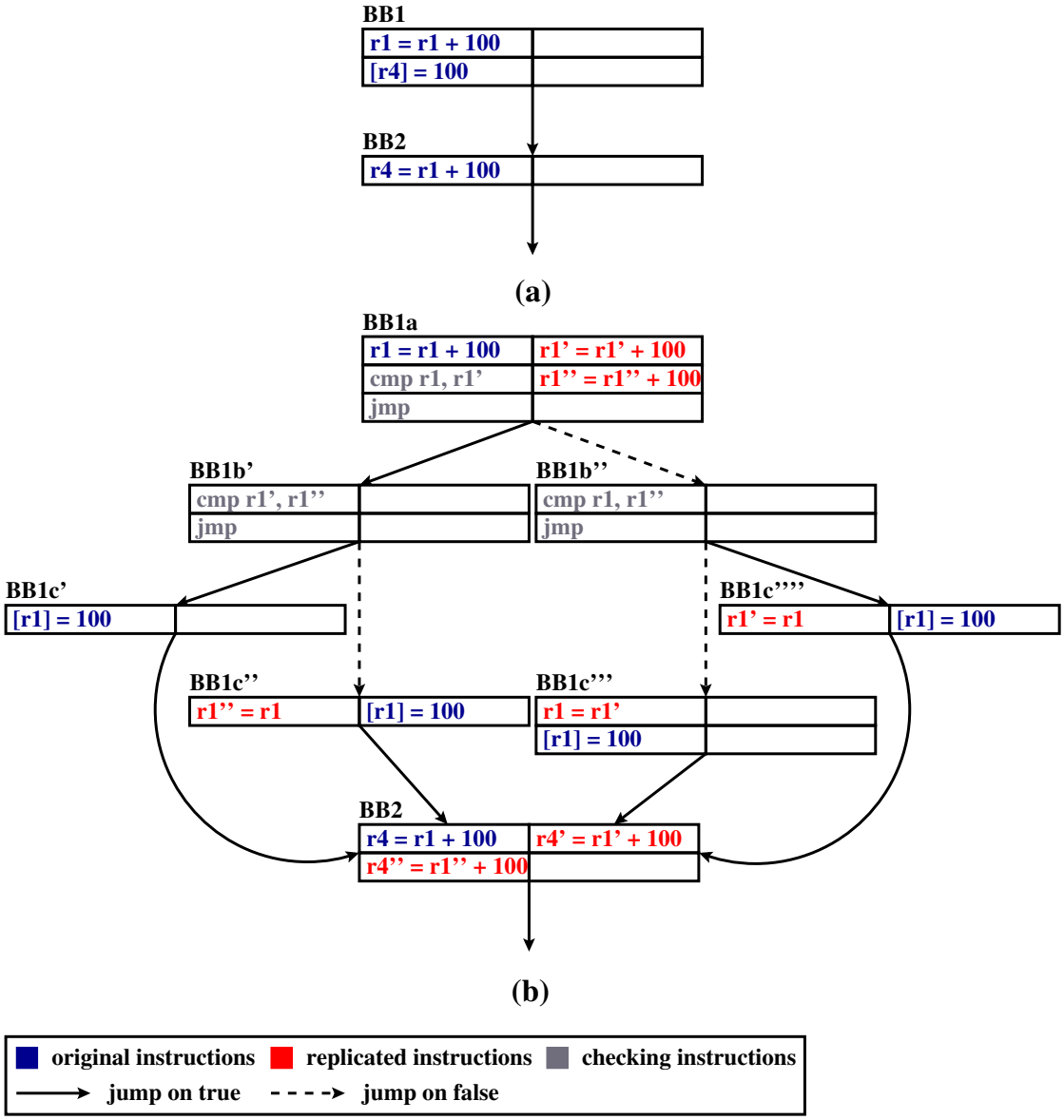


Figure 6.2: (a) Original code, (b) Code after instruction-level triple-modular redundant error detection and correction.

