Fault-Tolerant Coarse-Grained Data-Flow Execution

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
der Fakultät für Angewandte Informatik
der Universität Augsburg

Universität
Augsburg
University

eingereicht von
Dipl.-Inf. Sebastian Weis

Fault-Tolerant Coarse-Grained Data-Flow Execution
Sebastian Weis

Erstgutachter:	Prof. Dr. Theo Ungerer
Zweitgutachter:	Prof. Dr.-Ing. Rudi Knorr
Tag der mündlichen Prüfung:	26. November 2015

Kurzfassung

Das Fortschreiten der Halbleitertechnologie und die damit verbundene Steigerung der Transistorintegrationsdichte hat die Entwicklung immer leistungsfähigerer Multicore-Prozessoren maßgeblich vorangetrieben. Bedingt durch die stetige Verkleinerung der Strukturgrößen in Richtung der physikalischen Grenzen steigen aber auch die Wahrscheinlichkeiten von transienten, intermittierenden und permanenten Fehlern in den Prozessor-Chips. Diese Entwicklung macht effiziente Fehlertoleranzmechanismen in Zukunft auch für universale Multicore-Prozessoren notwendig. Allerdings ist die redundante Programmausführung, wie sie in sicherheitskritischen und hochverfügbaren Lockstep-Systemen seit langer Zeit verwendet wird, für gegenwärtige Universalprozessoren aufwendig, da Änderungen an der Mikroarchitektur benötigt werden, welche die Skalierbarkeit der Systeme einschränken können. Zusätzlich wird die Verwendung moderner Energiesparmaßnahmen und die Ausführung paralleler Anwendungen durch die Lockstep-Implementierung erschwert.

Die vorliegende Dissertation beschreibt die Integration von flexiblen Fehlertoleranzmechanismen in ein grobkörniges Datenflussausführungsmodell, welche insbesondere die skalierbare Ausführung von parallelen Anwendungen in einem Multicore-Prozessor erhalten sollen. Zur Fehlererkennung wird die entkoppelte redundante Ausführung von Datenfluss-Threads (Double Execution) vorgestellt, welche die Korrektur von Fehlern durch Thread-Neustarts unterstützt. Dabei werden notwendige Anpassungen am Datenflussausführungsmodell und der Hardware-Architektur sowie Techniken zur Eingabereplikation, zur Thread-Synchronisierung und zum Ausgabevergleich beschrieben. Basierend auf der redundanten Ausführung von Datenfluss-Threads wird eine Methode zur Diagnose von permanenten und intermittierenden Fehlern vorgeschlagen, die im Fehlerfall eine Anpassung des Systems ermöglicht. Um die Parallelität der redundanten Datenflussausführung zu erhöhen, wird die optimistische redundante Ausführung von Datenfluss-Threads (Optimistic Double Execution) vorgestellt, welche nachfolgende Datenfluss-Threads spekulativ starten kann, noch bevor die Korrektheit der Berechnungsergebnisse verifiziert werden konnte. Um die Fehlerkorrektur auch für die optimistische redundante Ausführung zu ermöglichen, wird ein datenflussbasierter globaler Wiederherstellungsmechanismus vorgestellt.

Die datenflussbasierten Fehlertoleranzmechanismen wurden mit Hilfe des Open-Source-Multicore-Simulators COTSon untersucht. Dabei zeigte sich, dass die redundante Thread-Ausführung nur geringe zusätzliche Laufzeitkosten gegenüber einer optimalen Lockstep-Maschine aufweist und in bestimmten Fällen sogar Laufzeitvorteile bieten kann. Im Falle von permanenten Fehlern in den Prozessorkernen zeigte sich, dass die Ausführungsgeschwindigkeit des Systems schrittweise angepasst werden kann. Des Weiteren können transiente Fehler durch den Neustart von Datenfluss-Threads mit geringen Kosten korrigiert werden. Für die optimistische redundante Ausführung zeigte sich, dass Anwendungen mit geringer Parallelität ungenutzte Systemressourcen zur Beschleunigung der redundanten Ausführung nutzen können.

Abstract

The progress of the semiconductor technology and the resulting increase of the transistor integration density has driven the development of ever more powerful multi-core processors. The reduction of the feature sizes in the direction of the physical limits increases the probabilities of transient, intermittent, and permanent faults in the chips and demands efficient fault-tolerance mechanisms also for future general-purpose multi-core processors. However, the use of redundant execution mechanisms, as they have been implemented in safety-critical and high-availability lockstep systems for a long time, can be expensive in general-purpose processors, since they require changes to the microarchitecture, which can limit the scalability of the systems. Additionally, lockstep execution complicates the use of modern power management mechanisms as well as the execution of parallel applications.

This thesis describes the integration of flexible fault-tolerance mechanisms in a coarse-grained data-flow execution model, which are able to preserve the scalable execution of parallel applications in a multi-core system. For fault detection, we present double execution, the decoupled redundant execution of data-flow threads, which supports fault recovery by the restart of data-flow threads. In this context, necessary adaptions to the data-flow execution model and the hardware architecture as well as techniques for input replication, thread synchronisation, and output comparison are described. Based on the redundant thread execution, a method to diagnose permanent and intermittent faults is developed, which enables the adaption of the system in case of these faults. To further increase the parallelism of double execution, optimistic double execution is proposed, which is able to speculatively start subsequent data-flow threads before result verification. In order to recover from errors detected with optimistic double execution, a data-flow based global checkpointing mechanism is presented.

The data-flow based fault-tolerance mechanisms were studied with the open-source multi-core simulator COTSon. The evaluation shows that double execution introduces small run time overhead compared to an ideal lockstep machine and can even achieve a speedup in certain cases. Furthermore, the execution speed of double execution can be gradually reduced, when permanent faults in the processing elements occur. In case of transient faults, the restart of data-flow threads has only a small impact on the execution speed, while optimistic double execution can exploit underutilised system resources to speed up the redundant execution of applications with low parallelism.

Danksagung

An dieser Stelle möchte ich mich bei den Personen bedanken, ohne deren Unterstützung diese Dissertation nicht hätte geschrieben werden können.

Zuallererst bedanke mich bei meinem Doktorvater, Herrn Prof. Dr. Theo Ungerer, dass er mir die Möglichkeit geben hat, diese Dissertation im Rahmen des TERAFLUX Projekts verfassen zu können. In jeder Phase meines Promotionsvorhabens stand er mir mit konstruktiver Kritik und zahlreichen Verbesserungsvorschlägen zur Seite. Bei Herrn Prof. Dr.-Ing. Rudi Knorr bedanke ich mich für die Zweitbegutachtung dieser Dissertation. Herrn Prof. Dr. Bernhard Bauer danke ich, dass er sich als Prüfer zur Verfügung gestellt hat.

Weiterhin danke ich allen Mitarbeitern des Lehrstuhls, mit denen ich in den letzten Jahren erfolgreich an den verschiedensten Projekten, Forschungsanträgen und Lehrveranstaltung zusammenarbeiten durfte. Mein besonderer Dank gilt den Kollegen Stefan Metzlaff, Florian Haas, Jörg Mische und Arne Garbade für die unzähligen wissenschaftlichen und nicht-wissenschaftlichen Diskussionen, die wesentlich zu dieser Arbeit beigetragen haben.

Bei meiner Familie, insbesondere meiner Frau Verena, bedanke ich mich für die große Geduld und die grenzenlose Unterstützung.

Finally, I'd also like to thank Prof. Dr. Roberto Giorgi from the University of Siena for his great hospitality during my stays in Siena.

Sebastian Weis

Table of Contents

List of Abbreviations

ALAB	Active Load Address Buffer
CARER	Cache-Aided Rollback Recovery
CB	Comparison Buffer
CLB	Checkpoint Log Buffer
CRTR	Chiplevel Redundantly Threaded Multiprocessor with Recovery
DCC	Dynamic Core Coupling
DE	Double Execution
DMR	Dual Modular Redundancy
DRMT	Data-flow Scheduled Redundant Multi-Threading
DVFS	Dynamic Voltage and Frequency Scaling
FDU	Fault Detection Unit
HTM	Hardware Transactional Memory
LBRA	Log-based Redundant Architecture
LVQ	Load Value Queue
ODE	Optimistic Double Execution
PU	Processor Unit
RQ	Ready Queue
SDF	Scheduled Data-Flow
SMT	Simultaneous Multithreading
SRT	Simultaneous Redundant Threading
SRTR	Simultaneous Redundant Threading with Recovery
TF	Thread Frame
TLP	Thread-Level Parallelism
TLS	Thread-Local Storage
TMR	Triple Modular Redundancy
TPL	Thread-To-PE-List

TQ	Thread Queue
TSU	Thread Scheduling Unit

1 Introduction

Current microprocessor devices, like Nvidia's Fermi architecture [Wittenbrink et al., 2011] or Intel's Haswell [Hammarlund et al., 2014] microarchitecture incorporate several Billion transistors[1] and it is expected that the ongoing improvements of the semiconductor fabrication technology will let the number of transistors per chip further increase [ITRS, 2013]. While the ongoing device scaling provides opportunities for computer architects to build even more parallel and powerful computing devices, the shrinking feature sizes of future chips also increases the probability of transient, intermittent, and permanent faults [Constantinescu, 2003; Srinivasan et al., 2004; Borkar, 2005]. This means that external and internal influences on the chip, like voltage fluctuation, cosmic radiation, thermal cycling, variability in the manufacturing process, or silicon wearout, will lead to increasing transient, permanent, and intermittent fault rates in future semiconductor devices [Mukherjee, 2008; Constantinescu, 2003; Borkar, 2005]. Additionally, the increasing gate leakage of future microprocessors raises the cost for burn-in testing of the devices [Borkar, 2005]. It can be prospected that hardware faults in present multi-core and future many-core systems may become unavoidable and fault-tolerance mechanisms must be also considered for general-purpose multi-core processors.

While fault-tolerance mechanisms have a long tradition in safety-critical [Yeh, 1996] and high-availability computer systems [Siewiorek et al., 1998], the architecture of general-purpose microprocessors is much stronger influenced by economical constraints. This means that future multi-core processors will require fault-tolerance techniques, which are capable to scale with the number of cores and increasing hardware fault rates at a reasonable architectural effort [Borkar, 2005]. Although current server processors [Iyer et al., 2005] implement error correcting and detecting codes to detect and correct errors in the main memory, the caches, or the cores' register files [AMD, 2013], the computational logic in the pipelines remains often unprotected. While the well-established lockstep implementations of safety-critical and high-availability systems provide high fault coverage and are also able to detect faults in the computational logic of a processor, lockstep execution imposes also significant challenges for parallel systems and may limit the performance and the scalability of parallel applications and architectures. In particular, lockstep techniques require high spatial redundancy and fully deterministic

[1]`https://en.wikipedia.org/wiki/Transistor_count`, Retrieved on August 15, 2015

execution [Mukherjee, 2008, pp. 213 f.], which complicates the use of complex out-of-order processors and modern power management techniques and may limit the implementation in commodity multi-core processors [Bernick et al., 2005]. Additionally, when faults become more frequent, computer systems must also provide scalable recovery and fault adaption mechanisms to prevent frequent catastrophic breakdowns [Sorin, 2009, p. 61].

Today, most microprocessors are based on the control flow driven von Neumann architecture, which uses a program counter for the sequential execution of instructions. However, the strict memory access ordering of the program counter driven execution and shared variables in parallel applications impose challenges for the scalable implementation of redundant execution and checkpointing mechanisms [LaFrieda et al., 2007; Rashid et al., 2008; Mukherjee, 2008; Yazdanpanah et al., 2013]. In this thesis, we propose redundant execution and checkpointing mechanisms for a data-flow based execution model to support scalable redundant execution and checkpointing for parallel applications.

The data-flow execution principle is a well-known approach to overcome restrictions introduced by the von Neumann architecture. However, fine-grained data-flow architectures [Dennis et al., 1975; Gurd et al., 1985; K. Arvind et al., 1990] suffer from poor sequential execution performance or high synchronisation overhead, because of the fine-grained parallelism [Robic et al., 2000]. Therefore, coarse-grained data-flow models were developed, which combine data-flow driven thread scheduling and synchronisation with the efficient sequential execution of current von Neumann processors, while they still provide enough parallelism of the application to fully utilise a multi-core processor [Yazdanpanah et al., 2013]. The coarse-grained thread execution allows that data-flow threads can be started only when all input data is available. The data-flow threads can then execute without waiting for external input. In particular, with the focus on future many-core processors and the need for efficient, parallel execution paradigms, coarse-grained data-flow architectures and compilers have gained new attention in academia [Giorgi et al., 2014b; Etsion et al., 2010; Giorgi et al., 2007; G. Gupta et al., 2011; Hum et al., 1995; Li et al., 2012; Stavrou et al., 2005; Zuckerman et al., 2011].

1.1 Main Contributions

This thesis integrates redundant execution and checkpointing mechanisms in a coarse-grained data-flow execution model to overcome restrictions, which may limit the implementation of scalable redundant execution and checkpointing in commodity multi-core processors.

In detail, this thesis makes the following contributions:

1. A coarse-grained data-flow execution model, originally developed in the Teraflux project [Giorgi et al., 2014b], is extended with support for data-flow based redundant execution, fault recovery, and fault diagnosis.
2. The proposed fault-tolerance mechanisms are integrated in a subset of the Teraflux data-flow architecture [Giorgi et al., 2014b] and necessary fault-tolerance enhancements to the architecture and the data-flow runtime system are discussed in detail.
3. An optimistic redundant execution scheme is developed, which is able to increase the parallelism of the redundant thread execution by a speculative start of subsequent

threads to exploit underutilised system resources and therefore speed up redundant execution.

4. A data-flow based global checkpoint mechanism is proposed, which is able to reduce the overhead of global checkpoint creation.
5. The proposed data-flow based fault-tolerance mechanisms are evaluated with the open source multi-core simulator COTSon [Argollo et al., 2009], an x86_64 functional-first full system simulator, which was extended with support for data-flow execution in the Teraflux project. The data-flow based fault-tolerance mechanism are compared with an ideal lockstep system and a conventional global checkpointing scheme. The evaluation results show that a data-flow execution model supports the construction of scalable redundant execution, checkpointing, and flexible adaption in the case of permanent and intermittent faults.

1.2 Structure

The rest of this thesis is structured as follows. In Chapter 2, background information on the terminology and the technical details of faults in microprocessor systems are given. Furthermore, the challenges of redundant execution and checkpointing in parallel multi-core architectures are discussed and the data-flow execution principle and its advantages for the implementation of redundant execution and recovery mechanisms are described. Chapter 3 gives an overview of prior work on redundant execution, backward error recovery in shared-memory multi-cores, fault-tolerant data-flow architectures, and fault diagnosis and adaption mechanisms in case of permanent faults. Chapter 4 provides an overview of the baseline data-flow execution model and the x86_64 multi-core architecture used in this thesis. Chapter 5 proposes *double execution*, a redundant execution mechanism for data-flow threads and discusses how double execution can be integrated in the baseline data-flow execution model and architecture. The chapter also proposes a thread restart mechanism for fault recovery and a fault localisation and adaption technique, when the system suffers from permanent or intermittent faults. Chapter 6 describes *optimistic double execution*, a speculative variant of double execution, which can increase the parallelism of redundant thread execution by committing data-flow threads before result comparison. Furthermore, a data-flow based global checkpointing mechanism is presented. Chapter 7 presents evaluation results on the fault-tolerance mechanisms, which are proposed in the Chapters 5 and 6. In Chapter 8, the results of this thesis are summarised and future research opportunities are proposed.

1.3 Publications

Partial results of this thesis have been published in the following papers:

S. Weis, A. Garbade, F. Bagci and T. Ungerer. ‘Fault detection and reliability techniques for future many-cores’. In: *HiPEAC ACACES Summer School (Poster Abstract)*. 2010. isbn: 978-90-382-1631-7.

S. Weis, A. Garbade, J. Wolf, B. Fechner, A. Mendelson, R. Giorgi and T. Ungerer. ‘A Fault Detection and Recovery Architecture for a Teradevice Dataflow System’. In: *First Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)*. 2011, pages 38–44. DOI: `10.1109/DFM.2011.9`.

S. Weis, A. Garbade, S. Schlingmann and T. Ungerer. ‘Towards Fault Detection Units as an Autonomous Fault Detection Approach for Future Many-Cores’. In: *ARCS 2011 Workshop Proceedings*. 2011, pages 20–23. ISBN: 978-3-8007-3333-0.

S. Weis, A. Garbade, B. Fechner, A. Mendelson, R. Giorgi and T. Ungerer. ‘Architectural Support for Fault Tolerance in a Teradevice Dataflow System’. In: *International Journal of Parallel Programming* 44.2 (2016), pages 208–232. DOI: `10.1007/s10766-014-0312-y`.

2 Background

This chapter presents the background on concepts and techniques used within this thesis. Section 2.1 describes the background on hardware faults, including their classification and their sources in modern microprocessors. Section 2.2 introduces different types of redundancy in fault-tolerant computer systems. Section 2.3 gives an overview of the general concepts on redundant execution, while Section 2.4 describes the background on recovery mechanisms for shared-memory multi-core processors. Section 2.5 gives an introduction to the data-flow execution principle and its advantages for fault-tolerant execution.

2.1 Faults in Computer Systems

Avizienis et al. [1986] classify faults in computer systems, based on their origin, into *physical faults* and *human-made faults* [Siewiorek et al., 1998, p. 23]. Physical faults are caused by chip-external or chip-internal “physical phenomena”. These “physical phenomena” can be induced by chip-internal flaws like “threshold changes”, “open circuits”, “short circuits”, or by chip-external influences like “environmental conditions”, “electromagnetic interference”, or “vibration”. In contrast to physical faults, which are caused by a malfunction of the hardware, human-made faults are induced by humans, e.g. faults in the hardware or the software design, or by erroneous interaction of the user with the system. [Siewiorek et al., 1998, p. 23]

The fault-tolerance mechanisms presented in this thesis are intended to cope with physical faults. In particular, we assume that human-made faults are prevented by other mechanisms, e.g. testing or formal verification of the hard- and software. Therefore, the rest of this section focuses on physical faults induced by physical phenomena at run time or manufacturing time.

2.1.1 Manifestation of Faults

The effects of physical faults can be further distinguished according to their manifestation on the different levels of a computer system. The following definition are based on Avizienis et al. [2004].

A *fault* on the lowest hardware level describes an incorrect hardware state induced by a chip-internal or chip-external physical phenomena [Siewiorek et al., 1998, p. 23]. When a fault becomes visible and affects a hardware state, this manifestation is called an *error*. For instance, the system suffers from an error when a fault in a DRAM-cell induces a bit flip in the main memory [Sorin, 2009, pp. 3 f.]. An error can later cause a *failure*, which denotes a "deviation of the computing system from its expected or originally specified behaviour" [Avizienis et al., 1986]. For instance, when a bit flip in the main memory is later used by software for further calculations, leading to a memory access violation, this deviation is called a failure.

Faults, errors, and failures can be *masked* on each level of this hierarchy, which means that they will not propagate to the next level [Sorin, 2009, p. 2]. Several microarchitectural and architectural masking effects are discussed in [Mukherjee et al., 2003]. For instance, an error is microarchitectural masked, when it affects, e.g. the branch predictor state of a processor, which does not influence the functional correctness of the execution [Mukherjee et al., 2003]. A fault or error that has been masked does not need to be detected nor corrected, since it will never cause a failure in the system.

2.1.2 Duration and Sources of Faults

Depending on the underlying physical phenomena, physical faults are also classified according to the duration of their appearance. The literature on fault-tolerant computer systems usually distinguishes between *transient*, *intermittent*, and *permanent* faults [Sorin, 2009; Mukherjee, 2008; Siewiorek et al., 1998]:

Transient Faults Transient faults are caused by sporadic physical phenomena, which immediately disappear after their occurrence. The manifestation of a transient fault as an error is often called a *soft error*. [Sorin, 2009, p. 3]

Transient faults can be caused by environmental radiation, like cosmic rays or alpha particles and also by fabrication and architecture induced reasons, like "transistor variability", "thermal cycling", or "erratic fluctuations of minimum voltage at which a circuit is functional" [Mukherjee, 2008, p. 20]. While the radiation induced transient fault rate is usually randomly distributed and depends on the operating environment of the chip [Lehtonen et al., 2005], the architecture and fabrication induced transient fault rate may be influenced by power management and scheduling decisions at run time [Mukherjee, 2008, p. 20].

Permanent Faults A fault is called permanent, when its manifestation is continuous. In this case, the affected hardware component is permanently broken and the fault will be present over the lifetime of the device. This means, that a component suffering from a permanent fault will continuously produce wrong results. [Sorin, 2009, p. 3]

Mukherjee [2008, p. 14] further distinguishes between extrinsic and intrinsic permanent faults. Extrinsic permanent faults are induced by chip-external influences, e.g. defects in the manufacturing process, like contaminants [Mukherjee, 2008, p. 14] or residuals [Srinivasan et al., 2004]. Extrinsic faults usually influence the permanent fault rate at the beginning of the lifetime of a chip, called *infant mortality* phase. After this phase of infant mortality, the extrinsic permanent fault rate decreases. By contrast, intrinsic permanent faults are induced by wearout of chip material over the lifetime of the chip, induced by e.g. “electromigration”, “metal stress voiding”, “gate oxide wear out”, or “hot carrier injection” [Mukherjee, 2008, pp. 14 ff.]. Since possible wear out effects are increasing with the lifetime of the chip, the fault rate of intrinsic permanent faults is increasing with the lifetime of the chip, too [Mukherjee, 2008, p. 14].

Intermittent Faults Intermittent faults are only occasional present [Siewiorek et al., 1998, p. 22]. During their occurrence they may behave like permanent faults, however, unlike a permanent fault, intermittent faults may disappear after a certain time [Constantinescu, 2003]. Constantinescu [2003] uses three criteria to determine an intermittent fault:

1. It occurs repeatedly at the same place.
2. Errors caused by the intermittent faults occur in bursts.
3. Replacement of the affected transistor or hardware structure repairs the fault.