# Bibliography


[1] GCC: GNU Compiler Collection. *http://gcc.gnu.org*.

[2] SKI, An IA64 Instruction Set Simulator. *http://ski.sourceforge.net*.

[3] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable Isolation: Building High Availability Systems with Commodity Multi-core Processors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 470–481, New York, NY, USA, 2007. ACM.

[4] B. E. S. Akgul, L. Chakrapani, P. Korkmaz, and K. Palem. Probabilistic CMOS Technology: A Survey and Future Directions. In *Very Large Scale Integration, 2006 IFIP International Conference on*, pages 1–6, Oct 2006.

[5] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3-GHz Fifth-generation SPARC64 Microprocessor. *Solid-State Circuits, IEEE Journal of*, 38(11):1896–1905, Nov 2003.

[6] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 196–207, Washington, DC, USA, 1999. IEEE Computer Society.

[7] R. Baumann. Radiation-induced Soft Errors in Advanced Semiconductor Technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305–316, Sept 2005.

[8] R. Baumann. Soft Errors in Advanced Computer Systems. *Design Test of Computers, IEEE*, 22(3):258–266, May 2005.

[9] D. Bernick, B. Bruckert, P. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop Advanced Architecture. In *Dependable Systems and*

*Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 12–21, June 2005.

[10] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. Computing Architectural Vulnerability Factors for Address-Based Structures. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 532–543, Washington, DC, USA, 2005. IEEE Computer Society.

[11] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke. Cost-efficient Soft Error Protection for Embedded Microprocessors. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 421–431, New York, NY, USA, 2006. ACM.

[12] S. Borkar. Microarchitecture and Design Challenges for Gigascale Integration. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 3–3, Washington, DC, USA, 2004. IEEE Computer Society.

[13] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *Micro, IEEE*, 25(6):10–16, Nov 2005.

[14] F. Bower, P. Shealy, S. Ozev, and D. Sorin. Tolerating Hard Faults in Microprocessor Array Structures. In *Dependable Systems and Networks, 2004. International Conference on*, pages 51–60, June 2004.

[15] A. Branover, D. Foley, and M. Steinman. AMD Fusion APU: Llano. *Micro, IEEE*, 32(2):28–37, March 2012.

[16] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, pages 292–300, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[17] J. Chang, G. Reis, and D. August. Automatic Instruction-Level Software-Only Recovery. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 83–92, June 2006.

[18] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule. Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications. *Micro, IEEE*, 34(2):34–43, Mar 2014.

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

[20] M. De Kruijf and K. Sankaralingam. Exploring the Synergy of Emerging Workloads and Silicon Reliability Trends. *Workshop on Silicon Errors in Logic - System Effects*, 2009.

[21] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. Yale University, 1985.

[22] M. L. Fair, C. R. Conklin, S. Swaney, P. Meaney, W. Clarke, L. Alves, I. N. Modi, F. Freier, W. Fischer, and N. E. Weber. Reliability, Availability, and Serviceability (RAS) of the IBM eServer Z990. *IBM Journal of Research and Development*, 48(3.4):519–534, May 2004.

[23] P. Faraboschi, G. Desoli, and J. A. Fisher. *Clustered Instruction-level Parallel Processors*. Hewlett Packard Laboratories, 1999.

[24] P. Faraboschi and F. Homewood. ST200: A VLIW Architecture for Media-oriented Applications. In *Microprocessor Forum*, pages 9–13, 2000.

[25] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 385–396, New York, NY, USA, 2010. ACM.

[26] J. A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Elsevier, 2005.

[27] J. A. Fisher, P. Faraboschi, and C. Young. VLIW Processors. In *Encyclopedia of Parallel Computing*, pages 2135–2142. Springer, 2011.

[28] J. E. Fritts, F. W. Steiling, and J. A. Tucek. Mediabench II Video: Expediting the Next Generation of Video Systems Research. In *Electronic Imaging 2005*, pages 79–93. International Society for Optics and Photonics, 2005.

[29] M. A. Gomaa and T. N. Vijaykumar. Opportunistic Transient-Fault Detection. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 172–183, Washington, DC, USA, 2005. IEEE Computer Society.

[30] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai. Impact of CMOS Process Scaling and SOI on the Soft Error Rates of Logic Processes. In *VLSI Technology, 2001. Digest of Technical Papers. 2001 Symposium on*, pages 73–74, June 2001.

[31] W. Havanki, S. Banerjia, and T. Conte. Treegion Scheduling for Wide Issue Processors. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 266–276, Feb 1998.

[32] P. Hazucha, C. Svensson, and S. Wender. Cosmic-ray Soft Error Rate Characterization of a Standard 0.6-/SPL MU/M CMOS Process. *Solid-State Circuits, IEEE Journal of*, 35(10):1422–1429, Oct 2000.

[33] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2012.

[34] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, Jul 2000.

[35] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-assisted Soft Error Detection Under Performance and Energy Constraints in Embedded Systems. *ACM Transanctions on Embedded Computing Systems (TECS)*, 8(4):27:1–27:30, July 2009.

[36] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *the Journal of Supercomputing*, 7(1-2):229–248, 1993.

[37] T. Instruments. TMS320C6000 CPU and Instruction Set Reference Guide. *Texas Instruments Journal*, 2000.

[38] T. Karnik and P. Hazucha. Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes. *Dependable and Secure Computing, IEEE Transactions on*, 1(2):128–143, April 2004.

[39] C. LaFrieda, E. Ipek, J. Martinez, and R. Manohar. Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 317–326, June 2007.

[40] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra. ERSA: Error Resilient System Architecture for Probabilistic Applications. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1560–1565, March 2010.

[41] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 265–276, New York, NY, USA, 2008. ACM.

[42] X. Li and D. Yeung. Exploiting Soft Computing for Increased Fault Tolerance. *ASGI*, 2006.

[43] X. Li and D. Yeung. Application-Level Correctness and its Impact on Fault Tolerance. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 181–192, Feb 2007.

[44] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. Lichtenstein, R. P. Nix, J. S. O'donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *The journal of Supercomputing*, 7(1-2):51–142, 1993.

[45] S. A. Mahlke, W. Y. Chen, W.-m. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel Scheduling for VLIW and Superscalar Processors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 238–247, New York, NY, USA, 1992. ACM.

[46] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[47] A. Mahmood and E. McCluskey. Concurrent Error Detection Using Watchdog Processors - A Survey. *Computers, IEEE Transactions on*, 37(2):160–174, Feb 1988.

[48] T. May and M. H. Woods. Alpha-particle-induced Soft Errors in Dynamic Memories. *Electron Devices, IEEE Transactions on*, 26(1):2–9, Jan 1979.

[49] C. McNairy and D. Soltis. Itanium 2 Processor Microarchitecture. *Micro, IEEE*, 23(2):44–55, March 2003.

[50] A. Meixner, M. E. Bauer, and D. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 210–222, Washington, DC, USA, 2007. IEEE Computer Society.

[51] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender. Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer. *Device and Materials Reliability, IEEE Transactions on*, 5(3):329–335, Sept 2005.

[52] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of Service Profiling. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 25–34, May 2010.

[53] S. Moon and K. Ebcioğlu. Parallelizing Non-numerical Code with Selective Scheduling and Software Pipelining. *Transactions on Programming Languages and Systems*, 1997.

[54] S.-M. Moon and K. Ebcioğlu. An Efficient Resource-constrained Global Scheduling Technique for Superscalar and VLIW Processors. In *Microarchitecture, 1992. MICRO 25., Proceedings of the 25th Annual International Symposium on*, pages 55–71, Dec 1992.