Intermittent faults often precede permanent faults and are also caused by variability effects of the chip’s manufacturing process. For instance, a device may suffer from intermittent delay faults due to increasing resistance, before an open fault occurs, induced by wear out of chip material. [Constantinescu, 2003]

2.1.3 Increasing Fault Rates in Future Systems

Based on the assumption that the progress of the semiconductor fabrication technology will continue, Sorin [2009, pp. 5–7] describes three main causes for increasing fault rates in future processors:

Smaller Transistors The ongoing downscaling of semiconductor devices may lead to increasing transient faults caused by a reduction of the critical charge of the transistors [Shivakumar et al., 2002]. This means that environmental influences, like cosmic rays or alpha particles in the chip packaging, will more likely influence the critical charge of the transistors [Mukherjee, 2008, pp. 29 f.].

Furthermore, the downscaling complicates the manufacturing process, leading to a higher probability of manufacturing defects or variability between different chips [Borkar, 2005]. Borkar [2005] mentions two main run time independent sources of variability induced by the downscaling of the transistor sizes, i.e. random dopant fluctuation in the channels of a transistor, leading to variable threshold voltages in the transistors and the sub-wavelength lithography process, leading to “line edge roughness”.

More Transistors per Processor The overall fault rate of a chip is proportional to the number of the transistors per chip [Koren et al., 2007, p. 16], which means that the fault rate increases with number of the transistors per chip, particularly when single transistor fault rates are increasing with the ongoing downscaling [Sorin, 2009, p. 6].

Complexer Chips The quest for higher performance has led to complex superscalar out-of-order pipelines and more recently also general-purpose multi-core processors. On the other side, the increasing complexity introduced by multi-core processors, coherence protocols, on-chip memory controllers or even on-chip GPUs will make future processors harder to verify and to test [Sorin, 2009, pp. 6 f.]. In particular, burn-in tests, the conventional method to test for manufacturing defects before shipping of the device, gets more complicated with a reduction of the threshold voltage [Borkar, 2005].

2.2 Fault-Tolerant Computer Systems

With the reduced reliability of future silicon devices, faults may become unavoidable and require efficient architectural fault-tolerance mechanisms even for general-purpose computer systems.

"*Fault tolerance* is the ability of a system to continue to perform its tasks after the occurrence of faults. The ultimate goal of fault tolerance is to prevent system failures from occurring". [Pradhan, 1996, pp. 4–5]

A fault-tolerant computer system must be able to *detect* and *recover* from faults. In case of a transient fault, the system may continue its normal operation after fault detection and recovery, since the source of the fault usually has disappeared after recovery. However, in case of intermittent and permanent faults, a fault-tolerant system must also support *fault diagnosis*, in order to locate the faulty component and distinguish between transient or permanent/intermittent duration and *fault adaption*, in order to adapt the system in case of an intermittent/permanent faulty component. Fault adaption is important for permanent faults to guarantee forward progress of the system. [Sorin, 2009, p. 81]

Fault-tolerant systems use redundancy in different forms to detect, recover, and adapt the system in case of faults [Sorin, 2009, p. 19]. In the next subsection, we describe the main forms of redundancy, as they are used in fault-tolerant computer systems.

Redundancy in computer systems can be classified into *information redundancy*, *spatial redundancy*, and *temporal redundancy* [Sorin, 2009, pp. 19–25].

2.2.1 Information Redundancy

Information redundancy adds redundant information to the data to detect and correct errors by redundant coding [Koren et al., 2007, pp. 55 ff.]. Redundant coding techniques can be distinguished in error detecting (EDC), e.g. parity codes, and error correcting codes (ECC), e.g. Hamming codes, [Lehtonen et al., 2005; Koren et al., 2007]. While EDC is only able to detect errors, ECC is also able to correct errors. Redundant coding techniques are a well-established technique to protect the memory subsystem of a computer system against faults. For instance, off-chip DRAM or on-chip caches and the register file of modern server systems are already today protected by EDC/ECC [AMD, 2013].

However, information redundancy is usually not used to protect the arithmetic logic units of a processor pipeline. In order to protect the arithmetic logic and control units, usually spatial or temporal redundancy mechanisms are used.

2.2.2 Spatial Redundancy

Spatial redundancy physically replicates hardware modules and performs the same operation in parallel on the replicated modules. Afterwards, the results of the redundant modules are compared by a comparator. [Sorin, 2009, p. 19]

Figure 2.1 shows the general concept of a Dual Modular Redundant (DMR) system, a form of spatial redundancy, where two modules are physically replicated. During operation, all inputs are replicated for both modules. After both modules have processed their input values, the output values are compared, and the correct result can be forwarded to the next module.

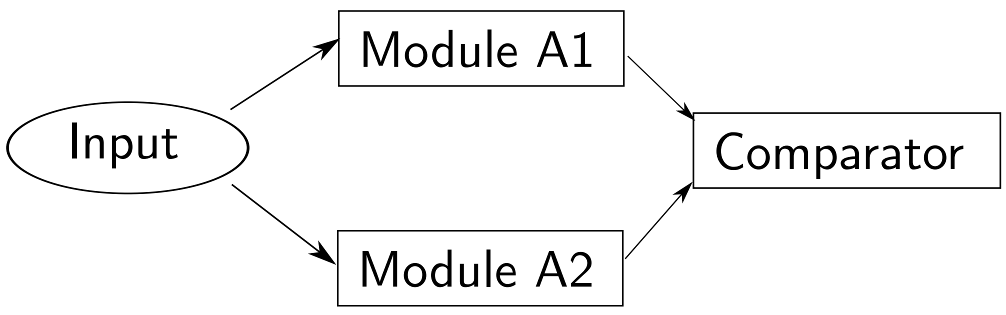


Figure 2.1: An abstract DMR structure.

Triple modular redundancy (TMR) uses a third module, which enables the system to vote between three results and to identify the component, which has produced the wrong result. As a consequence, the system can mask the error by directly forwarding the correct result. [Sorin, 2009, pp. 19 f.]

Spatial redundancy can be used at different levels of a computer system, e.g. at gate-level, where single flip-flops are replicated, or at core-level where complete pipelines are replicated for redundant program execution [Sorin, 2009, p. 20].

2.2.3 Temporal Redundancy

Unlike spatial redundancy, temporal redundancy does not require physically duplicated hardware components, instead an operation is executed n-times on the same hardware module. However, the performance of the component is reduced by the number of the re-executions. Beside the cost for re-execution, the component must also be able to safe the temporal results for comparison. Furthermore, temporal redundancy may not detect all possible permanent or intermittent faults, since permanent or intermittent faults may affect the re-executions in the same way. [Sorin, 2009, p. 22]

2.3 Redundant Execution

A well-established approach to implement fault detection in computer systems is redundant execution. Redundant execution systems execute a duplicated instruction stream in a spatial or temporal redundant manner and compares periodically the outputs of the instruction streams. By contrast to information redundancy techniques, redundant execution mechanisms cover multiple hardware structures and are able to detect faults also in the control and the arithmetic units of a processor pipeline. [Mukherjee, 2008, p. 207]

Although redundant execution mechanisms are a well-known and established fault detection approach with high fault coverage, prior redundant execution mechanisms have drawbacks in terms of hardware overhead, activation flexibility, implementation complexity, and their usage for parallel applications.

In the following, the general redundant execution mechanisms are described. Mukherjee [2008, pp. 207 ff.] classifies redundant execution mechanisms into tightly-coupled and loosely-coupled redundant execution. In the rest of this section, we will describe the principles of tightly-coupled and loosely-coupled redundant execution and discuss their advantages and drawbacks.

2.3.1 Tightly-Coupled Redundant Execution

Tightly-coupled redundant execution, also called *lockstep execution*, implements spatial redundancy on processor or system-level by coupling spatial redundant execution units in a cycle-by-cycle manner. Tightly-coupled lockstep systems usually implement result comparison after each cycle [Mukherjee, 2008, p. 212]. Usually lockstep execution requires a completely duplicated state in all redundant components. Lockstep execution is widely used in safety-critical embedded systems [Infineon Technologies AG, 2015; STMicroelectronics, 2014] and also high-availability systems [Siewiorek et al., 1998; Mukherjee, 2008].

However, the implementation of tightly-coupled lockstep systems is costly in terms of hardware overhead, since all redundant resources must be fully replicated. Furthermore, the execution of redundant components must be absolutely deterministic. This determinism includes the whole microarchitectural state at each cycle, incorporating branch prediction decisions, cache miss behaviour, or interrupt delivery [Mukherjee, 2008; Bernick et al., 2005]. The increasing complexity of future multi-core systems with support for fine-grained power-management, multi-cores, out-of-order-pipelines, multi-threaded workloads or performance heterogeneity of the cores due to process variability, makes the construction of tightly-coupled lockstep systems more complex [Bernick et al., 2005]. Additionally, tightly-coupled lockstepping demands that all hardware components are spatial redundant, which means that a permanent fault automatically renders one additional core or processor useless, since spatial redundancy is impossible with an uneven number of execution units [LaFrieda et al., 2007].

2.3.2 Loosely-Coupled Redundant Execution

By contrast to tightly-coupled redundant execution, loosely-coupled redundant execution mechanisms do not rely on strict cycle-by-cycle synchronisation and remove the strict timing constraints for the output comparison [Mukherjee, 2008, p. 222]. This simplifies the construction of loosely-coupled redundant execution mechanisms, since the execution units must not be synchronised at every cycle and further allows efficient temporal resource sharing, since the redundant threads can be executed asynchronously [Mukherjee, 2008, pp. 222–223]. Resource sharing between redundant execution streams is usually impossible in a tightly-coupled lockstep implementation without sacrificing the execution speed.

Since loosely-coupled redundant execution does not require a complete duplication of the hardware, Reinhardt et al. [2000] developed the notion of the *Sphere of Replication*, which describes the logical area of a chip, which is covered by temporal or structural redundancy. Due to the redundancy, errors within the sphere of replication can be detected by the redundant execution mechanism. Figure 2.2 depicts the sphere of replication for a DMR structure.

As a consequence, the redundant execution scheme must provide consistent *input replication* for input data, which enters the sphere of replication. Furthermore, all computational results, which leave the sphere of replication must be compared in order to detect errors, which may have occurred during execution. Any error which leaves the sphere of replication, cannot be detected at a later stage without additional mechanisms. [Reinhardt et al., 2000]

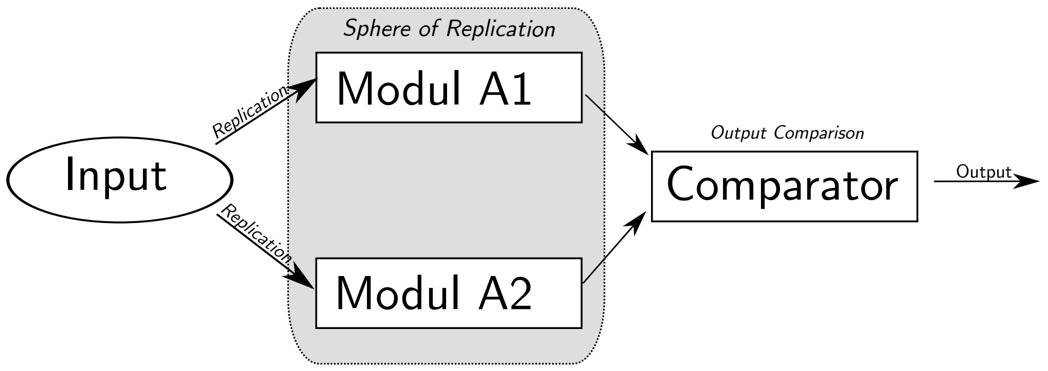


Figure 2.2: Sphere of Replication for the abstract DMR system.

Although the loosely-coupled lockstep mechanisms provide advantages in terms of hardware- and synchronisation overhead, the asynchronous execution in loosely-coupled systems also introduces new problems for input replication and output comparison and complicates the use of loosely-coupled redundant execution for parallel applications. In the following, we will describe the main challenges for input replication and output comparison in more detail.

Input Replication By contrast to tightly-coupled lockstep execution, loosely-coupled redundant execution may execute redundant load and store operations asynchronously at different times. This asynchronous execution complicates the input replication, since redundant streams may read input data that has been changed in the meantime [Reinhardt et al., 2000]. Smolens et al. [2006] call these race conditions between redundant execution streams "*input incoherence*". Input incoherence between redundant execution streams can be caused, when the input data of redundant instruction streams is changed, while the streams are executed asynchronously. Reasons for this behaviour can be e.g. memory mapped I/O, interrupts, or shared variables in parallel applications [Smolens et al., 2006; Reinhardt et al., 2000].

The input incoherence problem in loosely-coupled lockstep implementations is addressed by different publications.

- Reinhardt et al. [2000] propose a dedicated hardware structure, the *Load Value Queue* (LVQ) to pass already fetched values of the leading stream to the trailing stream. The LVQ leads to a tight coupling of the redundant execution streams, since the trailing execution stream must wait for input values, which are produced by the leading stream and requires a hardware coupling of the redundant execution units. LaFrieda et al. [2007] identify that this static coupling limits the dynamic adaption of the system in case of permanent faults.
- Smolens et al. [2006] solve the input replication problem by detecting inconsistent input with the implemented fault detection and recovery mechanisms. However, LaFrieda et al. [2007] show that *relaxed input replication* can lead to severe performance penalty for parallel applications with large comparison intervals. Furthermore, *relaxed input replication* may lead to livelock situations, which can only be solved by tightly-coupled lockstep execution of the redundant streams [Smolens et al., 2006].
- Reinhardt et al. [2000], LaFrieda et al. [2007] and Rashid et al. [2008] use the cache and the coherence protocol to guarantee consistent input values for the redundant execution streams. Reinhardt et al. [2000] propose the *Active Load Address Buffer* (ALAB), which tracks unfinished loads and prevents the replacement of these values in the cache. LaFrieda et al. [2007] propose a sliding window, which enables the consistent input replication for redundant streams of parallel applications. While they solve the input replication problem, their solution requires bus snooping, changes to the private caches and the coherence protocol. Rashid et al. [2008] propose another input replication scheme, which proposes hardware support for input replication by using special cache structures and adapting the cache coherence protocol. This scheme requires complex changes to the coherence protocol and the private caches, too.

Output Comparison All state which leaves the sphere of replication must be compared for errors. When data leaves the sphere of replication, the redundant processing units must synchronise and compare their state. As a consequence, a larger sphere of replication may reduce the synchronisation frequency, but also increase the error detection latency. Reinhardt et al. [2000] distinguish between two spheres of replication, one which compares results on register write back and store commit, and another one, which only compares retired store instructions. They concluded that output comparison of stores may increase the performance of the redundant execution, since the synchronisation frequency is reduced, leading to more flexible utilisation of the multi-thread processor resources.

Beside the synchronisation overhead, the output must be also transferred to a comparison unit. For a sphere of replication, which excludes the register file, all register write backs and all store instructions must be compared [Reinhardt et al., 2000]. To increase the synchronisation interval and reduce the exchanged data, Smolens et al. [2006] propose to generate a CRC-16 signature, called *fingerprint*, of the output. This reduces the cost for transferring the computational state to the comparison unit. However, some errors may be masked by collisions of the CRC-16 signatures.

2.4 Recovery

When a redundant execution mechanism has detected an error, the system must be able to recover or mask the erroneous execution state.

2.4.1 Terminology

Recovery mechanisms can be distinguished in backward error correction (BER) and forward error correction (FER) [Sorin, 2009, p. 61]. FER allows masking the fault without the need to recover to a prior execution state. To be able to forward the correct result, FER must be able to identify the erroneous execution. In a DMR system, the identification of the faulty state requires additional knowledge, e.g. by information redundancy. However, FER mechanisms often use TMR to identify the erroneous execution state by a majority voting mechanism. [Mukherjee, 2008, p. 255]

Since TMR requires significant hardware overhead, it is usually restricted to time critical environments where not only the functional correct result is required, but also the time when the correct result is available, e.g. safety-critical embedded systems [Yeh, 1996].

In contrast to FER, BER recovers the execution to a previous state. However, this requires the creation of a checkpoint of an intermediate state of the execution (called *recovery point*) [Sorin, 2009, p. 62].

In BER systems, the part of the architecture, which can be recovered by the recovery mechanism is called the *sphere of recoverability* [Sorin, 2009, p. 63]. The *output commit problem* describes the problem that execution results, which have left the sphere of recoverability can no longer be recovered. The output commit problem directly constrains the fault detection mechanism, since data that leaves the sphere of recoverability must always be checked for faults before leaving the sphere of recoverability. [Sorin, 2009, p. 63]

The *input commit problem* describes the problem that state, which is read from the outside world, e.g. by an I/O operation, can not be replayed, when the execution is recovered. The input commit problem can be solved by logging and replaying I/O in case of recovery. [Sorin et al., 2002]

2.4.2 BER in Shared-Memory Processors

In order to implement BER in a shared-memory multiprocessor, the system must be able to establish periodic checkpoints of the execution state. When a fault is detected by the fault detection mechanism, the system is recovered to the state of the last checkpoint.

Depending on the sphere of recoverability, backward error recovery can be implemented in shared-memory systems at different granularity, i.e. the cores, the caches, and the main memory [Gold et al., 2006].

Some proposals [Smolens et al., 2006; Ray et al., 2001; Gomaa et al., 2003; Vijaykumar et al., 2002] based on speculative out-of-order processors use the architectural state of the cores for recovery. In these approaches, the complete speculative state of the cores, including the speculative results in the reorder buffers and the functional units, can be recovered. Here, the state of a checkpoint consists of the architectural registers and the memory state. A new checkpoint is inherently created when an instruction commits. When a fault is detected by a fault detection mechanism, the speculative state in the pipeline is squashed, similar to a branch mispredict, and the execution is restarted at the last committed instruction. Consequently, the sphere of recoverability does not include the architectural registers, which must be protected by an ECC mechanism.

Some other proposals [Wu et al., 1990; LaFrieda et al., 2007; Rashid et al., 2008] extend the sphere of recoverability to the cache hierarchy. In these cases, the sphere of recoverability consists of the cores, including all microarchitectural structures and the data caches. A new checkpoint in such systems is established by creating a copy of the architectural state of the cores and preventing the write back of modified cache lines [Hunt et al., 1987]. Whenever a modified cache line is evicted and written back to the main memory, a new checkpoint must be established, since the main memory is no longer recoverable from errors, once the evicted cache line has overwritten the value in the main memory [Ahmed et al., 1990].

When the sphere of recoverability comprises the whole system, including the caches and main memory, a copy of the state of the architectural registers of the cores and the memory must be established [Prvulovic et al., 2002; Sorin et al., 2002].

While copying the architectural registers to a shadow register set or the main memory is simple, copying the whole memory would impose huge copy and storage overhead [Sorin, 2009, p. 72]. Therefore, the main memory state is usually covered by a logging mechanism. The logging mechanisms records modifications to the main memory with respect to the last checkpoint. When a fault is detected, all changes are reverted by the log. [Sorin, 2009, p. 72]

Establishing a new checkpoint in a multi-core processor can be done in two ways, i.e. global or local [Prvulovic et al., 2002]. Global means that all cores in the system agree to take a checkpoint at a specific point in time.

In shared-memory systems a global checkpoint can be created by stopping the execution on all cores and flushing the caches. Afterwards, all cores have the same view on the main memory. Finally, a global checkpoint can be established. However, in case of a fault, the global system, i.e. all cores, not only the erroneous execution, must be recovered, leading to high recovery overhead. [Pradhan, 1996, pp. 160 ff.]

Local means that each core can establish a checkpoint independently from the other cores. While local checkpoint creation may reduce the communication overhead between the cores when a new checkpoint is established, it can also lead to the so-called *domino effect*, which is induced by data dependencies between the threads and can cause cascading rollbacks in order to reach a consistent system state [Sorin, 2009, p. 66].

2.4.3 Relation between Fault Detection and Checkpointing

There are several relations between checkpointing and the fault detection mechanism [Mukherjee, 2008, pp. 256 f.]. First, when a new checkpoint is established, the system must ensure that the new checkpoint is fault free. If the checkpoint would contain an error, a rollback would always recover to the erroneous state of the checkpoint, and the system would never be able to recover to an error free state [Sorin, 2009, p. 67]. Second, all data that leaves the sphere of recoverability cannot be recovered and must be checked for errors [Gold et al., 2006].

In a conventional shared-memory multi-core the first point always means that the architectural state of each core in the system must be checked for faults, when a new checkpoint is established. Since in a von Neumann architecture, the architectural registers must always be included in a checkpoint, every instruction write back to the architectural registers and store instruction must be checked for errors in order to guarantee a fault-free checkpoint. Vijaykumar et al. [2002] identify that this can result in significant read pressure to the register file for result comparison and also result transfer to the comparison unit. Therefore, Vijaykumar et al. [2002] propose a *Register Value Queue*, an additional hardware structure, to prevent frequent register accesses. Smolens et al. [2004] propose fingerprinting to reduce the communication bandwidth to the comparison unit. However, also with fingerprinting every register write back must be included in the CRC-16 signature.

Since all data that leaves the sphere of recoverability must be checked for errors, the output of the redundant execution streams must always be compared and synchronised [Gold et al., 2006]. Consequently, a smaller sphere of recoverability, e.g. including only the cores' pipelines, usually requires frequent output comparison. When the sphere of recoverability is larger, i.e. including the main memory, the redundant streams must compare results only when I/O takes place [Mukherjee, 2008, p. 256]. However, in this case the complete main memory must be included in the checkpoint, increasing the overhead for checkpoint creation and logging [Prvulovic et al., 2002; Sorin et al., 2002].

2.4.4 Fault Diagnosis and Adaption for Permanent/Intermittent Faults

While transient faults usually disappear after error recovery, permanent and intermittent faults require additional mechanisms to diagnose the duration of the fault and adapt the system to prevent livelocks of the system [Sorin, 2009, p. 81]. Fault diagnosis means that the system can localise the faulty component, identify whether a permanent/intermittent fault occurred and trigger an adaption mechanism to circumvent the faulty component [Sorin, 2009, p. 81]. Sorin [2009, pp. 83–85] distinguishes between built-in-self-tests (BIST) and diagnosis during normal execution. BIST require that a component is stopped and a BIST is executed. A BIST can be implemented in hardware [Agrawal et al., 1993] or software [Kranitis et al., 2005] and executes a specific test pattern on the component. When the output of the component differs from the expected output of the BIST, the component suffered from a fault. Fault diagnosis during normal execution uses available redundancy for fault detection and recovery, e.g. ECC or redundant execution, to localise faults at run time. For instance, a TMR system can use majority voting to identify the faulty execution unit.