[55] S. Mukherjee, J. Emer, and S. Reinhardt. The Soft Error Problem: An Architectural Perspective. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 243–247, Feb 2005.

[56] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of the 29th Annual*

*International Symposium on Computer Architecture*, ISCA '02, pages 99–110, Washington, DC, USA, 2002. IEEE Computer Society.

[57] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 29–, Washington, DC, USA, 2003. IEEE Computer Society.

[58] T. O'Gorman, J. M. Ross, A. H. Taber, J. Ziegler, H. Muhlfeld, C. Montrose, H. W. Curtis, and J. Walsh. Field Testing for Cosmic Ray Soft Errors in Semiconductor Memories. *IBM Journal of Research and Development*, 40(1):41–50, Jan 1996.

[59] N. Oh, P. Shirvani, and E. McCluskey. Control-flow Checking by Software Signatures. *Reliability, IEEE Transactions on*, 51(1):111–122, Mar 2002.

[60] N. Oh, P. Shirvani, and E. McCluskey. Error Detection by Duplicated Instructions in Super-scalar Processors. *Reliability, IEEE Transactions on*, 51(1):63–75, Mar 2002.

[61] A. Parashar, A. Sivasubramaniam, and S. Gurumurthi. SlicK: Slice-based Locality Exploitation for Efficient Redundant Multithreading. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 95–105, New York, NY, USA, 2006. ACM.

[62] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. Architectural Core Salvaging in a Multi-core Processor for Hard-error Tolerance. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 93–104, New York, NY, USA, 2009. ACM.

[63] M. D. Powell and T. N. Vijaykumar. Pipeline Damping: A Microarchitectural Technique to Reduce Inductive Noise in Supply Voltage. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 72–83, New York, NY, USA, 2003. ACM.

[64] P. Racunas, K. Constantinides, S. Manne, and S. Mukherjee. Perturbation-based Fault Screening. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 169–180, Feb 2007.

[65] V. Reddy and E. Rotenberg. Inherent Time Redundancy (ITR): Using Program Repetition for Low-Overhead Fault Tolerance. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 307–316, June 2007.

[66] V. Reddy and E. Rotenberg. Coverage of a Microarchitecture-level Fault Check Regimen in a Superscalar Processor. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 1–10, June 2008.

[67] K. Reick, P. Sanda, S. Swaney, J. Kellington, M. Mack, M. Floyd, and D. Henderson. Fault-Tolerant Design of the IBM Power6 Microprocessor. *Micro, IEEE*, 28(2):30–38, March 2008.

[68] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 25–36, New York, NY, USA, 2000. ACM.

[69] G. A. Reis. *Software Modulated Fault Tolerance*. Princeton University, 2008.

[70] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[71] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and Evaluation of Hybrid Fault-Detection Systems. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 148–159, Washington, DC, USA, 2005. IEEE Computer Society.

[72] B. F. Romanescu and D. J. Sorin. Core Cannibalization Architecture: Improving Lifetime Chip Performance for Multicore Processors in the Presence of Hard Faults. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 43–51, New York, NY, USA, 2008. ACM.

[73] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 84–91, June 1999.

[74] S. Sahoo, M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou. Using Likely Program Invariants to Detect Hardware Errors. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 70–79, June 2008.

[75] S. K. Sastry Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve. mSWAT: Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 122–132, New York, NY, USA, 2009. ACM.

[76] E. Schuchman and T. N. Vijaykumar. BlackJack: Hard Error Detection with Redundant Threads on SMT. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 327–337, June 2007.

[77] N. Seifert and N. Tam. Timing Vulnerability Factors of Sequentials. *Device and Materials Reliability, IEEE Transactions on*, 4(3):516–522, Sept 2004.

[78] H. Sharangpani and H. Arora. Itanium Processor Microarchitecture. *Micro, IEEE*, 20(5):24–43, Sep 2000.

[79] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389–398, 2002.

[80] A. Shye, T. Moseley, V. Reddi, J. Blomstedt, and D. Connors. Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 297–306, June 2007.