In the case of a permanent or intermittent fault, the system must be adapted to prevent recurring errors caused by the permanent fault. The functionality of the faulty component must be replaced. The component can be replaced by a "cold" or "hot" spare component. Cold spare means that the replacement is deactivated during non-faulty operation of the system and only activated when a permanent fault occurs. A hot spare component is already activated and can take over the execution instantaneous. [Sorin, 2009, p. 89]

With the rising probability of manufacturing and wearout defects, the ability of a fault-tolerant system to compensate permanent and intermittent faults is important. In this context *Graceful Degradation* of the systems become important. Graceful Degradation describes the ability of a system to repair or deactivate faulty components, while remaining operational with reduced performance. [Pradhan, 1996, p. 6]

2.5 Combining Fault Tolerance and Data-flow Execution

This section describes the general data-flow execution principle and how von Neumann and data-flow execution models can be combined to a hybrid data-flow/von Neumann execution model. Finally, we discuss how the data-flow execution paradigm can support fault-tolerance mechanisms.

2.5.1 The Data-flow Execution Principle

Compared to the control flow driven von Neuman architectures, data-flow architectures implement a data-flow driven execution model. In a data-flow execution model, a program is organised in a data-flow graph. The nodes in the graph represent instructions, while the arcs between the nodes describe data dependencies between instructions. During execution, the data flows along the arcs from node to node, while each node's instruction can only be executed, when the data has arrived at its input arcs. When all inputs have arrived at the input arcs, the node is allowed to *fire*, i.e. the instruction can be executed. [Yazdanpanah et al., 2013; Robic et al., 2000]

Since nodes can only be executed, when their data has arrived at the input arcs, the data-flow execution principle solves some problems of the von Neumann architecture. In particular, a data-flow program does not require a program pointer, since instructions are issued only on data availability. This solves all control flow hazards between instructions and enables data-flow architectures to exploit the available instruction level parallelism of the applications. [Yazdanpanah et al., 2013; Robic et al., 2000]

However, first fine-grained static data-flow architectures, like [Dennis et al., 1975], suffered from poor sequential execution performance [Robic et al., 2000] and could not exploit enough parallelism for subroutines and loop executions [Yazdanpanah et al., 2013]. While later fine-grained dynamic data-flow architectures, like [Gurd et al., 1985; K. Arvind et al., 1990], improved the parallelism of the data-flow execution, they require performance critical token matching hardware [Robic et al., 2000]. The token matching in dynamic data-flow architectures requires associative memory implementations to handle the massive parallelism and increases the hardware costs and the power consumption [Yazdanpanah et al., 2013].

2.5.2 Hybrid Data-flow/Von Neumann Execution

Today, most commodity computer systems are derivatives of the control flow driven von Neumann architecture, i.e. they make use of a program counter to control the sequential execution of a program [Yazdanpanah et al., 2013]. While modern von Neumann processors can efficiently execute sequential applications, parallel execution requires shared-memory synchronisation between threads to exchange data, which can lead to high synchronisation overhead and memory latencies [Arvind et al., 1988].

The problems of fine-grained data-flow architectures and the efficient sequential execution of von Neumann processors led to the idea of hybrid data-flow/von Neumann execution models [Yazdanpanah et al., 2013; Robic et al., 2000]. These architectures try to combine the advantages of a data-flow execution model, namely the efficient synchronisation and latency hiding of memory accesses, with the efficient sequential execution of von Neumann architectures.

Hybrid data-flow/von Neumann execution models do not execute single instructions in a data-flow manner, but whole sequences of instructions. This enables to combine the efficient sequential execution of modern processors with efficient data-flow driven synchronisation mechanisms. While the data-flow based synchronisation can reduce the synchronisation overhead and support data-driven scheduling, the sequential execution can exploit the cache hierarchy and out-of-order pipelines of current microprocessors to speed up the sequential execution [Yazdanpanah et al., 2013].

Although combining data-flow and von Neumann architectures promises advantages for efficient resource utilisation in multi-core architectures, it also introduces new complexity on the software side. In particular, hybrid data-flow/von Neumann architectures require that the program is structured in a data-flow graph, which requires the programmer or the compiler to model data dependencies and may restrict the use of certain data structures. [Yazdanpanah et al., 2013]

2.5.3 Combining Fault Tolerance and Data-flow Execution

The data-flow execution principle provides several advantages for the implementation of fault-tolerance mechanisms (based on Gaudiot et al. [1985]): (1) The decoupled and functional execution of the nodes in a data-flow graph provides inherent support for redundant execution, since the inputs and outputs of the nodes are known by the data-flow runtime system and can be used for input replication. Furthermore, the explicit data dependencies between nodes provide inherent comparison points for a redundant execution mechanism, because only the result of a data-flow node is consumed by subsequent nodes in the data-flow graph. (2) Data-flow architectures are parallel by design. The parallelism can be used for flexible and redundant execution of data-flow nodes. (3) The side-effect free semantic of a data-flow execution model also provides support for recovery mechanisms, since the nodes of a data-flow graph only depend on their input arcs. Therefore, a node can always be re-executed as long as the input arcs of a node are available [Najjar et al., 1990].

Using a hybrid data-flow/von Neumann execution model, the data-flow execution paradigm can be combined with the sequential von Neumann execution, while providing a redundant execution scheme, which simplifies input replication, synchronisation, output comparison, checkpointing and adaption of the system to permanent faults. An overview of prior hardware and software based fault-tolerance mechanisms, which use data-flow principles for fault detection and recovery are presented in Chapter 3.

2.6 Summary

This chapter presented background information on topics discussed in this thesis. In the first section, the terminology on hardware faults, different fault types, their sources in semiconductor devices, and the trend of fault rates in future systems were discussed.

Afterwards, the challenges for redundant execution in conventional shared-memory multi-cores were discussed. On the one hand, the required determinism of cycle-by-cycle lockstepping is hard to implement in parallel architectures and introduces significant hardware overhead, since time redundancy and resource sharing between redundant execution units is complicated. On the other hand, loosely-coupled redundant execution mechanisms, which allow flexible time sharing of components and a decoupled execution of redundant instruction streams, pose significant challenges for input replication, redundant thread management, synchronisation, and output comparison. In particular, loosely-coupled execution of parallel shared-memory applications introduces additional complexity, since accesses to shared variables from different redundant threads may lead to *input incoherence*.

Based on the fact that in future systems fault rates will increase, shared-memory checkpointing mechanisms were described. The checkpointing of shared-memory multi-core systems can have an impact on the scalability of a system, since global checkpoint creation must incorporate the architectural registers of all cores and the memory state, including the caches and the coherence protocol.

Finally, the advantages and drawbacks of the data-flow execution principle and the benefits of the hybrid data-flow/von Neumann architectures were described. The side-effect free execution, the inherent parallelism, and the implicit synchronisation of a data-flow execution model were discussed in the context of redundant execution and checkpointing, where the combination of von Neumann and data-flow execution models promise advantages for the scalable implementation of fault-tolerance mechanisms in multi-core architectures.

3 Related Work

This chapter presents prior work on topics related to this thesis. The chapter is structured as follows: Section 3.1 presents related work on redundant execution techniques, Section 3.2 gives an overview of backward error recovery mechanisms for shared-memory multi-cores, while Section 3.3 discusses prior research on fault-tolerant data-flow systems. Section 3.4 presents related work on fault-tolerant architectures, which can cope with permanent faults. The chapter is summarised in Section 3.5.

3.1 Redundant Execution Mechanisms

This section presents a broad overview of different redundant execution mechanisms; from tightly-coupled lockstep execution and microarchitectural redundant execution in superscalar to loosely-coupled, thread-level redundant execution in SMT and multi-core systems.

3.1.1 Lockstep Redundancy

Tightly-coupled lockstep systems have been used in commercial safety-critical and high-availability systems for several years [Siewiorek et al., 1998]. Due to the broad range of different lockstep architectures, we present here an excerpt of prior developments, which implement lockstepping on different levels.

The Stratus ftServer [Mukherjee, 2008, pp. 216 ff.] is build upon standard Intel x86 processors and able to execute off-the-shelf operating systems like Linux or Windows. The sphere of replication comprises the main memory, the chipset, and the processors in DMR or TMR configurations. Output comparison and input replication are implemented by an additional "fault detection and isolation" component, located before I/O is released to the PCI bus.

The NonStop Himalaya [Mukherjee, 2008, pp. 218 f.] architecture from Hewlett Packard uses off-the-shelf lockstepped MIPS processors, which are coupled by a custom "ASIC" component, taking care of input replication and output comparison. The sphere of replication of the Himalaya architecture includes the MIPS processors and the private

L2 caches. Output comparison and input replication is done before data is written back to the main memory.

Another example of a commercial lockstep system is the IBM S/390 microarchitecture [Slegel et al., 1999]. The S/390 microarchitecture supports lockstep execution of the instruction fetch (I-Unit) and the execute stage (E-Unit) of the pipeline. The results of the I-Unit and the E-Unit are compared by a "recovery and fault detection unit" (R-Unit), which holds a valid checkpoint of the processor's microarchitectural state. In case that the R-Unit detects a mismatch of the lockstepped I- or E-Units, the R-Unit can recover the microarchitectural state of the processor and restart the execution from a prior fault-free state.

Lockstep redundancy is also implemented in current embedded processors for safety-critical automotive systems, like the Infineon Aurix [Infineon Technologies AG, 2015] or the STMicroelectronic's SPC56 processor family [STMicroelectronics, 2014], however these processors usually do not support error recovery.

Although tightly-coupled lockstepping has been implemented in several commercial high-availability and mission-critical systems, it still requires tight synchronisation intervals between redundant execution units. This fine-grained synchronisation requires highly deterministic execution and complicates the use of tightly-coupled lockstepping for parallel applications and further impedes the use of complex out-of-order processors, as well as power saving mechanisms, like dynamic voltage and frequency scaling (DVFS) [Bernick et al., 2005].

3.1.2 Microarchitectural Redundancy

On microarchitectural level several mechanisms are proposed to exploit the inherent parallelism and rollback capabilities of complex out-of-order processors for redundant execution.

Austin [1999] proposes DIVA, which uses a simple in-order checker core to verify the execution of a complex out-of-order processor before instruction commit. This allows the dynamic verification of complex and possible error-prone out-of-order superscalar processors by a simple in-order processor with a small execution time overhead. The DIVA approach is able to detect and recover from transient and permanent faults by correcting possible errors with the fault-free checker core.

Mendelson et al. [2000] propose time redundant execution of instructions within an out-of-order processor to detect and recover from transient faults. They extend the reorder buffer to control the redundant execution of single instructions. However, the instructions are not physically replicated, but tagged by a flag to indicate the number of executions. This simple technique allows exploiting underutilised ALUs in the pipeline. The sphere of replication is restricted to only a subset of the pipeline, including the reservation stations and the ALUs. Input incoherence can not occur, since the instruction duplication excludes load instructions.

Ray et al. [2001] propose another mechanism that dynamically replicates instructions in the reorder buffer of an out-of-order pipeline. Here, the sphere of replication includes the reorder buffer and the ALUs. Recovery is provided by the rollback capability of the out-of-order pipeline, which means that in case of an error, the speculative, uncommitted pipeline state is discarded and the execution is restarted at the last committed instruction.

Timor et al. [2010] extend the technique proposed by Mendelson et al. [2000] by a spatial redundant decode stage, which also checks the decode stage before instruction commit.

The microarchitectural techniques presented here have in common that they try to exploit underutilised resources of complex out-of-order processors. Furthermore, all discussed techniques do not redundantly access the main memory by load and store operations, which could lead to input incoherence in the case of I/O operations or interrupts. The efficient redundant execution of instructions in an out-of-order superscalar processor mainly depends on the available parallelism in the pipeline and requires wide out-of-order pipelines [Smolens et al., 2005]. Finally, the techniques are inherently single-threaded, which means that they can not exploit available thread level parallelism (TLP) of multi-core processors.

3.1.3 Thread-Level Redundancy in SMT Architectures

With the advent of simultaneous multithreading processors (SMT) [Tullsen et al., 1995], researchers investigated their use for redundant execution. Rotenberg [1999] proposes a loosely-coupled redundant execution on an SMT processor, called AR-SMT. Rotenberg's approach dynamically duplicates a thread and executes the two redundant streams, denoted as active stream (A-stream) and redundant stream (R-stream). The A- and R-streams are independently executed on the SMT processor. The A-stream stores its computational results in a *delay buffer*, which is used by the R-stream for state comparison. Rotenberg proposes to exploit the asynchronous execution of the A- and R-thread for performance optimisation by using the execution results of the A-thread in the delay buffer for branch and data-flow predictions. Since the R-stream can only commit an instruction when the computational results of both threads do not differ, the R-thread can be used for recovery by discarding the speculative, uncommitted processor state and restarting the execution from the last committed instruction of the R-stream. Possible input incoherence between the A- and the R-stream is prevented by including the main memory in the sphere of replication, which means that the operating system must allocate additional physical memory for the R-stream. Therefore, both streams operate on different physical data, which practically halves the size of the caches for each stream [Reinhardt et al., 2000]. Sundaramoorthy et al. [2000] propose, based on Rotenberg's approach, *slipstream execution.* In the slipstream processor, a program is executed twice, similar to AR-SMT, however, the A-stream running ahead executes only a subset of the instructions of the R-stream. This technique allows performance to be traded against fault-detection coverage.

Reinhardt et al. [2000] propose *Simultaneous Redundant Threading* (SRT). In contrast to Rotenberg, they exclude the main memory from the sphere of replication. Their approach dynamically duplicates the instruction streams as leading and trailing threads, when the instructions are fetched, making the mechanism transparent to the software. Since SRT excludes the memory from the sphere of replication, the authors propose two techniques, which prevent input incoherence between the redundant threads. The *Load Value Queue* (LVQ) allows only the leading thread to access the main memory. The leading thread then puts all loaded values and their addresses in the LVQ. Instead of reading the values from the memory, the trailing thread obtains its input values directly from the LVQ. The *active load address buffer* (ALAB) locks the values read by the leading thread in the shared cache until the trailing thread has loaded the same address. Reinhardt et al. also propose two optimisation mechanisms to enhance the performance of the redundant execution. The *slack fetch technique* executes the trailing thread time-shifted to the leading thread. Since both redundant threads share the same physical addresses, the leading thread can serve as a prefetcher for the trailing thread. The *branch outcome queue* is a hardware queue, delivering the committed branches of the leading thread to the trailing thread. The trailing thread uses the branch outcome queue as branch target buffer to prevent possible branch mispredictions.

Mukherjee et al. [2002] study the implementation of SRT in a "commercial-grade" SMT processor. They extend SRT to support dual-core execution, which can detect transient and permanent faults. However, they also discover that redundant multithreading in a multi-core architecture can significantly suffer from performance degradation due to increasing comparison latencies between the distant cores.

Since the original SRT approach [Reinhardt et al., 2000] only supports fault detection, Vijaykumar et al. [2002] extend SRT by a recovery mechanism, called *Simultaneous Redundant Threading with Recovery* (SRTR), exploiting the rollback mechanism for misspeculated branches of the SMT pipeline for recovery. However, SRTR requires significant changes to the original SRT approach. First, SRTR requires fault comparison before instruction commit. Second, the leading thread can not commit an instruction before comparison with the trailing thread has taken place, since only uncommitted, speculative instructions can be discarded by a rollback. To prevent stalling of the leading thread, while waiting for the trailing thread's results, Vijaykumar et al. modify the SRT approach to not only compare committed instructions but also speculative instructions. However, this simple extension requires to modify all SRT-specific data structures to support speculative state and recovery functionality.

Gomaa et al. [2003] propose *Chiplevel Redundantly Threaded Multiprocessor with Recovery* (CRTR), another SRT variant, which supports chip-level redundant execution and recovery. In contrast to SRTR, CRTR supports a long slack between the redundant threads to hide the effect of long latency comparison between the distant cores. To further reduce the overhead, CRTR allows the leading thread to commit instructions and uses the trailing thread as recovery point when a fault was detected. To reduce the communication bandwidth between the processors, the system tracks dependencies between instructions in order to communicate and compare only the latest value of a register or store operation.

Sánchez et al. [2009] identify atomic instructions as a serialisation point for CRTR, which can introduce significant performance overhead, since the leading and trailing core must be synchronised on each atomic instructions, which reveals the slack between the redundant threads. Therefore, they propose REPAS, which commits the leading thread to the L1 cache in order to make speculative forward progress. However, the speculatively written data is tagged in the L1 cache by an unverified bit to prevent a write back of unverified results to the main memory.

3.1.4 Thread-Level Redundancy in Multi-Core Architectures

Redundant execution mechanisms are also studied in the context of chip multiprocessors (CMPs) and parallel applications. However, as described in Section 2, control-flow based redundant execution of parallel applications poses significant challenges for input replication, redundant thread synchronisation, and output comparison. In this section, we present previous work on loosely-coupled redundant execution on multi-core processors.

[1]*Reunion* [Smolens et al., 2006] is a loosely-coupled redundant execution scheme for shared-memory multi-core architectures. The approach groups cores in a multi-core into redundant execution pairs, while the *vocal core* issues stores and updates the coherence system, and the *mute core* can only read from the memory without updating or manipulating the memory state. The sphere of replication is restricted to the out-of-order pipeline, which means that updates to the registers and the store buffer must be checked for errors, such that the rollback capability of the out-of-order pipeline can be re-used for error recovery. Reunion uses *relaxed input replication*, which allows redundant execution pairs to consume possible diverging data by concurrent memory accesses. Possible input incoherence, caused by relaxed input replication, is detected by comparing the computational results of the vocal and mute cores. However, unlike transient faults, concurrent memory accesses can not always be resolved by re-execution. In this case, *Reunion* uses lockstep execution between the vocal and the mute cores to guarantee forward progress. *Reunion* uses fingerprints [Smolens et al., 2004] to reduce the execution state for result comparison and therefore safes communication bandwidth between the cores.

Dynamic Core Coupling (DCC) [LaFrieda et al., 2007] is a loosely-coupled redundant execution scheme, which supports dynamic core coupling in a multi-core for redundant execution. In DCC, the sphere of replication is restricted to the cores' pipelines, where computational results must be compared at each store. Computational results are buffered in the private caches of the cores, which allows comparison intervals of more than 10,000 cycles. Additionally, DCC provides explicit support for parallel applications and prevents possible input incoherence by a so-called *master-slave memory access window*, which tracks possible input incoherence between redundant threads. The master-slave memory access window is implemented by an *age table* attached to the L1 cache per core. The age table stores the number of executed load/store operations for each memory address. When a store operation in a parallel core has issued an upgrade coherence request, the age tables of all master-slave pairs must be compared in order to identify the possibility of input incoherence between the master and the slave core. Enforcing the order and identifying possible read-write violations between the master and slave cores is implemented by a shared bus, since on each store operation, all cores must snoop on the bus to trigger an age table search.

Rashid et al. [2008] propose another redundant execution approach for parallel applications on a shared-memory multi-core. The redundant threads are executed in a so called *computing* and a *verification wavefront*, which separately manage the coherence among their caches. Redundant threads are compared at the granularity of several thousand instructions, which are called *epochs*. The creation of epochs is driven by the computing wavefront. The computing and the verification wavefront can be executed

[1]This paragraph has also been published in [Weis et al., 2016]

asynchronously, requiring a state management for different epochs, implemented by a *post-commit buffer* per core, which keeps committed results parallel to the L1 cache. When the results of the epochs are verified, the computing wavefront commits the buffered stores in the post-commit buffer to the shared L2 cache. However, in parallel applications the decoupled execution of redundant threads may lead to possible input incoherence between redundant threads. To prevent input incoherence between redundant threads, the communication between the threads is tracked by the cores and so called *subepochs* are created, which enforce the same load/store ordering in the computing and the verification wavefront. However, enforcing the same execution order in the computing and the verification wavefront requires global synchronisation points for each subepoch. To prevent the overhead of strictly enforcing this order in the verification wavefront, a speculative ordering is proposed. When input incoherence is detected by a mismatch of the execution results between the computing and the verification wavefront, the system is recovered and the load/store ordering is strictly enforced. Finally, a combination of strictly and speculative ordering is proposed, which tracks the timing of the races and allows enforcing strict ordering only for races, which are “close in time”, and speculative ordering for races, which are “far apart in time”.

Recently, several proposals discussed hardware transactional memory (HTM) for checkpointing and redundant execution. Sánchez et al. [2010] propose a *Log-based Redundant Architecture* (LBRA), which leverages a log-based HTM to implement redundant execution and recovery. In LBRA, redundant threads are executed within hardware transactions, called p-XACTs. LBRA uses an eager version management and eager conflict detection hardware transactional memory. The redundant execution is transparent to the software. The redundant threads are executed both in a transactional context, which means that store operations of the master thread can commit to the main memory, while the HTM system keeps the old values in a hardware-managed log. The log is used to guarantee input coherence for the master and the slave threads, while the execution of the redundant threads is completely decoupled. A slave thread can obtain values, which have been seen by the master thread before by searching the master thread’s log. LBRA implements local recovery of transactions. However, LBRA allows sharing of unverified data between threads, which means that errors may propagate to several consumer threads before detection. As a consequence, several consumer transactions must be recovered. Therefore, LBRA tracks the producer-consumer dependencies and recovers all dependent transactions.

Yalcin et al. [2013] propose another HTM approach for redundant execution and recovery. Unlike Sánchez et al. [2010], they use a lazy versioning and lazy conflict detection (lazy-lazy) HTM system. They propose to exploit the rollback capability of the HTM system to recover from possible input incoherence in parallel applications.

3.2 Backward Error Recovery in Shared-Memory Multi-Cores

This sections presents related work on checkpointing and recovery mechanisms. While recovery techniques are already covered in some redundant execution mechanisms in the prior section, this section focuses on generic backward error recovery mechanisms for shared-memory multi-core processors.

Based on *Cache-Aided Rollback Recovery* (CARER) [Hunt et al., 1987], which use the cache to buffer computational results of a single-core processor, cache based multi-core backward error recovery mechanisms were developed. Ahmed et al. [1990] and Wu et al. [1990] both propose cache-aided rollback recovery mechanism for checkpointing in multi-cores. While the private caches provide a low-cost, efficient recovery mechanism, Janssens et al. [1994] identify that the checkpoint frequency has high variability and depends on the executed application.

Prvulovic et al. [2002] propose *ReVive*, a global checkpoint mechanism for a distributed shared-memory multi-core processor. A checkpoint in ReVive is established by synchronising all cores, flushing all dirty cache lines, and establishing a log, covering the distributed shared memory. After checkpoint creation, all changes to the distributed shared memory are stored in the log. To support recovery of complete node failures, which means that a complete node with a part of the distributed shared memory became unavailable, ReVive uses distributed parity, which groups pages on different nodes into recoverable "parity groups". The parity groups can be recovered even when a complete node becomes unavailable. ReVive can be implemented in an enhanced directory coherence controller, managing the logging and parity maintenance. However, checkpoint creation can be costly, since all cores in the system must synchronise and the caches flushed before the execution can continue.

Agarwal et al. [2011] present an extension to ReVive by tracking the communication among nodes in the coherence directory, which prevents costly global checkpoint creation and rollbacks, since only a subset of the nodes must establish new checkpoints or recover from errors.

Sorin et al. [2002] propose *SafetyNet*, another global checkpoint mechanism, which also supports faults in the coherence protocol of a distributed shared-memory processor. SafetyNet focuses on transient faults in the processing elements and message loss in the interconnection network, but cannot cope with complete node failures. SafetyNet tries to reduce the checkpointing overhead by overlapping the checkpoint creation with normal execution. Therefore, SafetyNet adds *register checkpoint buffers* and *Checkpoint Log Buffers* (CLBs) to each node. In order to prevent global system stalls during creation of a checkpoint, SafetyNet uses logical checkpoints and considers the caches, the register checkpoint buffers, and the CLBs as safe storage. A checkpoint in SafetyNet includes the state of the register files, the memory values, and the coherence permissions in the caches. Similar to ReVive, SafetyNet uses logging to track the changes to the memory system since the last checkpoint. However, SafetyNet also tracks changes to the coherence state. To save logging storage, only the first modification to an address is logged. Checkpoint creation is coordinated between the nodes such that the nodes create globally consistent checkpoint by checkpointing their local state, including the state of the cores, the caches, and the memory controller.

3.3 Fault-Tolerant Data-Flow Execution

The benefits of data-flow execution for fault tolerance are also studied in the context of different data-flow architectures.

Nguyen-tuong et al. [1996] propose a fault-tolerance scheme for a wide-area distributed system, considering a large-grained data-flow software runtime based on Mentat, a C++ runtime for "high-performance, object-oriented parallel processing systems". The application programmer can select fault-tolerant tasks, which are transformed to a data-flow graph. The fault-tolerant tasks are then executed on different hosts. While Mentat also supports persistent data-flow tasks, the replication focuses on pure data-flow tasks, which are re-entrant and can be executed multiple times on different hosts. The approach can cope with complete host failures in a distributed system. Therefore, they propose "dormant" tasks, which are only executed when the "active" task's host became unavailable.

Jafar et al. [2005] exploits the medium-grained data-flow execution model of KAAPI [Gautier et al., 2007] for a checkpoint/recovery method. KAAPI uses a C++ library on commodity chip multiprocessor clusters that exposes a data-flow execution model. [Jafar et al., 2005] used the functional semantic of the data-flow thread execution of KAAPI for recovery of data-flow threads. They propose two variants. The first creates a checkpoint for each data-flow thread, storing the input data and the task ID to restart a task on a different host. The second variant periodically creates checkpoints at pre-defined points in time or on a work-stealing request of another processor.

Najjar et al. [1990] propose to exploit a data-driven execution model for dynamic distributed checkpointing in large-scale distributed systems. They identify that a data-driven activation of side-effect free tasks can be used for task restarts. This means that the input data of a task is stored on another processor, which allows restarting any unfinished task on a different processor or node.

Cummings [2009] integrates data-flow based rollback into COSMOS, a large-grained data-flow system for distributed systems, implemented in software. Cummings focuses on the data-flow execution model of COSMOS for efficient recovery and assumes a fail-stop behaviour for the processing elements. In the context of COSMOS, the author investigates two different recovery mechanisms, a process restart mechanism, which requires the preservation of data-flow input tokens and a distributed checkpointing mechanism, which allows establishing a global checkpoint.

Recently, data-flow execution model gained new attention to provide fault detection and recovery. Alves et al. [2014] use an abstract data-flow model for redundant execution and re-execution of data-flow threads. They implement fault detection by redundant execution of data-flow threads. The results of the redundant threads are compared by additional commit threads, which are inserted in the data-flow graph. They propose two variants for re-execution of data-flow threads. In the first variant, commit threads have edges to the redundant producer tasks as well as to the subsequent consumer threads, ensuring that subsequent threads can only be started, when the commit stage forwards the correct results. The second variant allows starting subsequent threads without waiting for the commit thread. In this case, the dependencies between the unverified threads must be tracked to be able to re-execute all unverified threads, which have consumed input data from an erroneous thread.

Fu et al. [2014] propose *Data-flow scheduled Redundant Multi-threading* (DRMT) for a data-flow scheduled multi-core processor. DRMT redundantly executes the same version of a program on separate cores, as master and a redundant thread. DRMT uses relaxed input replication and relies on the error recovery mechanism to resolve input incoherence between redundant threads. DRMT executes master and redundant threads on the same SMT core. Threads can be suspended when data can not be read from the register file but from memory. Output comparison is implemented by a *Comparison Buffer* (CB), which is used to store computational results. The CB buffer is structured in different sets and addressable by the redundant threads. In case that the CB set of a redundant thread pair has no entries left, the thread can be suspended, which gives the trailing thread the opportunity to catch up.

3.4 Tolerating Permanent Faults

Since this thesis also covers mechanisms, which can identify and tolerate permanent faults, we also provide related work on systems, which can adapt to permanent faults. Based on Sorin [2009, pp. 90 ff.], we distinguish between adaption mechanisms on core and microarchitectural level.

3.4.1 Deactivation and Reconfiguration of Cores

The permanent disabling of cores can be seen as a state-of-the-art technique to cope with permanent faults in multi-core architectures. The IBM S/390 microarchitecture [Slegel et al., 1999] provides support for cold spare cores. These spare cores can be transparently activated for the running application and the operating system. However, if no spare cores are available, the processor can raise a machine check exception to notify the operating system that the core is permanent faulty and must be excluded by the operating system scheduler. The IBM Blue Gene/Q compute chip [Haring et al., 2012] has 18 processor units (PU), however, to increase the manufacturing yield, only 17 PU are activated, while 1 PU is used as a spare and can be dynamically activated at boot time, when a permanent fault in a PU is detected.

Aggarwal et al. [2007] propose a configurable architecture based on commodity hardware components. The architecture is configured in “fault zones”, which can be separately deconfigured, when a permanent fault occurs. This allows restricting the effect of a permanent fault to the affected fault zone. LaFrieda et al. [2007] investigate core disabling in the context of a loosely-coupled redundant execution scheme. Since DCC allows dynamic coupling of redundant cores, the underlying hardware can be better utilised, since DCC allows to dynamically form new master-slave processor pairs, when a core becomes permanent faulty.

3.4.2 Deactivation and Reconfiguration of Microarchitectural Components

Since deactivating complete cores due to a small permanent faulty component is costly in terms of hardware overhead, some authors also propose fine-grained techniques, which can reconfigure microarchitectural components.

Bower et al. [2005] use the DIVA checker to identify permanent faults in an out-of-order pipeline. They focus on pipeline structures, which are inherently redundant and can be therefore deactivated in the case of a permanent fault. Per reconfigurable entity, they use a saturation counter. The saturation counter is incremented for components, which were used by a faulty instruction. When the counter has reached a specified limit, the hardware unit is considered as permanent faulty and deactivated. S. Gupta et al. [2008] propose *StageNet*, which allows broken pipeline stages to be circumvented in a multi-core by connecting the pipeline stages of different cores by a crossbar. This enables the system to dynamically form a functional pipeline by combining different functional stages of different pipelines. While fine-grained reconfiguration of microarchitectural components may reduce the performance degradation in case of permanent faults, they also require deep changes to the processor pipelines and may increase the hardware overhead and design complexity.

3.5 Summary

This chapter presented an overview of prior research on topics discussed in this thesis.

Section 3.1 discussed related work on redundant execution mechanisms. The presented tightly-coupled lockstep architectures duplicate hardware on different levels, from complete computing nodes of the Stratus ftServer to a replication of the instruction fetch and execute stages in the IBM S/390. However, tightly-coupled lockstepping requires high hardware overhead and strict deterministic execution, which can complicate the use of tightly-coupled lockstep execution in future multi-cores as already pointed out by Bernick et al. [2005]. To remedy the high hardware overhead and strict deterministic redundant execution of tightly-coupled lockstep systems, different loosely-coupled redundant execution mechanisms have been proposed. Loosely-coupled redundant execution can reduce the hardware and execution time overhead, since resources can be shared by the redundant execution streams, while the decoupled streams can serve as prefetchers for each other. Loosely-coupled redundant execution can be implemented in different ways, from loosely-coupled redundant execution of instructions in an out-of-order processor to redundant multithreading in SMT processors or decoupled redundant execution in multi-cores. However, decoupled redundant execution mechanisms, which allow that the redundant execution streams can access the memory independently, require either a complete duplication of the memory for the redundant streams or mechanisms to cope with possible input incoherence. Particularly, input coherence for parallel shared-memory applications can be challenging and requires modifications to the hardware, the caches and can significantly increase the hardware overhead.

Section 3.2 presented backward error recovery mechanisms for shared-memory architectures. Global checkpointing in shared-memory systems must incorporate the cores' contexts and the state of the caches to establish a consistent checkpoint. This either requires stopping the execution and flushing the caches or demands additional hardware structures to store and manage the cache and core contexts.

Section 3.3 described different fault-tolerant data-flow architectures, which show that the data-flow execution principle can provide advantages for the implementation of redundant execution and backward error recovery mechanisms.

Section 3.4 presented proposals, which can deactivate permanent faulty components. The deactivation of cores may also deactivate non-faulty components of the processor. Additionally, deactivating cores in redundant execution systems may further degrade the performance of the system, since the redundant execution unit must be disabled as well. While the fine-grained disabling of microarchitectural components can reduce the hardware costs in case of permanent faults, they require complex changes to a processor's pipeline.

In the next chapter, we will present a data-flow execution model and architecture, which will be enhanced by data-flow based redundant execution, checkpointing, and can cope with permanent and intermittent faults.

4 Baseline Execution Model and Architecture

This chapter describes the baseline data-flow execution model and hardware architecture of this thesis. The baseline execution model and the hardware architecture are subsets of the execution model and architecture developed in the TERAFLUX project[1] [Giorgi et al., 2014b]. The baseline architecture described in this chapter will be extended with redundant execution and checkpointing mechanisms in the Chapters 5 and 6.

The chapter is structured as follows. A description of the baseline data-flow execution model is presented in Section 4.1. The parallel hardware architecture is described in Section 4.2. Section 4.3 provides details on the architectural support for data-flow execution, while Section 4.4 presents the memory organisation of the system.

4.1 A Coarse-Grained Data-Flow Execution Model

This section describes the basic operation principle of the baseline data-flow execution model, which is based on the DTA-C execution model [Giorgi et al., 2007]. The DTA-C execution model is an extension of the Scheduled Data-Flow (SDF) [Kavi et al., 2001] paradigm for clustered architectures [Giorgi et al., 2007]. The original SDF execution paradigm describes a coarse-grained threaded data-flow model, designed to exploit the thread level parallelism (TLP) of applications by decoupling memory accesses from the execution and supporting efficient data-flow thread synchronisation mechanisms. However, the original SDF as well as DTA-C both require special processing units to enable a decoupling of the load/store phase and execution phases. In the TERAFLUX project the DTA-C execution principle has been ported to a commodity x86_64 [AMD, 2013] shared-memory architecture with data-flow hardware enhancements. In contrast to the original TERAFLUX execution model and architecture, this thesis uses subsets of the DTA-C execution model and architecture investigated within the context of the TERAFLUX project.

[1] `www.teraflux.eu`, Retrieved on September 30, 2016

According to the classification of [Yazdanpanah et al., 2013], the execution model of this thesis is a hybrid data-flow/von Neumann execution model, which executes coarse-grained blocks of x86_64 instructions in a data-flow driven way, i.e. the code blocks can be executed when all input data is available. In the rest of this thesis, we will call these blocks *data-flow threads*, or just *threads*. Within the coarse-grained data-flow threads, the execution is control flow driven, i.e. an instruction pointer is used to control the sequential execution of the intra-thread instructions.

A data-flow thread can be comprised of several thousand x86_64 instructions, incorporating control flow, jumps, and function calls. Recursion within a thread is also possible. Each data-flow thread has a dedicated input set. Since the execution is data-flow driven, the threads are only ready for execution when their input set is complete. The synchronisation count represents the number of currently missing inputs of a thread to get started. When the synchronisation count reaches zero, the thread is ready for execution. When a thread has finished its execution, its computational results are published to subsequent threads.

The input sets of waiting threads can be accessed by all threads by special read and write instructions, described in Section 4.3. Since data-flow threads depend only on their own input sets, they can be executed in a decoupled and nonblocking way.

During execution, a data-flow thread uses the following memory sections:

- A *code* section, which keeps the code of the thread. The code section is always read-only.
- A *thread-local storage* (TLS) section, which is dynamically allocated right before the thread is started. Within a thread, read and write accesses to its thread-local stack and heap are allowed. Within the thread-local storage, the data-flow thread keeps temporary run time information, e.g. the stack and the dynamically allocated memory. After the thread has finished execution, the thread-local storage is freed.
- The *thread frame* (TF) keeps the data-flow input set of a thread. Each data-flow thread has a designated TF. The TFs are used for shared-memory communication between data-flow threads. A thread frame is only writable, as long the thread is not ready for execution. That means when all input data has been written to the TF, the TF becomes immutable. The TFs are used for communication between data-flow threads, i.e. threads directly write their results to the TFs' of subsequent threads.
- The *owner-writable memory* (OWM) section [Giorgi et al., 2014a] is part of the globally addressable OWM region and can be accessed by all threads. To prevent race conditions or the usage of locks, only one thread at a time has exclusive read and write access to a specific section of the OWM. To ensure that only one thread can read and write, the data dependencies between threads must ensure that only one thread at a time can access the section.

When the execution of a thread is finished, the computational results of the thread are written to the consumer threads' TFs. Afterwards all private memory sections of a thread, including the TLS and TF, can be freed. This is supported by the data-flow execution principle, since the execution of subsequent threads does only rely on the data stored in their own TF or the OWM. The main difference to a commodity multi-core is that the heap of a thread only stores thread-local data and can not be used for inter-thread communication. Since the TF and the OWM sections store all input data required for the execution of a thread, a started thread is never suspended and hence must never wait for data produced by other threads or to acquire a lock [Giorgi et al., 2007].

This coarse-grained data-flow execution model combines data-flow and control flow driven execution: (1) The efficient control-flow driven execution of the x86_64 pipeline is leveraged, including out-of-order execution and local caches. (2) The decoupled data-flow thread execution supports parallel execution, since the synchronisation between threads is decoupled and nonblocking.

4.2 Baseline Hardware Architecture

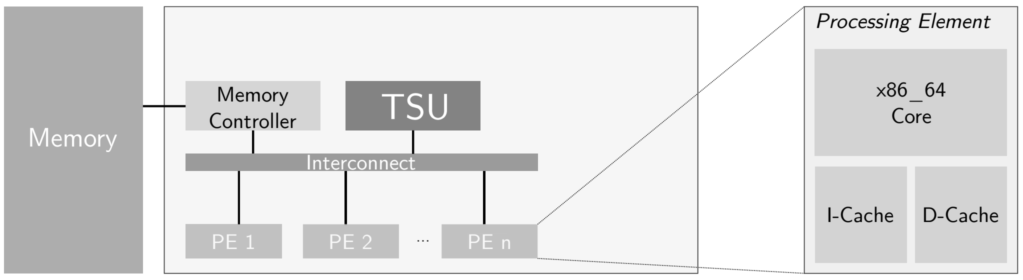


Figure 4.1: High-level system organisation of the baseline system (subset of a Teraflux node).

This section gives an overview of the baseline hardware architecture. Like the execution model, the baseline architecture is also a subset of the Teraflux architectural template [Giorgi et al., 2014b]. Figure 4.1 depicts the high-level organisation of the architecture.

The baseline architecture of this thesis is a conventional shared-memory system, which encloses a number of processing elements (PEs), a *Thread Scheduling Unit* (TSU) to provide data-flow hardware support, and a memory controller to access the main memory. The communication between the components is implemented by an interconnect, which connects the PEs, the TSU, and the memory controller. The interconnect manages all communication within the system, incorporating the PEs, the TSU, and memory accesses to the main memory.

The PEs consist of a standard x86_64 core with private instruction and data caches. Each core comprises a conventional out-of-order x86_64 core, however, the architecture is not restricted to x86_64 cores, which means that simpler in-order cores with a different instruction set architecture would be also possible.

The architecture supports shared-memory communication, which allows that each PE in the system can access any memory location. Hardware support for data-flow thread creation, thread synchronisation, and thread management is provided by the TSU.

4.3 Architectural Support for Data-Flow Execution

The data-flow execution is supported in hardware by the TSU. This section gives an overview of the TSU hardware/software interface and the TSU internals, which are required to control the data-flow thread execution.

4.3.1 The T*-Instruction Set Extension

The hardware/software interface for data-flow execution is implemented by a subset of the T*-instruction set extension, described in [Giorgi, 2012].

These instructions are:

- `tschedule` instructs the TSU to allocate a new thread. After the thread has been allocated, a new frame pointer is returned to the calling PE.
- `tdestroy` indicates the end of a data-flow thread. After reception by the TSU, all private resources, which are currently acquired by this thread are released.
- `tread` allows a thread to read data from its own TF. Memory accesses to other TFs are prohibited by the TSU.
- `twrite` enables a thread to write to the thread frames of subsequent threads. As a side effect, the synchronisation count of the destination thread's continuation (see Subsection 4.3.2) is decreased.

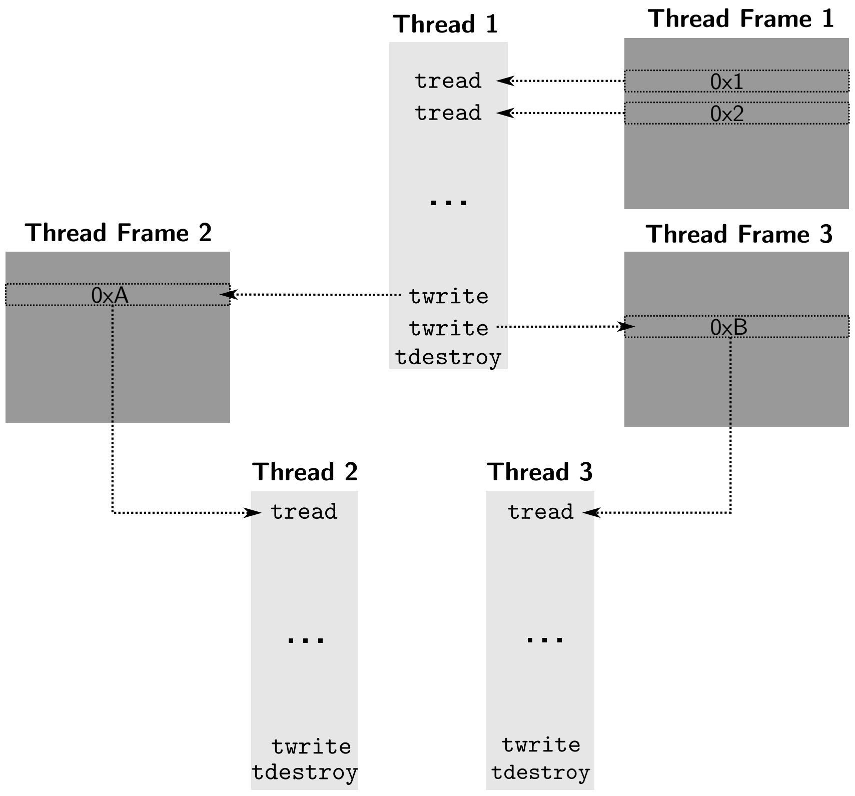


Figure 4.2: Example usage of the T*-instruction set.

For more efficient compilation, Li et al. [2012] have extended the T*-instruction set of Giorgi [2012] to enable the development of data-flow applications by pragmas in the C/C++ programming languages. Since we use the OpenStream compiler for some benchmarks in Chapter 7, we additionally use the following instructions:

- `tdecrease` decreases the synchronisation count of a continuation without manipulating data within a thread frame.
- `twritep` writes directly to the thread frame of a subsequent thread without decreasing the synchronisation count of the thread's continuation.

Example. Figure 4.2 shows the usage of the T* instructions in multiple data-flow threads. Thread 1 is ready for execution when all input data has been written to its TF. In this example, the input set of the thread consists of the integer values $0x1$ and $0x2$. The synchronisation count of Thread 1 becomes zero with the last `twrite` to its TF. Accordingly, Thread 1 becomes ready for execution. Thread 1 will be scheduled immediately, when a PE becomes available. During execution, Thread 1 fetches the input data from its TF, executing a `tread` instructions. When the thread has finished its computation, its results are written to thread frames of subsequent threads, which are waiting for their input data. In this example, Thread 1 writes ($0xA$ and $0xB$) to the TFs of Thread 2 and 3. Afterwards, Thread 1 executes a `tdestroy` instruction, indicating the end of the thread. The TSU can now release all resources of Thread 1, including the TF and the TLS. Since Thread 2 and 3 both require only input from Thread 1, they will be ready for execution after Thread 1 has been finished. When Thread 1 or Thread 2 are ready for execution, they can be scheduled to the next available PE. ⧫

4.3.2 Thread Scheduling Unit

The Thread Scheduling Unit (TSU) provides hardware support for data-flow thread management, thread creation, thread scheduling, and TF accesses.