[81] T. Slegel, I. Averill, R.M., M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb. IBM's S/390 G5 microprocessor design. *Micro, IEEE*, 19(2):12–23, Mar 1999.

[82] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-Effective Multicore Redundancy. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 223–234, Washington, DC, USA, 2006. IEEE Computer Society.

[83] J. C. Smolens, J. Kim, J. C. Hoe, and B. Falsafi. Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 257–268, Washington, DC, USA, 2004. IEEE Computer Society.

[84] D. J. Sorin. Fault Tolerant Computer Architecture. *Synthesis Lectures on Computer Architecture*, 4(1):1–104, 2009.

[85] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 123–134, Washington, DC, USA, 2002. IEEE Computer Society.

[86] L. Spainhower and T. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(5.6):863–873, Sept 1999.

[87] V. Sridharan and D. Kaeli. Eliminating Microarchitectural Dependency from Architectural Vulnerability. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 117–128, Feb 2009.

[88] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Dependable Systems and Networks, 2004 International Conference on*, pages 177–186, June 2004.

[89] A. Suga and K. Matsunami. Introducing the FR500 Embedded Microprocessor. *Micro, IEEE*, 20(4):21–27, Jul 2000.

[90] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The RAW Microprocessor: A Computational Fabric for Software Circuits and General-purpose Programs. *Micro, IEEE*, 22(2):25–35, Mar 2002.

[91] C. Wang, H.-s. Kim, Y. Wu, and V. Ying. Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society.

[92] N. Wang, M. Fertig, and S. Patel. Y-Branches: When You Come to a Fork in the Road, Take It. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 56–, Washington, DC, USA, 2003. IEEE Computer Society.

[93] N. Wang and S. Patel. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):188–201, July 2006.

[94] N. Wang, J. Quek, T. Rafacz, and S. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Dependable Systems and Networks, 2004 International Conference on*, pages 61–70, June 2004.

[95] N. J. Wang, A. Mahesri, and S. J. Patel. Examining ACE Analysis Reliability Estimates Using Fault-injection. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 460–469, New York, NY, USA, 2007. ACM.

[96] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 264–, Washington, DC, USA, 2004. IEEE Computer Society.

[97] P. M. Wells, K. Chakraborty, and G. S. Sohi. Mixed-mode Multicore Reliability. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 169–180, New York, NY, USA, 2009. ACM.

[98] V. Wong and M. Horowitz. Soft Error Resilience of Probabilistic Inference Applications. *Workshop on Silicon Errors in Logic - System Effects*, 2006.

[99] Y. Yeh. Triple-triple Redundant 777 Primary Flight Computer. In *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, volume 1, pages 293–307 vol.1, Feb 1996.

[100] Y. Zhang, S. Ghosh, J. Huang, J. W. Lee, S. A. Mahlke, and D. I. August. Runtime Asynchronous Fault Tolerance via Speculation. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 145–154, New York, NY, USA, 2012. ACM.

[101] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August. DAFT: Decoupled Acyclic Fault Tolerance. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 87–98, New York, NY, USA, 2010. ACM.

[102] H. Zhong, S. Lieberman, and S. Mahlke. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 25–36, Feb 2007.

[103] J. Ziegler. Terrestrial Cosmic Ray Intensities. *IBM Journal of Research and Development*, 42(1):117–140, Jan 1998.

[104] J. Ziegler, H. W. Curtis, H. Muhlfeld, C. Montrose, B. Chin, M. Nicewicz, C. A. Russell, W. Y. Wang, L. B. Freeman, P. Hosier, L. E. LaFave, J. Walsh, J. M. Orro, G. J. Unger, J. M. Ross, T. O'Gorman, B. Messina, T. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, and C. W. Wahaus. IBM Experiments in Soft Fails in Computer Electronics (1978-1994). *IBM Journal of Research and Development*, 40(1):3–18, Jan 1996.