To manage the threads, the TSU uses a dedicated control structure per data-flow thread called *continuation*. The continuations of the threads are exclusively managed by the TSU. A continuation stores control information about a thread, including the pointer to the TF, the pointer to the code section, the pointer to the TLS and the synchronisation count.

In detail, a continuation stores the following thread information:

- `fp` (*Frame Pointer*): the pointer to the TF is used by the TSU to grant access to the thread's TF.
- `ip` (*Instruction Pointer*): the pointer to the code section is used as initial instruction pointer, when the thread is dispatched to a PE.
- `tlsp` (*Thread-Local Storage Pointer*): the pointer to the thread-local data is used to initialise the stack pointer.
- `sc` (*Synchronisation Count*): controls the required input of a thread. The synchronisation count is decremented, when a `twrite` accesses the TF of a subsequent thread or a `tdecrease` directly decreases the synchronisation count. When all input data of a thread is available, the synchronisation count must be zero.

Figure 4.3 depicts the pointers stored in the continuation and the associated memory sections of a thread.

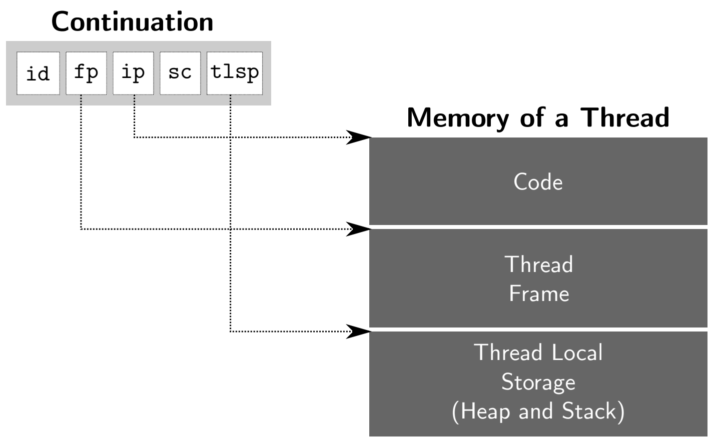


Figure 4.3: Continuation and memory organisation of a data-flow thread (based on [Arandi et al., 2011, p. 18]).

Internally, the TSU uses three tables to manage the different states of a data-flow threads (based on [Arandi et al., 2011, pp. 22 ff.]):

- The *Thread Queue*(TQ) stores the continuations of all threads in the system. In this queue, all allocated threads in the system are waiting for execution. When a PE executes a `tschedule` instruction, the TSU creates a new continuation in the TQ and allocates a new TF in the main memory.
- When the synchronisation count has reached zero and a thread becomes ready for execution (after a `twrite` or `tdecrease`), the TSU creates an entry for this thread in the *Ready Queue* (RQ). The RQ keeps all threads which are ready for execution. Although different scheduling and resource allocation algorithms are possible, the baseline architecture uses a simple FIFO scheduling policy. When a core becomes available for thread execution, the first element in the RQ is selected and scheduled to the available PE.
- When a thread is scheduled to a PE, the corresponding entry in the RQ is deleted and an entry in the *Thread-to-PE-List* (TPL) is allocated, in order to track the current thread-to-PE mapping of the currently running threads.

  When a thread executes a `tdestroy` instruction, its continuation is removed from the TQ and the TPL.

4.4 Physical Memory Organisation

This section describes the different memory regions of the execution model and how these regions are organised in the baseline architecture.

The system's memory is divided in four memory regions, which store the different memory sections of the threads. These regions are the frame memory region, the code memory region, the thread-local storage region and the owner writable memory region. These different regions provide different access semantics, which are described in the following:

Frame Memory The frame memory region stores all TFs of the threads. Access to the thread frames is only possible by explicit `tread` and `twrite` instructions. Furthermore, a thread never writes to its own TF and every thread can write to every thread frame in the system, as long as the synchronisation count has not reached zero. Finally, a thread can only read from its own TF. The accesses to the TFs are controlled by the TSU.

Code Memory The code sections of the threads are stored in the code region. The code region is read-only and can not be modified.

Thread-Local Storage Each thread can access its private thread-local storage section. The thread-local storage sections of all threads are stored in the thread-local storage region. The thread-local storage section of a thread comprises the stack for the execution of the data-flow thread and the heap, where memory can be dynamically allocated during the execution of a thread. The thread-local storage is allocated on thread start and its lifetime is limited by the lifetime of the thread, i.e. the TLS section of a thread is relinquished, when the thread has finished execution.

Owner Writable Memory In the original DTA-C data-flow execution model results can be only published by `twrite` operations to TFs of other threads. However, a main problem of all data-flow architectures is programmability. Programmers are usually used to C-like languages, using memory aliasing, pointers, and in-place updates [Yazdanpanah et al., 2013]. The functional semantic of a data-flow execution model, however, requires resolving dependencies at design time and prohibits in-place updates. To simplify programming and ease the porting of legacy applications, the architecture supports the owner writable memory (OWM) region, which was developed in the TERAFLUX project [Giorgi et al., 2014b]. A data-flow thread can access the OWM by a pointer. Within a thread, in-place updates to the OWM are allowed. To prevent race conditions between different data-flow threads, only one thread at a time can access a specific part of the OWM. To prevent different threads accessing the same part at the same time, the programmer must insert data dependencies between those threads. Therefore, the TSU ensures that threads, accessing the same addresses in the OWM region, are executed subsequently. This technique may reduce potential parallelism by unnecessary dependencies between threads, but simplifies data communication between threads by just copying a pointer to the TF of subsequent consumer threads.

4.5 Summary

This section described an x86_64 based data-flow execution model and its integration in a multi-core architecture with data-flow hardware extensions. The data-flow execution can be controlled by the TSU, a dedicated data-flow thread scheduling unit, which starts data-flow threads on availability of their input data. Thread creation, thread destruction, and inter-thread communication is supported the T*-instruction set extensions. Furthermore, the data-flow architecture supports shared-memory communication by the OWM region.

In detail, the presented execution model and architecture provides the following benefits:

- Efficient data-flow thread execution, exploiting complex x86_64 out-of-order pipelines and caches.
- Decoupled data-flow driven scheduling of data-flow threads: threads are only started, when their input values are available. As a consequence, locks for synchronisation between data-flow threads are not necessary and synchronisation between data-flow threads is nonblocking.

In the next chapter, the described baseline execution model and architecture will be extended by data-flow based fault-tolerance mechanisms.

5 Fault-Tolerant Data-Flow Execution

This chapter proposes redundant execution, thread restart, and fault diagnosis mechanisms for the baseline data-flow execution model, presented in Chapter 4.

The chapter is structured as follows. Section 5.1 gives an overview of the proposed data-flow based mechanisms for fault detection, recovery, diagnosis, and adaption. Section 5.2 describes the extended baseline architecture, including hardware enhancements for fault-tolerant data-flow execution. Section 5.3 proposes *double execution* of data-flow threads, a data-flow based loosely-coupled redundant execution scheme, while Section 5.4 presents a restart mechanism for data-flow threads. Section 5.5 describes the necessary enhancements to the data-flow runtime system, which are required for double execution and thread restart recovery. Section 5.6 proposes a fault diagnosis scheme to identify permanent and intermittent faulty processing elements. The proposed fault-tolerance techniques are summarised in Section 5.7.[1]

5.1 Overview of the Data-Flow Based Fault-Tolerance Mechanisms

5.1.1 Fault-Tolerant Coarse-Grained Data-Flow Execution

The central idea of this thesis is to exploit the baseline data-flow execution model, described in Chapter 4, for efficient and scalable redundant execution and recovery. Based on the proposed fault detection and recovery mechanisms, we further present a fault diagnosis and adaption scheme to handle intermittent and permanent faulty PEs.

In the following, an overview of the data-flow based fault-tolerance mechanisms is presented.

Fault Detection by Redundant Execution Faults are detected by *double execution*, a data-flow based redundant execution mechanism. Double execution is a pure hardware

[1]Parts of the Sections 5.1–5.5, including the description of the double execution and the thread restart mechanism, the fault-tolerant hardware architecture, and the extended continuations, have been published in [Weis et al., 2011a] and [Weis et al., 2016].

mechanism and completely transparent to the programmer. Since double execution uses data-flow threads for redundant execution, the architectural support for the data-flow thread execution available in the baseline architecture can be re-used.

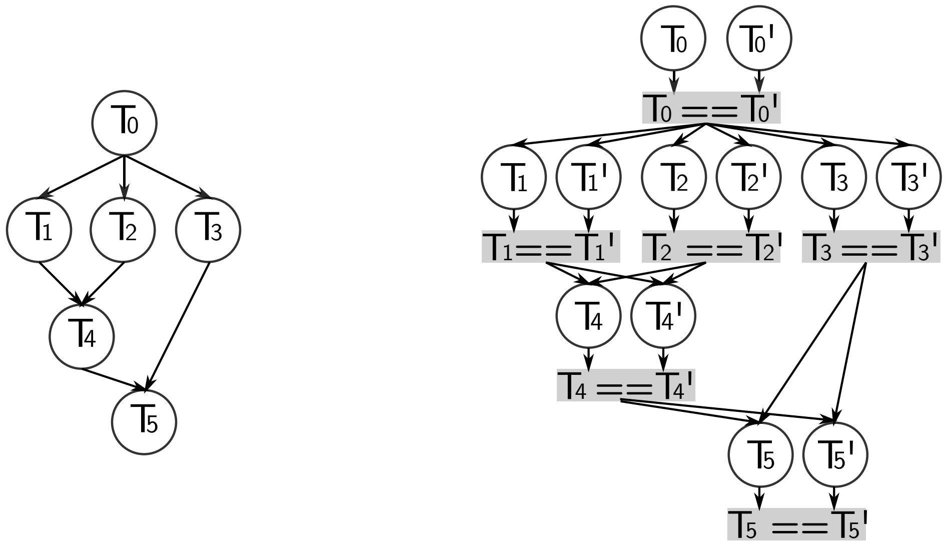


Figure 5.1: Dynamic dependency graph of a regular data-flow execution (left) and double execution (right).

Example. Figure 5.1 shows the dependency graph of a data-flow application (left) and the dynamically created dependency graph of the same application during double execution (right). A thread is dynamically duplicated, when it becomes ready for execution, i.e. all input data is available. Since the execution of a data-flow thread has no side effects, duplicated threads can be independently executed from each other. To safe communication bandwidth and reduce the comparison overhead, a CRC-32 [Koren et al., 2007, p. 76] signature of the output of the threads is created. When both threads have finished their execution, the CRC-32 signatures of the redundant threads are compared and subsequent threads can be started. ♦

Double execution promises the following advantages over prior loosely-coupled redundant execution mechanisms:

(1) Input incoherence induced by race conditions of shared variables is impossible, since the hardware architecture, i.e. the TSU, is aware of data dependencies between threads, which prevents possible input incoherence. Threads are only started when their input data is available. This means that the input data of redundant threads is immutable, when the synchronisation count of a thread has reached zero. (2) Result comparison is only necessary, when data is exchanged between data-flow threads. (3) The hardware support for data-flow thread execution can be re-used for the management and the synchronisation of the redundant threads.

Recovery by Thread Restarts A recovery mechanism must support low overhead checkpointing in order to create frequent checkpoints and also provide low overhead in case of faults.

The data-flow execution supports the restart of threads by its side-effect free execution semantic. Side-effect free means that a data-flow thread does not affect the state of the system until the computational results of the redundant threads have been compared and committed. Since a data-flow thread only depends on its immutable TF and OWM sections, an error can be recovered in a PE by discarding the current execution state of the PE and restarting the data-flow thread.

Diagnosis and Adaption to Permanent and Intermittent Faults To tolerate permanent and intermittent faults in the PEs, a faulty PE must be identified and the system must be able to circumvent the PE. The fault localisation mechanism of this thesis is based on double execution and thread restart recovery to identify permanent and intermittent faults in the PEs by majority voting. When a prolonged fault is detected by recurring thread re-executions on a PE, the data-flow runtime system deactivates the faulty PE. Since the PE may only suffer from an intermittent and not a permanent fault, the data-flow runtime periodically starts threads on the faulty PE in order to be able to re-activate the PE, when the fault has disappeared.

5.1.2 Faults Covered by the Data-flow Based Fault-Tolerance Mechanisms

The fault-tolerance mechanisms proposed in this chapter can detect and tolerate the following faults in the PEs of the baseline architecture:

Transient Faults in the PEs Increasing transient fault rates of future semiconductor devices, induced by environmental radiation or other external noise, require mechanisms to detect and recover from transient faults to prevent frequent catastrophic breakdowns of the system. Double execution can detect transient faults in the PEs by comparing the produced results of the data-flow threads, while the thread restart mechanisms can recover from these faults by restarting them.

Intermittent and Permanent Faults in the PEs The rising variability of the semiconductor manufacturing process and increasing wearout effects demand that fault-tolerance mechanisms also incorporate manufacturing and wearout induced intermittent and permanent faults over the whole lifetime of the processor. This requires dynamic mechanisms to identify and adapt the system at runtime. The diagnosis and adaption mechanism proposed in this chapter can localise and detect intermittent and permanent faulty PEs, which can be deactivated by the TSU. After a certain time period, the system re-activates the PE in order to test, whether the PE is still faulty.

Limitations Double execution detects faults when the execution of the data-flow threads is compared. However, faults in the control flow of a data-flow thread can only be detected when a `tdestroy` instruction is executed in both redundant threads and their results compared. This means that faults, which cause an endless loop, may never reach a `tdestroy` instruction and therefore result comparison never takes place. Furthermore, the fault diagnosis scheme relies on the assumption that faults in different components never cause exactly the same effect. However, multiple faults at the same time in different components are very seldom [Mukherjee, 2008].

5.2 Extended Fault-Tolerant Hardware Architecture

Several additional architectural extensions are required to support data-flow based fault detection, recovery, diagnosis, and adaption. Figure 5.2 gives an overview of the fault-tolerant hardware extensions. In the rest of this section, we will describe these additional hardware extensions in more detail.

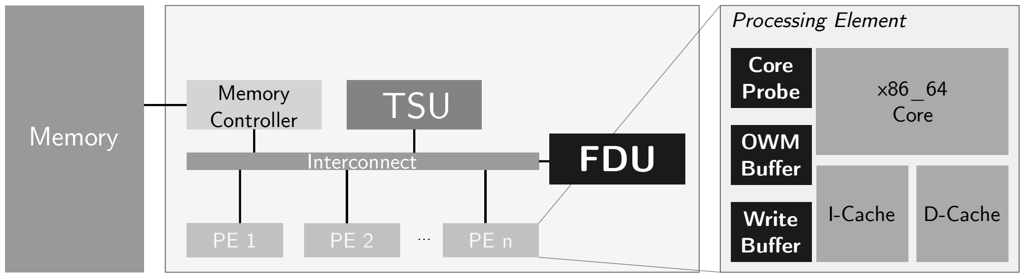


Figure 5.2: Extended fault-tolerant hardware organisation.

Fault Detection Unit The central hardware component of our fault-tolerance approach is the so-called *Fault Detection Unit* (FDU). The FDU manages the result signatures of double execution and forwards signatures of redundant threads in order to speed up result comparison (see Subsection 5.3.4). Furthermore, the FDU is responsible for fault diagnosis of permanent and intermittent faulty PEs. Therefore, the FDU stores the signatures of the redundant thread executions, when a thread is restarted. This allows the FDU to localise the faulty PE by majority voting and also to distinguish between permanent, transient, and intermittent faulty PEs. Whenever the FDU identifies a PE as permanent or intermittent faulty, it notifies the TSU about the fault, which will deactivate the PE.

Core Probe The core probe is a small hardware unit implemented on each PE to control the data-flow thread commit and signature comparison. In particular, the core probe creates the CRC-32 signature of the output of a data-flow thread. This signature is later used for comparison by another core probe. Furthermore, the core probe manages and controls the memory accesses of the data-flow threads. When a data-flow thread commits a `tdestroy` instruction, the core probe sends its current CRC-32 signature to the FDU. Finally, the core probe is able to compare received signatures from redundant threads.

Write Buffers Although the data-flow execution model supports side-effect free execution, the underlying architecture must additionally guarantee isolation between the committed and uncommitted execution states of the data-flow threads. This isolation is implemented by two additional write buffers, which are used to buffer uncommitted execution state of a thread until the redundant threads are compared.

Both buffers are similar to the write buffers proposed in [Hammond et al., 2004] for hardware transactional memory with pessimistic version management and are managed by the core probe.

- Write Buffer: `twrite/twritep/tdecrease` instructions executed by a PE may contain errors. In order to guarantee fault isolation and hence avoid the possibility that errors can modify the global state of the system, i.e. manipulating the synchronisation count of a wrong thread or overwriting data at the wrong address, buffering of those instructions is necessary. The write buffer stores all `twrite/twritep/tdecrease` instructions until the core probe has compared its signature with the signature of the redundant thread. When no error was detected, the PE then forwards all buffered instructions to the TSU, which processes them in the usual way, i.e. decreases the target thread's synchronisation counts and stores data in the TFs of subsequent threads.
- OWM Buffer: Stores to the OWM sections must be also isolated. Therefore, to identify OWM accesses of a thread, the core probe stores the start pointer and the length of the complete OWM region. Whenever a store accesses this region, the store is buffered in the OWM buffer until the thread finishes execution and is allowed to commit. By contrast to the write buffer, the OWM buffer must be fully associative, since values stored in the write buffer may be re-used by the same thread.

5.3 Double Execution of Data-Flow Threads

Double execution (DE) is a loosely-coupled redundant execution scheme, which supports decoupled redundant execution of data-flow threads. Compared to prior loosely-coupled redundant execution mechanisms, like [Rotenberg, 1999; LaFrieda et al., 2007; Rashid et al., 2008; Reinhardt et al., 2000], double execution makes use of the data-flow execution principle for fault detection. In particular, data-flow execution is leveraged for *input replication*, *redundant thread synchronisation*, and *output comparison*. Beside the advantages inherent to loosely-coupled redundant execution mechanisms (see Section 2.3), double execution supports decoupled, asynchronous execution of redundant data-flow threads, which does not require hardware based permanent coupling between redundant PEs, reduces the performance degradation when PEs are permanently faulty and provides support for asynchronous execution of redundant threads.

5.3.1 Double Execution Overview

In the following, we will describe how the data-flow execution model facilitates *input replication*, *thread synchronisation*, and *output comparison.*

[2]A thread is duplicated when its synchronisation count becomes zero, i.e. a thread has received all its inputs and is ready for execution. Similar to [Reinhardt et al., 2000], we call the duplicated thread *trailing thread* and its copy *leading thread.* The terms trailing and leading are used to distinguish between the start time of the threads, which means that the leading thread is always started before the trailing thread. However, due to the decoupled execution of both threads, it may happen that the trailing thread finishes before the leading thread. To duplicate a thread, the TSU, which keeps all continuations in the system, creates a copy of the original continuation. To indicate the thread's

[2]The rest of this section is a revised part of [Weis et al., 2016].

duplication, the TSU sends a notification message with the thread IDs of the leading and the trailing thread to the FDU. While the redundant threads have different continuations and IDs, they share the same TF and OWM. Furthermore, the TSU scheduler ensures that leading and trailing threads never share the same PE to support the localisation of permanent and intermittent faults.

After the continuation has been copied, both threads can be independently scheduled and executed on different PEs without synchronisation. During execution, the core probe creates a CRC-32 signature of the threads' output.

When the leading thread finishes execution, the core probe sends the CRC-32 signature to the FDU, which immediately forwards the signature to trailing thread's core probe, The core probe of the trailing thread's PE compares its signature with the forwarded signature from the FDU and signals the TSU and the FDU the commitment or a detected error. In the fault-free case, the trailing thread is committed, similar to the execution in the baseline system. When a fault has been detected by the core probe, the FDU triggers a thread restart, described in Section 5.4.

Figure 5.3 shows the execution stages of double execution:

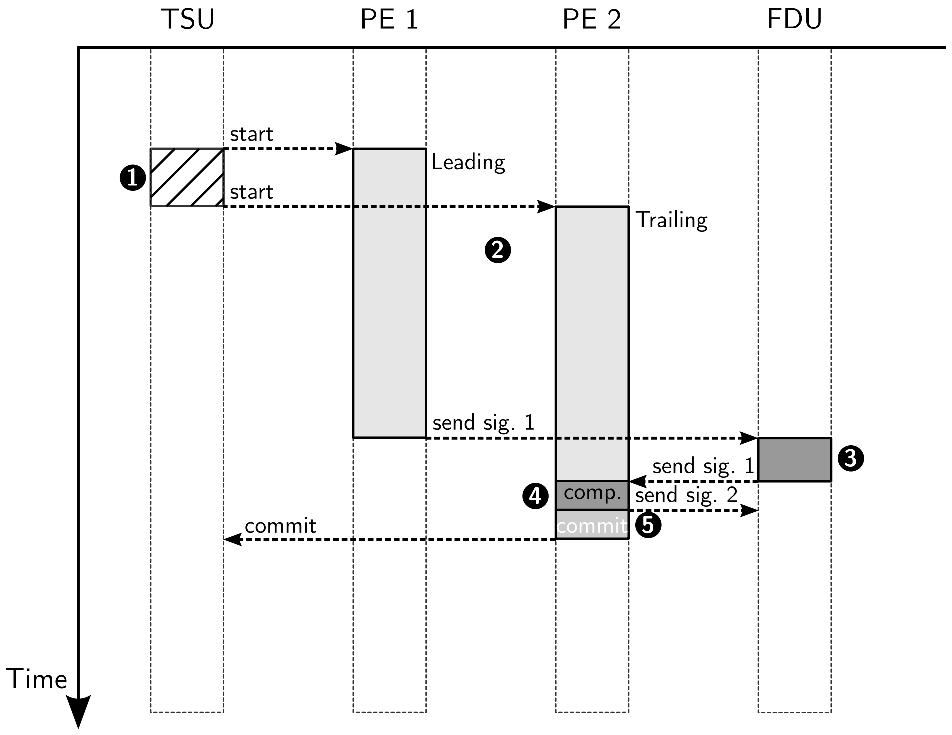


Figure 5.3: Execution stages of double execution.

❶ **Thread start** The leading thread is immediately started when a PE becomes available. The trailing thread is started when the next PE becomes available. The time between the start of the leading thread and the trailing thread is called *start slack*. Since threads are started on their availability, the *start slack* is influenced by the utilisation of the system.

➋ **Thread execution** During execution, the PE of the trailing thread buffers its results in the write buffers (write buffer and OWM buffer). Simultaneously, the core probe creates a CRC-32 signature the thread output.

The core probe of the leading thread's PE also creates a CRC-32 signature of its output, however, after signature creation, the output of the leading thread can be discarded, with exception of the OWM writes, which can be only discarded after the leading thread has reached the `tdestroy` instruction.

➌ **Thread end:** When the leading thread has finished execution, indicated by a `tdestroy` instruction, the core probe sends the CRC-32 signature to the FDU, which immediately forwards the signature to the trailing thread's core probe. Since the output of the leading thread was not buffered, the PE of the leading thread is immediately ready for execution of the next waiting data-flow thread in the TSU's RQ. By contrast, the trailing thread's PE has buffered the trailing thread's output and must wait until signature comparison allows the commitment.

➍ **Output comparison:** The core probe of the trailing thread's PE compares the signatures of the redundant threads. Additionally, the FDU, which has been notified by the TSU on duplication of the redundant continuation, waits for the signatures of the leading and the trailing thread, compares them again and stores the result per PE. In case of a mismatch of the signatures, the FDU keeps the signatures for later majority voting in order to localise the faulty PE.

➎ **Thread commit or recovery:** In case of a non-faulty execution of both threads, the core probe of the trailing thread commits the write buffer and the OWM buffer. After commit of the trailing thread, the TSU deletes the continuations of both threads. When a fault was detected, the core probe of the trailing thread's PE flushes the buffers and informs the TSU to trigger a thread restart.

5.3.2 Sphere of Replication of Double Execution

As described in Section 2.3, the *sphere of replication* defines the hardware region of a redundant execution scheme, where components are either spatial or temporal redundant. Within the sphere of replication faults can be detected by a redundant execution mechanism. However, input data that enters the sphere must be replicated in a consistent manner for both execution streams. Furthermore, data that leaves the sphere of replication must be checked for faults, otherwise faults can not be detected at a later stage.

Figure 5.4 shows the sphere of replication of double execution for the enhanced fault-tolerant architecture. The sphere is restricted to the PEs of the system, excluding the local caches, the core probes, and the write buffers. As a consequence, these hardware components must be protected by other fault-tolerance mechanisms.

Beside the PEs, the baseline architecture also includes several single-point-of-failure hardware components, e.g. the interconnect, the memory controller, the FDU, and the TSU. For the interconnect, we assume an end-to-end ECC implementation, which is able to detect and correct transient faults. Permanent faults within the interconnect are out of scope of this thesis, nevertheless fault-tolerant network-on-chip techniques, as they are discussed in [Garbade, 2014], can be additionally implemented.

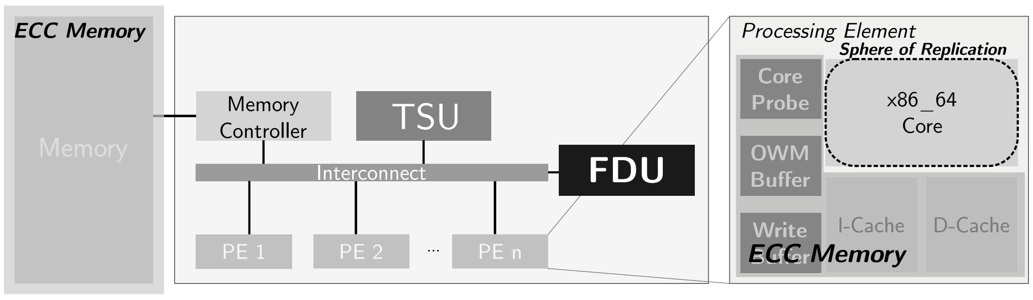


Figure 5.4: Sphere of replication for double execution.

Furthermore, all off- and on-chip memory structures, which are not part of the sphere of replication must be protected by information redundancy mechanisms as well, e.g. ECC. In the case of a permanent broken DRAM chip, chipkill memory [Dell, 1997] could be implemented to dynamically cope with broken memory chips. Fault-tolerant memory is of particular importance, since the data-flow based fault-tolerance mechanisms require the main memory to store the current state of the system.

For all other non-redundant components, e.g. the memory controller, the FDU, or the TSU, we do not assume additionally fault-tolerance mechanisms, however, spatial redundancy can be implemented for all of them. Nevertheless, spatial redundancy of these components does not directly influence the data-flow thread execution and is therefore left for future work.

5.3.3 Input Replication

As double execution uses data-flow threads for redundant execution, the input data of the redundant threads must be consistently replicated. The following input values may enter the sphere of replication during execution:

- **TLS Reads**: The redundant threads can read from the TLS, which stores a thread's private stack and heap. Since the TLS is private per data-flow thread and exclusively created for each thread, it must not explicitly replicated for double execution.
- **OWM Reads**: Writes to the OWM region must be buffered until the redundant threads are verified. Hence, writes to the OWM region can only become globally visible when the thread commits. This means that reads from the redundant threads can share the same OWM section, since the OWM section will not change until the execution of the trailing thread is verified and committed.
- **Thread Frame Reads**: The TF is immutable after the synchronisation count has reached zero, which allows redundant threads to read from the same TF, because data inconsistencies between redundant threads, induced by race conditions of concurrent `twrite`s, are impossible.

[3]Beside the read accesses discussed before, the leading and trailing threads may execute `tschedule` instructions to dynamically allocate subsequent threads. If the TSU receives

[3]This paragraph has also been published in [Weis et al., 2016].

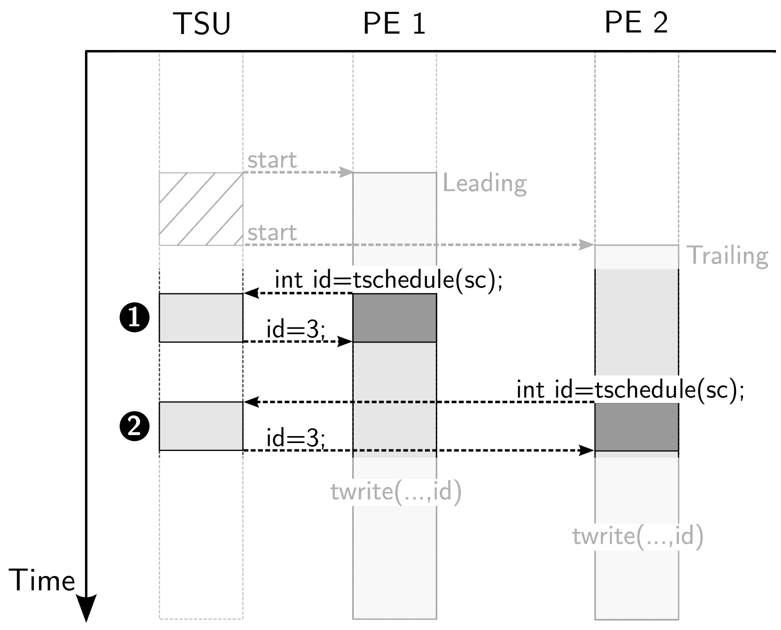


Figure 5.5: Consistent input replication for redundant `tschedule` instructions.

a `tschedule` request from a PE, a new continuation is created and a new TF, which is required to store the thread's input data, is allocated. The TSU finally returns the ID of the newly created thread. However, the returned ID is run time dependent and may be later passed to subsequent TFs by `twrite` instructions. To ensure equal output for the leading and trailing threads, the received IDs must be equal for the trailing and the leading threads. However, the redundant thread execution is not synchronised on instruction level and it may happen that a leading thread runs behind its trailing thread or vice versa. In order to prevent stalls induced by `tschedule` synchronisation between leading and the trailing threads, we allow both PEs to issue `tschedule` instructions. To guarantee consistent IDs in the leading and the trailing thread, the TSU follows the algorithm depicted in Algorithm 5.1. Therefore, the TSU maintains a counter per continuation (*tsched_count*), which keeps the number of issued `tschedule` instructions per thread. When a thread has issued a `tschedule` request, the TSU compares the *tsched_count* in its continuation with the *tsched_count* of the redundant thread. If the *tsched_count* is greater than the *tsched_count* of the redundant thread, the TSU knows that the calling thread is running ahead of its redundant copy. In this case, the TSU processes the `tschedule` request as usual and stores the created thread ID in the ID Table (see Section 5.5.2). If the `tschedule` count is lower than the *tsched_count* of the redundant thread, the TSU has already processed this `tschedule` request from the redundant thread, which is running ahead. In this case, the TSU proceeds with an ID Table look up to retrieve the previously stored thread ID, created by the redundant thread's `tschedule`, which is running ahead.

Example. Figure 5.5 depicts an example execution of a redundant `tschedule` execution. ❶ It can be seen that PE 1 executes the leading thread, which issues a `tschedule` instruction. The TSU now compares the *tsched_count* of the leading thread with the *tsched_count* of the trailing thread. Since no thread has executed a `tschedule` at this point, the TSU processes the `tschedule` and increments the *tsched_count* of the leading thread and stores the new thread ID ($id = 3$) in the ID Table. ❷ Afterwards, PE 2, which executes the trailing thread, executes a `tschedule` instruction. The *tsched_count* of the leading thread is now greater ($tsched\_count_{leading} > tsched\_count_{trailing}$) than the trailing thread, in this case the TSU retrieves the previously stored thread ID from the ID Table ($id = 3$). ♦

Algorithm 5.1 Replication of a thread ID for a redundant `tschedule` instruction.

```
Input: $sc$ // synchronisation count
Output: $id = tschedule(sc)$
  if $tsched\_count_{caller} \leq tsched\_count_{redundant}$ then // thread is behind
    $id \leftarrow retrieve$ from ID Table
  else // thread is ahead
    $id \leftarrow$ process $tschedule(sc)$
  end if
  return $id$
```

5.3.4 Synchronisation and Output Comparison

Leading and trailing threads are synchronised for comparison on thread commit. Likewise, the synchronisation frequency depends on the length of the data-flow threads. As a consequence, the granularity of double execution thread synchronisation can not be adapted at run time and depends on the data-flow application. Unfortunately, faults can only be detected when the leading and trailing thread synchronise, which means that the worst case fault detection latency depends on the start slack of the redundant threads, their execution speed, and the length of the threads.

Data, which leaves the sphere of replication and changes the global state of the system must be verified for errors. For data-flow threads, the following output leaves the sphere of replication and can manipulate the global system state:

- **Writes to the thread frame**: Data-flow threads use `twrite`/`twritep` instructions to write to TFs of subsequent threads. Furthermore, `tdecrease` instructions manipulate the synchronisation count of waiting threads in the TSU.
- **Writes to the OWM**: Writes to the OWM become only visible after a thread is committed.
- **Scheduling of new threads**: Data-flow threads can spawn new threads by `tschedule` instructions.

All other memory accesses of the threads are only allowed to manipulate thread-local data, which does not directly influence the global state of the system. However, an error that has been temporary stored in the thread-local storage may be later passed to subsequent thread frames or the OWM region.

Therefore, output comparison for double execution needs to incorporate all `twrite`/`twritep`/`tdecrease` buffered in the write buffer, writes to the OWM region buffered in the OWM buffer, and all executed `tschedule` instructions.

Result Compression In order to save interconnection bandwidth and result comparison latency, the output of a thread is reduced to a CRC-32 signature, similar to the fingerprint technique proposed by Smolens et al. [2006]. As a consequence, the CRC-32 signature is updated on every `twrite/twritep/tdecrease`, every write to the OWM buffer, and every `tschedule`. For `twrite` and `twritep` instructions, the target thread's ID, the address and the value must be incorporated in the signature. For stores to the OWM section the address and the value must be incorporated. For the `tschedule`, the address of the thread's code region and the synchronisation count must be incorporated in the signature. For the `tdecrease`, the target thread's ID and the value are incorporated.

Compared to [Smolens et al., 2006], double execution does not require creating a signature of the complete context of each PE and all store instructions, which reduces the pressure on the CRC hardware in the core probe.

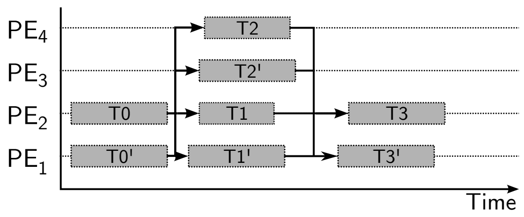
Reducing the PE Blocking Time Double execution implements fault isolation on PE level, which simplifies the recovery mechanisms, as proposed in Section 5.3. However, the output of the trailing thread must be buffered in the PE until the signatures of both threads have been compared. As a consequence, the PE that has executed the trailing thread is blocked until commit or recovery. Furthermore, when the leading thread finishes after the trailing thread, the waiting time increases. Although this method supports fast recovery by only discarding the execution state of the trailing and leading thread's PE, it may introduce additional overhead by blocking of the PE until signature comparison, because the blocking time of the processing element depends on the comparison latency. This means that the comparison latency is on the critical path of the execution, since it directly increases the PE blocking time of the trailing thread.

However, the *commit slack*, i.e. the time difference between the commit of the trailing and the leading thread, can be used to mask the comparison latency by transferring the signature of the leading thread to the trailing thread's PE core probe, while the trailing thread is still executed. The evaluation results in Chapter 7 show that this technique can significantly reduce the average PE blocking time of the trailing thread.

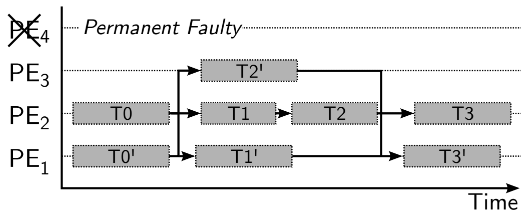
5.3.5 Asynchronous Thread Execution

After the synchronisation count of a thread has reached zero and a thread is duplicated, both threads can be independently scheduled. This means that both threads can be executed completely asynchronous on different PEs. The start slack depends on the PE availability at run time. Nevertheless, the TSU tries to execute leading and trailing threads in parallel to limit the start and commit slack. The decoupled redundant thread execution allows for flexible utilisation of the available PEs. In the case of permanent faults, the system can efficiently exploit the remaining PEs even when an uneven number of PEs is left operational.

Example. Figure 5.6 shows the flexibility of the asynchronous thread execution on a system with 4 PEs. In Figure 5.6(a) it can be seen that the threads T1, T1', T2, and T2' are executed in parallel. However, when in Figure 5.6(b) PE_4 becomes permanent faulty, the TSU can only execute T2' on PE_3 and must wait for the next available PE to schedule T2. Since the next available PE is PE_2, the TSU schedules T2 to PE_2. In this case, T2 is executed completely time-shifted with respect to the leading thread T2'. ♦



(a) Double execution on four PEs.



(b) Double execution on four PEs with one permanent faulty PE.

Figure 5.6: Asynchronous execution of redundant threads.

5.3.6 Influence of Double Execution on the Execution Time

Beside the doubled PE utilisation inherent to all redundant execution schemes, the overhead of double execution compared to conventional data-flow execution is influenced by two factors: The *deferred commit* and the *PE blocking* of the trailing thread.

Deferred Commit For the global progress of the system, double execution looks like non-redundant data-flow execution, since only the trailing thread is allowed to commit. However, compared to non-redundant data-flow execution, the time between start (of the leading thread) and commit (of the trailing thread) is longer than for a non-redundant execution of the same thread. The deferred commit of the trailing thread can reduce the parallelism in the system, since subsequent threads can only be started when both the leading and the trailing threads have finished their execution.

PE Blocking of the Trailing Thread While the PE of the leading thread is immediately relinquished for subsequent threads after `tdestroy` has been retired, the trailing thread has to buffer the results of the data-flow execution in its write buffers. During result comparison, the PE can not execute subsequent data-flow threads, since the PE's write buffers would be otherwise overwritten. Therefore, the PE of the trailing thread is blocked for further data-flow threads until the FDU has finished result comparison. The blocking time of the PE is minimised, when the leading thread always finishes execution before the trailing thread. In this case, the PE is blocked for other leading threads until the core probe has received and compared the signature from the leading thread. The commit slack between the redundant threads can reduce the PE blocking time induced by the result comparison latency, as described in Section 5.3.4.

5.4 Restart of Data-Flow Threads

This section describes the data-flow based thread restart mechanism to recover from errors. The buffers of the PEs prevent that an unverified trailing thread can modify the global state of the system, i.e. TFs of subsequent threads, the synchronisation counts of waiting data-flow threads in the TSU, or the OWM region. As a consequence, error propagation for *double execution* is restricted to the PEs. Since no global state is manipulated until double execution has verified the thread execution, the main memory must not be recovered. This means that the data-flow thread execution creates inherent execution checkpoints on each thread commit. The valid and fault-free checkpoint state consists of the global state of the system, i.e. the thread frames and the OWM region in the main memory and all continuations within the TSU, which are not marked as speculative (see Subsection 5.4.1). When a redundant thread pair commits, a new checkpoint is implicitly created. Hence, the unsafe, recoverable state consists of the execution state of a thread, including the PE context, the buffers, and the TLS.

When the core probe detects a signature mismatch, the TSU discards the uncommitted state of the faulty threads, i.e. the buffers of the waiting thread's PE's and all speculatively created continuations. Afterwards, the TSU restarts the redundant threads.

The overhead in case of a fault is dominated by the wasted execution time for the restarted threads. Since double execution can detect errors only when both threads commit, the wasted execution time depends on the start slack, the length of the data-flow threads of the executed application, and the execution speed of the redundant threads.

5.4.1 Speculative Thread Creation

Data-flow threads can schedule new threads by executing the `tschedule` instruction. In this case, a new continuation is created within the TSU and a new TF is allocated. However, in case of a thread restart, the same `tschedule` instruction will be executed again. To prevent multiple thread creations in such a case, all scheduled threads are marked as speculative in the TSU until the parent thread has been checked for errors. When a redundant thread pair is restarted, all speculatively created threads must be discarded. In order to do so, the ID of the parent trailing thread is stored in each continuation. In case of a thread restart, the TSU deletes all speculative continuations created by the erroneous thread and releases the allocated TFs of the threads.

Please note that the TSU will never schedule a thread to a PE when its continuation is marked as speculative. After result comparison and commitment of the trailing thread, the TSU changes all successor threads from speculative to non-speculative.

5.5 Data-Flow Runtime Enhancements

Double execution requires changes to the baseline execution model and architecture. In particular, the data-flow runtime must be able to create and manage redundant threads, create CRC-32 signatures, and support deterministic `tschedule` input replication for the redundant threads.

This section describes the necessary enhancements to the TSU to support double execution and thread restart recovery.

5.5.1 Extended Continuation for Double Execution

Figure 5.7 shows the extended continuations of the trailing and leading thread with the additionally allocated thread local storage for the redundant thread. We added the following fields to the original continuation:

- *trail* (Trailing Thread): indicates, whether the thread is a trailing thread.

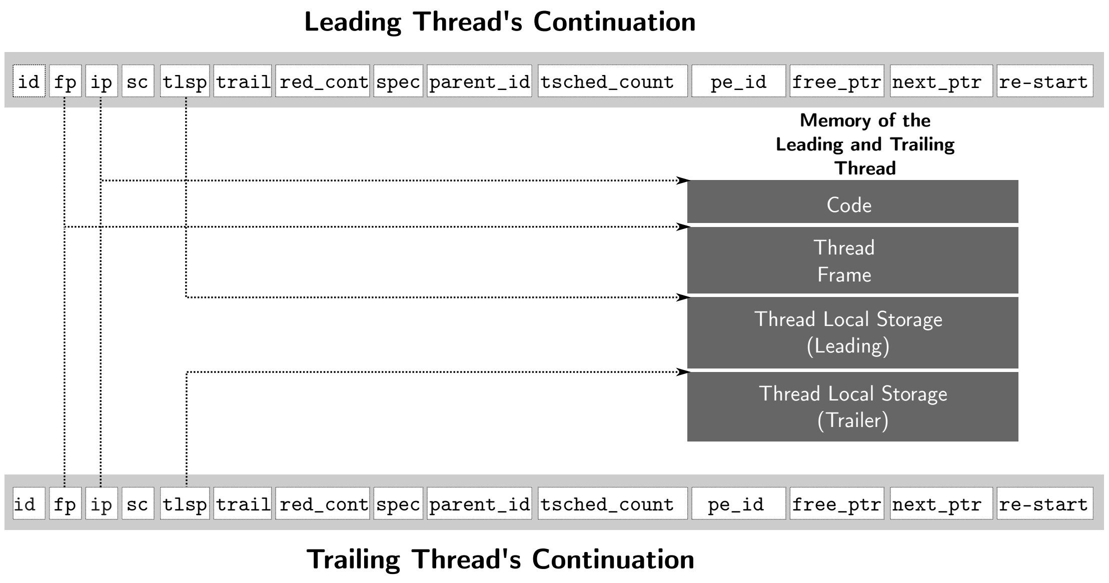

Leading Thread's Continuation
id
fp
ip
sc
tlsp
trail
red_cont
spec
parent_id
tsched_count
pe_id
free_ptr
next_ptr
re-start
Memory of the Leading and Trailing Thread
Code
Thread Frame
Thread Local Storage (Leading)
Thread Local Storage (Trailer)
Trailing Thread's Continuation

Figure 5.7: Redundant continuations and memory organisation of double execution.

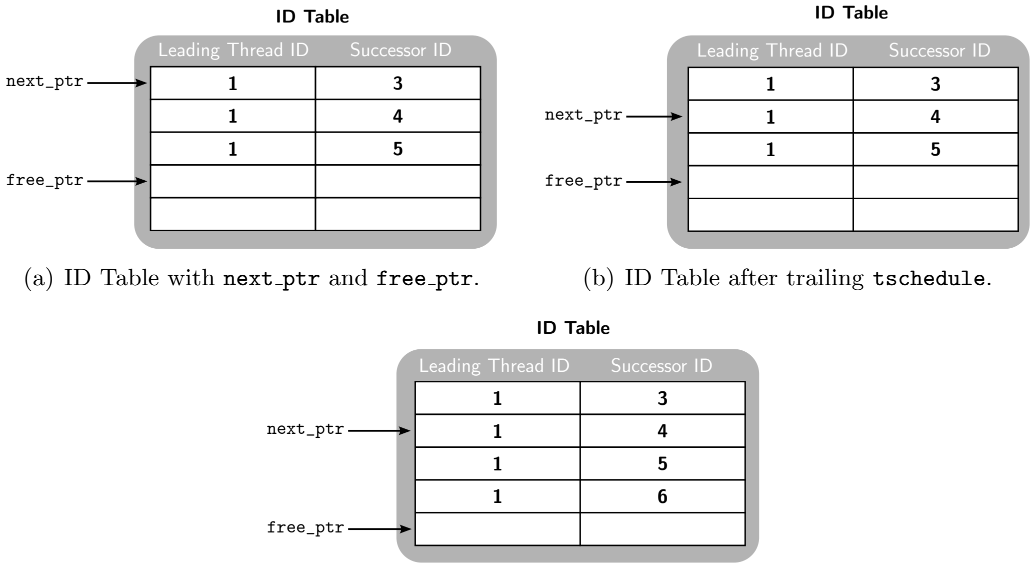
- *red_cont* (Redundant Continuation Pointer): stores the thread ID of the redundant continuation.
- *spec* (Speculative Thread): indicates, whether the thread is speculative.
- *parent_id* (Parent ID): stores the thread ID of the parent thread. The parent ID is required to discard the speculative continuations, if the parent thread needs to be restarted.
- *tsched_count* (`tschedule` Count): stores the number of issued `tschedule` instructions of the thread.
- *pe_id* (PE ID): stores the ID of the PE, which has executed the thread.
- *free_ptr* (ID Table Free Pointer): points to the next free entry in the *ID Table*.
- *next_ptr* (ID Table Next ID Pointer): points to the next thread ID in the *ID Table* to be returned on a redundant `tschedule`.
- *re-start* (Restarted Flag): indicates whether the thread has been restarted.

Please note that the synchronisation count of the redundant thread is always zero, since the redundant continuation is only created, when the synchronisation count of the original thread became zero.

5.5.2 ID Table

As described in Section 5.3.3, double execution requires deterministic thread IDs returned on redundant `tschedules` to guarantee consistent input replication. The TSU must therefore temporarily store the created thread IDs and their sequential order. This is supported by the ID table, which allows storing and retrieving thread IDs in case of redundant `tschedule` instructions. The TSU keeps all IDs created on `tschedule` in an additional memory structure, the *ID table*. The ID table stores the thread ID of the trailing thread together with the newly created thread ID, when a `tschedule` has been executed for the first time.

To support fast access, a thread's continuation maintains two additional pointers, one which stores the index of the next thread ID (*next_ptr*) to be returned and another one, which points to the next free entry in the ID Table (*free_ptr*) (see Figure 5.8(a)). After the TSU has returned a thread ID stored in the ID table, it tries to find the next ID to be returned by traversing the list until the next ID of the leading thread is found (see Figure 5.8(b)).



(a) ID Table with `next_ptr` and `free_ptr`.

(b) ID Table after trailing `tschedule`.

(c) ID Table after leading `tschedule`.

Figure 5.8: ID table for redundant `tschedule`.

When a new entry is entered, the TSU uses the *free_ptr* in the continuation to store the leading thread's ID together with the new ID. Afterwards, the free pointer is moved to the next empty entry in the list (see Figure 5.8(c)). To support the fault localisation mechanism proposed in Section 5.6, the entries in the ID Table are only removed in case of a fault-free execution.

5.6 Fault Diagnosis and Periodic Tests

Fault diagnosis and adaption are necessary to cope with permanent and intermittent faults. For instance, a permanent fault in a PE may lead to continuing recovery actions, bringing the complete system to a halt [Sorin, 2009]. To prevent such situations, the FDU identifies permanent or intermittent faulty PEs and instructs the TSU to deactivate the faulty PE.

The technique is based on the fault detection and recovery mechanisms proposed in the Sections 5.3 and 6.1. Here, the FDU works as a fault diagnosis unit, which tracks detected errors per PE and determines whether a PE is transient, intermittent, or permanent faulty.

The FDU can also be used to track thermal condition or detected errors by other fault detection mechanisms, like the machine check architecture (MCA), however, advanced proactive mechanisms based on these information are out of scope for this thesis. For further ideas on proactive fault management by the FDU please refer to Weis et al. [2011b].

5.6.1 Fault Diagnosis

The FDU stores all received signatures of the threads to localise the faulty component. This is possible since the signatures of the restarted threads are always equal in the absence of errors. When the FDU receives the signatures of the thread re-executions, the PE, which suffered from the fault can be identified by a majority voting. Under the assumption that faults in different PEs never produce the same signature, the FDU can identify the erroneous thread execution and assume that the PE, which produced an error, has suffered from a fault. However, the scheduler must ensure that in the case of a re-execution at least one thread uses a different PE. Furthermore, re-executed threads always use the ID Table to retrieve the same thread IDs as the execution before. After having identified the correct signature, we can consider all diverging signatures as erroneous.

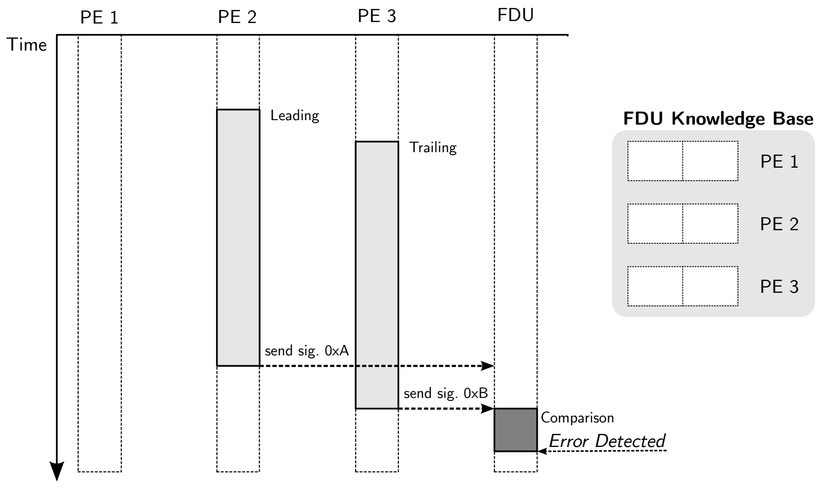
During execution, the FDU continuously tracks all identified faulty PEs and stores them in the knowledge base. The knowledge base uses a list per PE to keep the localised faults as binary values. A bit value indicates, whether a fault has been localised for this PE. The FDU also distinguishes between transient and permanent faulty PEs. In order to do so, it looks in its knowledge base, whether the last execution localised a fault on this PE, too. In this case, the PE is considered as permanent faulty. When a PE is identified to be permanent faulty, the FDU informs the TSU to deactivate the PE.

Example. Figure 5.9 shows how the permanent faulty PE 3 can be localised. First, the redundant threads are executed on PE 2 and PE 3. On result comparison, an error is detected. However, the FDU does not know, which PE produced the error, stores the signatures produced by the threads and restarts them. The redundant thread pair is re-started on PE 1 and PE 3. Since PE 3 is still permanent faulty, the FDU detects an error again. The FDU now uses majority voting to detect that PE 1 and PE 2 have produced the same signature. After result comparison, the FDU updates its knowledge base about PE 1 and PE 3. Since PE 3 suffered from two errors in a row, the FDU considers PE 3 as permanent faulty. ♦

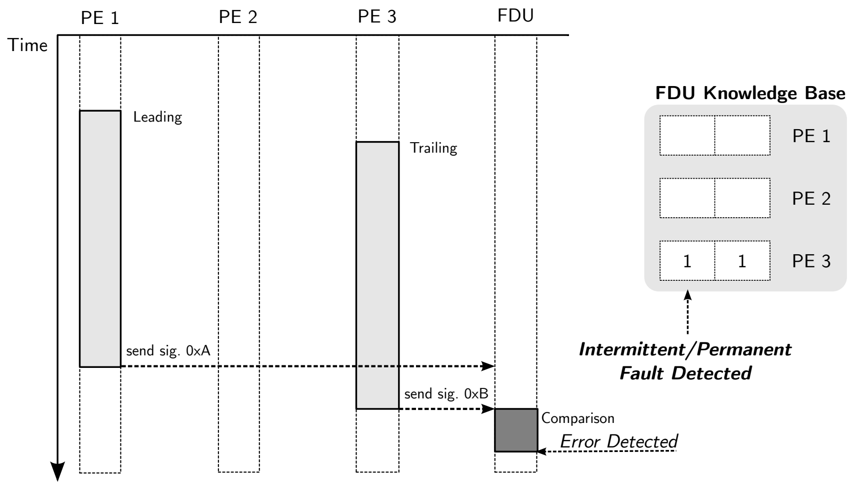
5.6.2 Periodic Testing

The system uses periodic tests to check, whether a PE is still faulty. Generally, the FDU does never consider a PE as permanent faulty, but only assumes intermittent faults. After a given number of n committed threads, the FDU re-tests a deactivated PE. Therefore, it resets the state of the PE from faulty to operational. When the next executed thread is faulty again, the test is repeated after n committed threads.

The test interval depends on the recovery overhead of the selected recovery mechanism. This means that periodic tests are cheaper for double execution with thread restart recovery than optimistic double execution with global checkpointing (see Chapter 6).



(a) Double execution on PE 2 and PE 3, while PE 3 is permanent faulty. The FDU detected a error and keeps the signatures produced by double execution. The TSU re-starts the threads, while at least one thread is executed on a different PE than before.



(b) Double execution on PE 1 and PE 3, while PE 3 is permanent faulty. After the erroneous execution was detected, the FDU compares the signatures of all thread executions. In this case, we can see that PE 1 and PE 2 have produced the same signature, which means that PE 3 has produced the erroneous signature. Since PE 3 has produced an erroneous signature two times in a row, we store that information in the knowledge base.

Figure 5.9: Fault diagnosis using double execution.

5.7 Summary

This chapter proposed data-flow based fault-tolerance mechanisms to cope with transient, intermittent, and permanent faults in the processing elements of a multi-core architecture. Fault detection is implemented by double execution, a loosely-coupled redundant execution scheme for data-flow threads. Double execution exploits the data-flow execution principle of the baseline architecture for input replication, redundant thread management, and output comparison. Based on double execution, a data-flow thread restart recovery mechanism was proposed. The required enhancements for double execution and thread restart recovery to the data-flow execution model and the hardware architecture were discussed in detail. Additionally, a fault diagnosis mechanism was proposed, which uses double execution and the thread restart recovery for fault localisation.

The data-flow based fault-tolerance mechanisms proposed in this chapter provide the following benefits:

1. Double execution uses data-flow threads for redundant execution. This allows leveraging the data-flow runtime to execute redundant data-flow threads. In particular, the data-flow scheduler can dynamically schedule redundant threads on PE availability.
2. Result comparison is only necessary for data which is consumed by subsequent threads. This reduces the amount of data, which must be compared.
3. Redundant data-flow threads can be executed completely decoupled from each other, supporting a flexible utilisation of the baseline multi-core architecture.
4. Result propagation is only required when a thread has finished execution, which simplifies input replication, output comparison, and enables thread restart recovery.
5. Double execution and thread restart recovery can be used for the localisation and the diagnosis of permanent and intermittent faulty PEs by the FDU.
6. The integration of double execution and thread restart recovery in the data-flow execution model supports the flexible deactivation of permanent faulty PEs. The asynchronous redundant thread execution can also exploit uneven numbers of PEs.

6 Optimistic Double Execution and Global Checkpointing

This chapter presents in Section 6.1 *Optimistic Double Execution* (ODE), a speculative variant of double execution, which can increase the parallelism of double execution. Since ODE can not recover from errors by the thread restart recovery mechanism proposed in Chapter 5, Section 6.2 proposes a data-flow based global checkpoint mechanism. Finally, Section 6.3 gives a summary on the fault-tolerance mechanisms of this chapter.

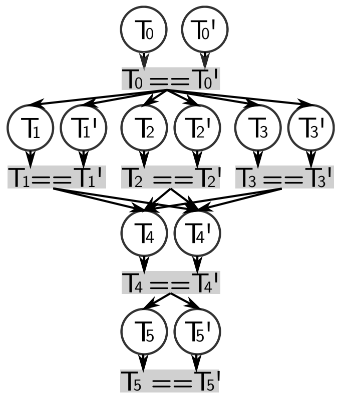
6.1 Optimistic Double Execution

This section gives an overview of ODE. First, we describe how ODE can increase the parallelism of double execution of data-flow threads. Second, we present the execution behaviour of ODE. Third, we discuss input replication and output comparison for ODE and present a mechanism for the input replication of the OWM sections.

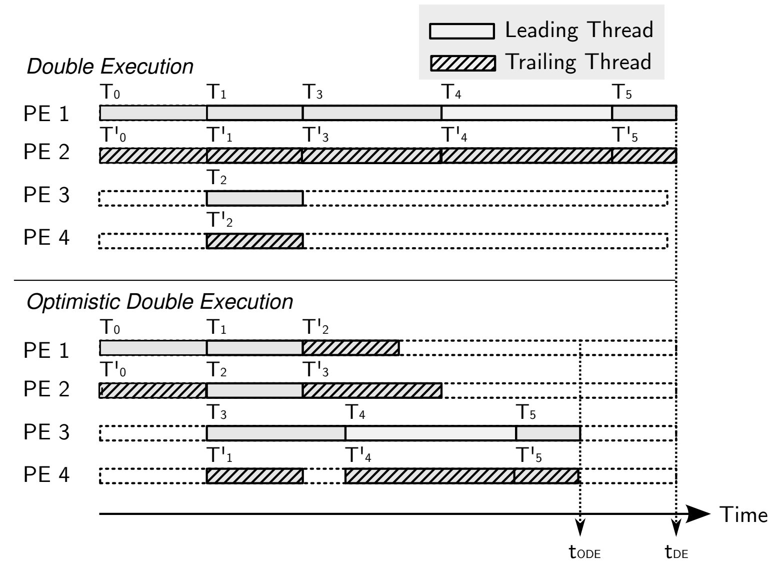
6.1.1 Increasing Parallelism by Optimistic Thread Commit

The parallelism of double execution is restricted by the parallelism of the data-flow application, since redundant threads must always be compared before the commit of the trailing thread and before subsequent threads can be started. As a consequence, the scalability of double execution is limited by the parallelism of the data-flow application and can not be significantly higher than the execution on a non-fault tolerant system with half of the cores.

To overcome this restriction of double execution, ODE speculates that the redundant thread execution is always correct and allows forward progress by committing the leading thread before the trailing thread has finished execution or is even started. This enables to optimistically start subsequent threads before the trailing thread has verified the execution and therefore increases the parallelism. Optimistic double execution enables a complete decoupling of the leading and the trailing threads, since the system can make



(a) Dynamic data-flow graph of double execution.



(b) Thread execution of double execution and optimistic double execution.

Figure 6.1: Example for increasing parallelism by the optimistic thread commit of ODE.

forward progress without waiting for the signature comparison by the trailing threads and can defer the verification by the trailing threads to a later point in time.

As mentioned before, the optimistic commit of the leading thread can increase the parallelism, since the parallelism of double execution is restricted by the dependencies between the data-flow threads. However, in case of ODE, the data dependencies between the threads restrict only the execution of the leading threads, but not the execution of the trailing threads, because the trailing threads' dependencies have already been resolved when the threads were duplicated.

Example. Figure 6.1 shows by example the increasing parallelism in case of ODE. The dynamic data-flow graph of double execution, depicted in Figure 6.1(a), is executed by double execution and ODE on a system with 4 PEs in Figure 6.1(b). In the example execution for double execution in Figure 6.1(b) it can be seen that the data dependencies restrict that the application can fully utilise the system, since T_4 and T'_4 can only be started when $T_1...T_3$ and $T'_1...T'_3$ have finished execution. By contrast, ODE fully utilises the system to finish the threads $T_1...T_3$. Since ODE commits threads before the trailing thread has verified the computation, T_4 and T'_4 become ready for execution sooner and can run in parallel to the trailing threads T'_2 and T'_3. It can be further seen that the trailing threads T'_2, T'_3, and T'_4 can be executed in parallel, which increases the parallelism of the execution of the trailing threads and reduces the execution time of the data-flow application, since $t_{ODE} < t_{DE}$. ♦

6.1.2 Run time Behaviour of Optimistic Double Execution

The run time behaviour of ODE can be divided into the following stages:

❶ **Thread start:** Similar to double execution, the leading and trailing thread are started when a PE becomes available.

❷ **Thread execution:** When the threads are executed, the leading thread's PE buffers all `twrite`/ `twritep`/`tdecrease` instructions in the write buffer and the OWM buffer. Simultaneously, the core probe creates the CRC-32 signature, similar to double execution. The core probe of the trailing thread's core creates also the signature of its output.

❸ **Thread end:** When the leading thread has finished execution, indicated by `tdestroy`, the PE's core probe sends the CRC-32 signature to the FDU. Afterwards, the leading thread immediately commits without waiting for the trailing thread and the synchronisation counts of the succeeding threads are decremented instantaneous. The TSU can then immediately start new threads, when their synchronisation counts has reached zero.

❹ **Output comparison:** The FDU waits for the signatures of both the leading and the trailing threads and compares them.

❺ **Thread commit or recovery:** In case of matching signatures, the TSU can continue execution. When a fault is detected, the FDU triggers the global recovery mechanism described in 6.2.

By contrast to double execution, ODE always schedules the trailing threads with lower priority than the leading threads, because only the leading threads can commit its (unverified) results and guarantee global progress of the system.

6.1.3 Input Replication

Similar to double execution, the input data must be replicated for the leading and trailing threads. Since the TF of a data-flow thread is immutable and can not be modified after the synchronisation count has reached zero, explicit duplication of the TF is not necessary, even when the leading thread commits.

In case of double execution, stores to the OWM region do not require explicit input replication, because concurrent OWM stores to the same memory location are prevented by data dependencies in the data-flow graph. This means that parallel executable data-flow threads can not write the same OWM addresses, while writes to the OWM region of redundant threads are isolated from each others in the OWM buffer. However, ODE allows the leading thread to write (unverified and possible erroneous) output to the

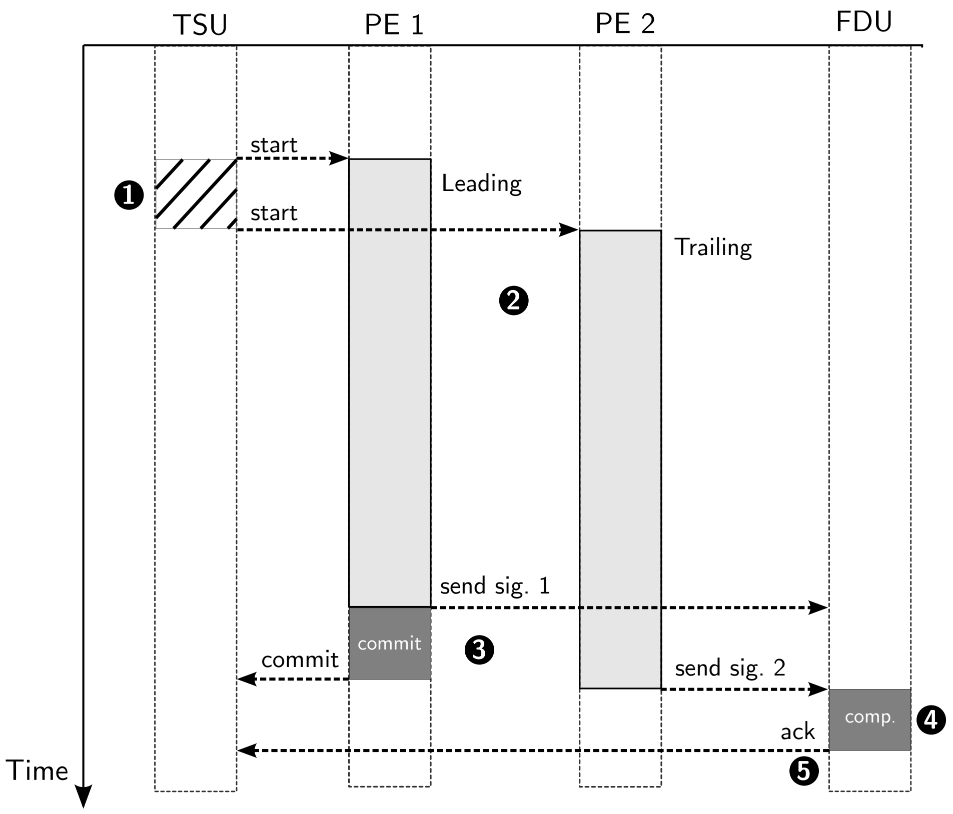


Figure 6.2: Execution stages of ODE.

OWM region. Furthermore, overwritten values in the OWM may be later required by the trailing thread. This can lead to input incoherence between trailing and leading threads. To prevent input incoherence for OWM reads of redundant threads, the accessed parts of the OWM must be replicated before they can be changed by the leading thread's commit. Therefore, the OWM section accessed by a thread must be copied before the leading thread can commit. To restrict the size of the copied section to a minimum, an additional instruction is used (`owm_mem`), which is executed at the beginning of the thread to inform the TSU about the start address and the size of the OWM section. The `owm_mem` instruction can be added by the compiler or the programmer and must be executed in both redundant threads. The TSU then initiates a replication of this section in the OWM region to prevent input incoherence between the trailing and the leading threads.

The management of the mapping between the original OWM section and the copied section is managed by the TSU by an OWM mapping table, depicted in Figure 6.3. The mapping table stores the ID of the trailing thread, the pointer to the original section, the pointer to the copied section, and the length of the section.

When the `owm_mem` is executed the first time, the TSU searches in the OWM mapping table, whether an entry for the trailing thread ID exists. If this is not the case, the TSU creates a copy of the section specified by the `owm_mem` instruction using a DMA transfer and allocates a new entry in the OWM mapping table. When the trailing thread calls the same `owm_mem` instruction, the TSU searches again in the OWM mapping table for the

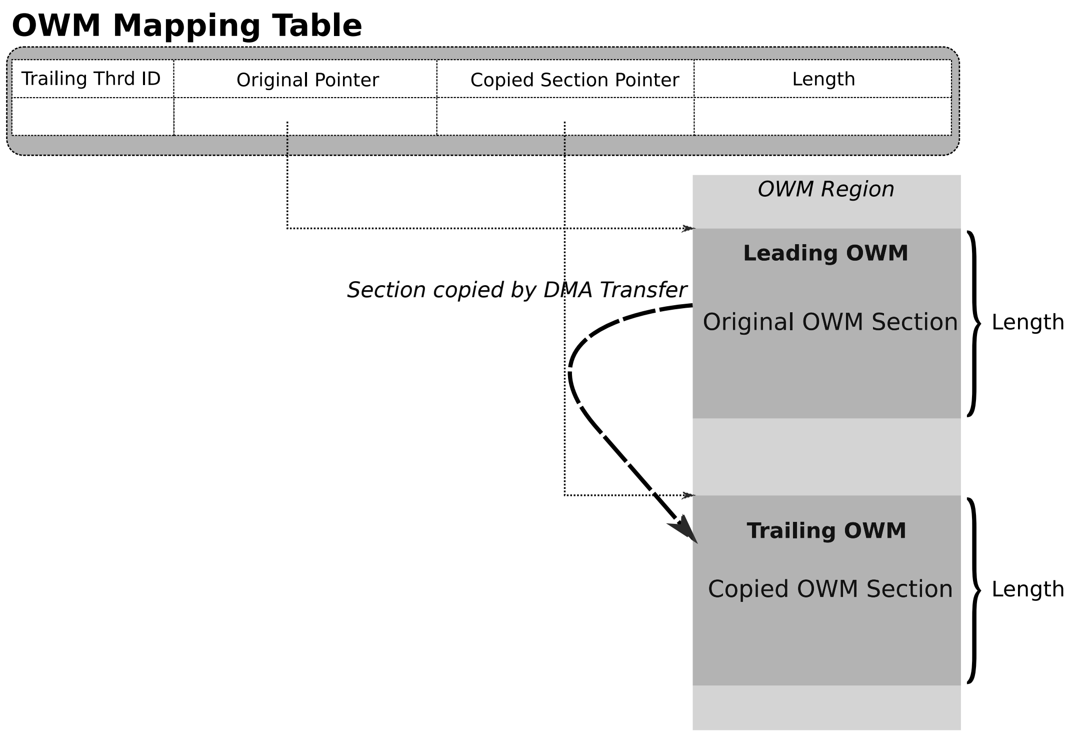


Figure 6.3: OWM mapping table and pointers to the original and the copied OWM sections.

entry of the corresponding thread. The corresponding entry of the OWM mapping table is then forwarded to the core probe. The core probe takes now care for the mapping and prevents the PE from writing to the original OWM region.

The core probe monitors all OWM read addresses of a PE and translates them to the appropriate physical addresses of the leading or trailing thread. As a consequence, the leading and trailing threads have different physical OWM addresses in the system. This prevents the leading thread from serving as a prefetcher for the trailing thread, since redundant threads read OWM input from different physical addresses.

Reducing the OWM Copy Overhead The copying of the trailing thread's shared memory section can induce execution overhead, since the OWM section specified in the `owm_mem` instruction must be copied before the leading thread can commit. To prevent stalling of the trailing thread's start, the OWM region is started immediately when the redundant threads become ready for execution and are moved to the RQ.

6.1.4 Output Comparison

Output comparison for optimistic double execution is similar to double execution, which means that writes the OWM region, `twrites`, `twriteps`, `tdecrease` instructions, and `tschedules` must be incorporated in the CRC-32 signature. For the write accesses to the OWM region, the original OWM addresses must be used for the trailing thread's signature.

6.2 Data-flow based Global Checkpointing

The optimistic thread commit of ODE before result verification allows that erroneous results have already been committed to the main memory and the TSU and may be consumed by subsequent threads before optimistic double execution can detect an error. This means that possible errors can no longer be isolated within the PEs and the restart of threads can not be used for recovery. As a consequence, in case of ODE, the TFs and the OWM region may contain errors and must be recovered, when an error is detected. Since errors can be at any address in the memory after detection by ODE, the recovery mechanism must recover the complete main memory and the TSU state. This section proposes a data-flow based global checkpointing mechanism, which supports a global rollback of the system, including the state of the TSU and the main memory. The proposed checkpointing mechanism exploits the data-flow execution model to reduce the checkpoint creation overhead.

6.2.1 Establishing a Checkpoint

Figure 6.4 shows the required steps to establish a global checkpoint with ODE:

❶ To establish a global checkpoint, the FDU must ensure that the checkpoint is safe, i.e. error free. This means that all committed, but still unverified threads must be checked until the new checkpoint can be established. A new checkpoint can be only established, when all committed leading threads have been compared with their trailing thread counterparts. The TSU scheduler changes the scheduling policy when a checkpoint creation is triggered, prioritising the outstanding trailing threads over the leading threads. The checkpoint interval can be therefore not exactly determined, since it depends on the number of unverified leading threads in the RQ.

❷ After all outstanding trailing threads have committed and their results have been compared, a checkpoint can be established. The state of the system at this point is called the *recovery point* [Sorin, 2009], which is recovered in the case of a rollback. To create the checkpoint, the TSU determines the start and end addresses of the current frame memory region and the OWM region in the main memory. Furthermore, the TSU creates a backup of its current context, including all internal tables and stores it in the main memory, too. The global state of the system consists of the TSU state, the thread frames and the OWM region, which is accessible by all threads.

❸ After the checkpoint has been established, all subsequent `twrite`/`twritep`s going to the checkpoint's memory region will be logged. This means the TSU maintains a log of all changes to the TFs within the checkpoint's memory region. Newly allocated TFs must not be recovered and are created outside of the checkpoint's log region. Similar to the TF, a log over the complete OWM region is established, too.

❹ When ODE detects a fault, the TSU recovers to the recovery point by restoring the frame memory log, the OWM log, and the stored TSU context.

Compared to prior global checkpoint mechanisms for shared-memory multi-cores, like [Prvulovic et al., 2002; Sorin et al., 2002], data-flow based global checkpointing only needs to keep a log of the frame memory and OWM regions, instead of maintaining a log of the complete main memory. Furthermore, a global checkpoint in a shared memory von Neumann architecture must always incorporate the architectural context of all cores in the system [Pradhan, 1996, pp. 160 ff.]. Additionally, data-flow based checkpointing does not explicitly checkpoint the state of the cache coherence protocol or flush the caches in order to establish a new checkpoint.

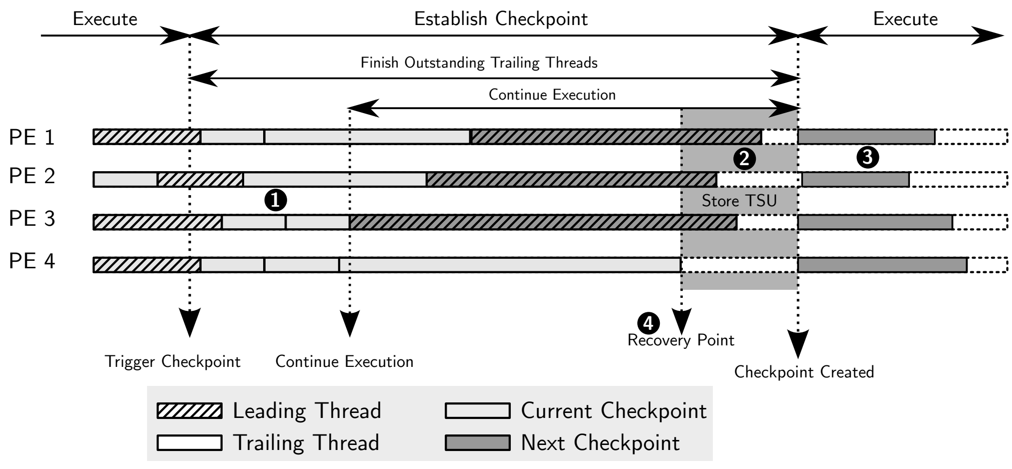


Figure 6.4: Optimistic double execution with reduced waiting time.

6.2.2 Logging of Memory Accesses

After a checkpoint has been established, the system must record all changes to the logged memory regions. Since the TSU state has been copied on checkpoint creation, only the accesses to the frame memory and OWM region must be logged. We propose optimistic logging for the thread frame and the OWM region. Optimistic means that old values are copied to a separate log region before the value can be written to memory. Since the globally visible state is only changed when a thread commits, for ODE only results of the leading thread must be logged.

6.2.3 Checkpointing Overhead

The checkpoint overhead, i.e. the additional time required to establish a new checkpoint, is influenced by two factors.

First, the system must finish all currently unverified threads. This may result in lower utilisation of the system, however, the effect can be reduced by overlapping the checkpoint creation with the data-flow execution of the next checkpoint interval, as described in Section 6.2.4. Second, the TSU must be stopped until the TSU context is stored in the main memory. However, this affects only threads, which execute a T*-instruction at the moment the TSU context is written to the main memory.

After a checkpoint has been established, all writes to the checkpoints's memory region must be logged, which means that data must only be logged, when the leading thread commits,

6.2.4 Overlapping of Checkpoint Verification and Data-Flow Execution

A new checkpoint can only be established when all unverified threads have been compared. However, this may lead to reduced utilisation of the system. Fortunately, data-flow threads of the next checkpoint interval can be started and executed even when the checkpoint creation is not yet finished. Starting new trailing threads is allowed when a PE becomes available, but no outstanding trailing threads of this checkpoint can be scheduled. Then the TSU starts a leading thread of the next checkpoint interval (see *Continue Execution* in Figure 6.4). The functional semantic of the data-flow execution supports this, since all side effects of the newly started threads are only visible in the system, when the thread is committed. This means that the started leading thread of the next checkpoint interval can not commit until the checkpoint is established. After the new checkpoint has been established, the waiting threads of the next checkpoint can be committed. This overlapping of checkpoint verification and data-flow execution of the next checkpoint interval reduces the checkpointing overhead, since idle times in case of checkpoint creation is exploited for useful data-flow thread execution.

6.2.5 Fault Diagnosis and Adaption for Optimistic Double Execution and Global Checkpointing

A global rollback of the system has disadvantages for the fault diagnosis mechanism described in Chapter 5, since on recovery, the faulty threads can not be re-executed, when they no longer exist in the system after a rollback. Therefore, before a global checkpoint is restored, the system tries to localise the faulty PE by triggering a thread restart. However, the leading thread can not be restarted, since the OWM section of the leading thread may have already been changed by the commit of leading thread. Therefore, only the trailing thread is restarted on a different PE. After the trailing thread has finished, the FDU uses majority voting to identify the PE of the faulty execution. Afterwards, the system can be recovered.

6.3 Summary

This chapter proposed optimistic double execution, which can increase the parallelism of double execution. Compared to double execution, ODE is able increase the parallelism, since unverified threads are allowed to immediately commit and spawn succeeding threads. Furthermore, result comparison can be deferred to a later point in time, increasing the flexibility of the PE usage. Additionally, the comparison latency is moved from the critical path, because the immediate commit of the leading thread completely eliminates the PE blocking issue.

The immediate commit of the leading thread can increase the start slack between the leading and the trailing threads, since the TSU scheduler prioritises leading threads over trailing threads. As a consequence, the commit slack between the leading and the trailing threads can be significantly higher compared to double execution.

The duplication of the accessed OWM sections and the OWM mapping mechanism may induce additional overhead and bus traffic. Furthermore, the OWM mapping may increase the cache occupancy and reduce the cache hit rate for OWM reads, because OWM read accesses from leading and trailing threads are mapped to different physical addresses.

Due to the speculative commit of unverified thread output, thread restart recovery can not be used with optimistic double execution. Therefore, a data-flow based global checkpointing mechanism was proposed. The global checkpointing mechanism uses the data-flow execution principle to reduce the checkpoint creation overhead by overlapping the checkpoint creation time with data-flow thread execution and the logging overhead, by only logging changes to the OWM region and a part of the thread frame region.

7 Evaluation

This chapter presents evaluation results of the proposed data-flow based fault-tolerance mechanisms. The section is structured as follows. Section 7.1 gives an overview of the simulation methodology, including a description of the simulator framework, the configuration of the simulator, and a characterisation of the data-flow benchmarks. Section 7.2 presents evaluation results on double execution and thread restart recovery. Section 7.3 presents results on optimistic double execution and data-flow based global checkpointing. Finally, Section 7.4 summarises the evaluation results.

7.1 Simulation Methodology

7.1.1 Simulator Framework

We used the open source full-system multi-core simulator COTSon [Argollo et al., 2009] for the evaluation of the proposed fault-tolerance mechanisms. The COTSon simulator was extended in the Teraflux project [Portero et al., 2012] with support for T*-based data-flow execution. For this evaluation, the proposed fault-tolerance mechanisms were integrated into COTSon, which enabled us to simulate double execution, thread restart recovery, optimistic double execution, and data-flow based global checkpointing.

The COTSon simulator is a functional-first simulator, which simulates a group of instructions within a time quantum, calculates the timing for this quantum, and adjusts the timing in the processor for the next simulation quantum. The length of the time quantum can be adjusted to trade simulation accuracy against simulation speed. For all simulations in this thesis, a fixed quantum length of 500 ns is used.

7.1.2 Baseline Machine Configuration

Table 7.1 depicts the overview of the baseline machine configuration for this evaluation. The baseline machine models a contemporary x86_64 multi-core processor with 1, 2, 4, 8, 16, or 32 PEs. The timing of a PE is calculated by an enhanced version of the `timer1` provided by the COTSon framework. The `timer1` was extended with timing support for T*-instructions and also write and OWM buffer accesses. Each PE consists of an out-of-order pipeline with 5 stages and a maximal fetch and commit width of 3 instructions per cycle. The pipeline has a reorder buffer for 192 instructions. The private cache hierarchy of each processing element consists of separate 32kB L1 instruction- and data caches, which are connected to the main memory by a memory bus with a bus latency of 25 cycles. The memory bus uses the MOESI cache coherence protocol with support for cache-to-cache transfers.

Table 7.1: Baseline system configuration.

Parameters	Values
Pipeline	Out-of-order, Pipeline length: 5, Fetch width: 3, Commit width: 3
L1 I- and D-Cache (private per core)	Size: 32kB, Line size: 64, 8-way set associative, Hit Latency: 1 cycle, writeback
Cache Coherence Protocol	MOESI
Memory Bus Latency (L1 to memory)	25 cycles
Memory Latency	100 cycles
Clock Rate	1 Ghz

Table 7.2 shows the additional latencies in the pipeline for the simulated T*-instructions. For `tdestroy` and `tschedule` instructions, we assume a fixed latency of 40 cycles. `tread` operations are simulated as normal load/store operations. We further assume that writing to the write buffers (write buffer and OWM buffer) and generating a CRC-32 signature causes no additional latencies in the pipeline per `twrite`/`twritep` or OWM store instruction. The same is assumed for `tdecrease` instructions. For already buffered `tread`s and reads from the OWM buffer, we assume a latency of 1 cycle, similar to a cache hit in the L1 data cache.

Table 7.2: Additional static T*-instruction latencies.

Parameters	Latency
`tread` (write buffer hit)	1 cycle
`tschedule`	40 cycles
`tdestroy`	40 cycles

7.1.3 Lockstep Machine Configurations

For the comparison of the execution overhead of double execution and optimistic double execution, we simulate two additional tightly-coupled lockstep configurations, which are based on the baseline machine configuration:

1. An ideal tightly-coupled lockstep machine (*Lockstep*), which executes all data-flow threads in cycle-by-cycle lockstepping on a redundant PE. It is assumed that the ideal lockstep machine can execute and compare instructions before instruction commit with zero overhead, i.e. the architectural registers of the PEs represent

an error-free execution state, which can be used for recovery of the PE. When the lockstep machine detects an error, the pipeline is flushed and the execution is restarted at the last committed instruction with no overhead. For the cycle-by-cycle lockstep technique, we assume that synchronisation and comparison between the redundant processing elements induces no additional costs. The *Lockstep* system was simulated by executing the benchmarks on half of the cores without double execution. *Lockstep* is used for comparison of the data-flow based redundant execution mechanisms with an ideal fault-tolerant system.

2. A tightly-coupled lockstep machine (*LockstepCheck*), which implements a global checkpointing mechanism on top of the ideal *Lockstep* system. *LockstepCheck* is used to compare the data-flow based checkpointing proposed in Chapter 6.2 with a conventional global checkpointing implementation. In *LockstepCheck* a global checkpoint consists of the memory state and the architectural registers of all PEs. A checkpoint is established as follows: The execution on all cores is stopped, the architectural registers of the lockstepped PEs are compared, stored at a safe place, and the caches are flushed [Prvulovic et al., 2002]. For simulation purpose, we assume that comparing the register files, flushing the caches, and stopping the PEs takes a fixed time. The checkpoint creation overhead is here on the critical path, since all PEs must be stopped and the system can not make forward progress until the checkpoint is established. Since the global system is stalled during checkpoint creation, we model the overhead for checkpoint creation by adding a constant factor for the creation of each checkpoint to the execution time of *Lockstep*. We assume a fixed checkpoint creation overhead of 0.1 µs for all simulations of *LockstepCheck*.

7.1.4 Fault Injection

The fault detection behaviour under the presence of faults is evaluated by a high-level fault injection mechanism, which simulates the occurrence of transient and permanent faults at run time. The purpose of the fault injection simulation is not to investigate the fault coverage of the proposed mechanisms, but to investigate the run time behaviour of the fault detection and recovery mechanisms under the presence of faults.

Therefore, the implemented fault injection mechanism does not change the functional execution of the simulator, but the signature generation on the processing elements to flag the occurrence of faults at certain points in time. After the signature of a data-flow thread has been manipulated on a PE, the system detects the fault by comparing the signatures and activates one of the recovery mechanism, when required. In the following, we will describe how transient and permanent faults are modelled in the simulator.

Permanent Faults Permanent faults can be simulated by completely deactivating the faulty PEs and never executing threads on them. This enables us to evaluate whether double execution is able to support graceful degradation, even when multiple PEs are permanent faulty.

Transient Faults Transient faults are injected at a fixed time interval. After the time interval elapsed, a PE is selected by a round-robin selection policy and the next thread's signature, which tries to commit on this PE, is manipulated.

Table 7.3: Benchmark overview.

Benchmark	OWM Required	OpenStream Required
Fibonacci	no	yes
Matmul	no	no
Sparse LU	yes	yes
Cholesky	yes	yes

7.1.5 Benchmarks

For the evaluation of the proposed techniques, the following benchmarks are used:

Fibonacci number (Fibonacci) calculates recursively the Fibonacci number. Two data-flow threads are scheduled on each recursive step, until a cut-off condition is reached. When the cut-off condition is reached, the rest is calculated sequentially. Fibonacci does not require OWM sections. This means that in Fibonacci communication between threads is implemented by `tread` and `twrite/twritep` instructions.

Cholesky Factorisation (Cholesky) computes the Cholesky factorisation of a matrix. The algorithm works in-place on different blocks of the input matrix and uses the OWM region for in-place updates.

Block-wise Matrix Multiplication (Matmul) is a standard block-wise matrix multiplication. The thread-level parallelism depends on the number of blocks per matrix. Matmul was ported by hand from a shared memory implementation and does only use the TFs and `tread/twrite` instructions for communication between threads. Matmul does not require the OWM region.

Sparse LU Decomposition (Sparse) calculates the LU decomposition of a sparse matrix. The algorithm works in-place using the OWM region.

Fibonacci, Cholesky, and Sparse LU were compiled with a variant of the OpenStream compiler[1], which provides a configuration with support for the T*-instruction set extensions [Li et al., 2012]. Table 7.3 gives an overview of the different benchmark requirements.

Input Sets

Table 7.4 shows the input sets for the benchmarks, which differently utilise the simulated system configurations.

- *HighUtil* provides high utilisation of the simulated system configurations. This enabled us to investigate the behaviour of the fault-tolerance mechanisms for applications with high parallelism.
- *LowUtil* provides lower utilisation of the simulated system configurations. The *LowUtil* input set can not fully utilities all PEs for all system configurations, which allows the exploitation of underutilised PEs for optimistic double execution.

[1]`http://openstream.info/`, Retrieved on September 30, 2016

Table 7.4: The input parameters of the *HighUtil* and *LowUtil* input sets.

Benchmark	*HighUtil* Input Set
Fibonacci	n: 36, cut-off: 20
Matmul	Blocks: 12×12, Block Size: 16×16
Sparse LU	Matrix Size: 512×512, Block Size: 16×16
Cholesky	Matrix Size: 512×512, Block Size: 16×16
Benchmark	***LowUtil* Input Set**
Fibonacci	n: 35, cut-off: 28
Matmul	Blocks: 6×6, Block Size: 16×16
Sparse LU	Matrix Size: 256×256, Block Size: 16×16
Cholesky	Matrix Size: 256×256, Block Size: 16×16

Benchmark Characteristics

Table 7.5 shows the characteristics of the benchmarks for both the *HighUtil* and *LowUtil* input sets. For the *HighUtil* input set, the benchmarks execute between 7,536 (Cholesky) and 1,728 (Matmul) data-flow threads and for the *LowUtil* input set between 68 (Fibonacci) and 1,208 (Cholesky) threads. As mentioned before, Fibonacci and Matmul do not require the OWM region. By contrast, Sparse LU and Cholesky make in-place updates on their input matrices and require the OWM region. The table shows that both benchmarks execute more read/write OWM accesses than `twrite`/`tread` instructions. Matmul uses only `tread`/`twrite` instructions, while Fibonacci executes a relatively small amount of T*-instructions compared to the other benchmarks.

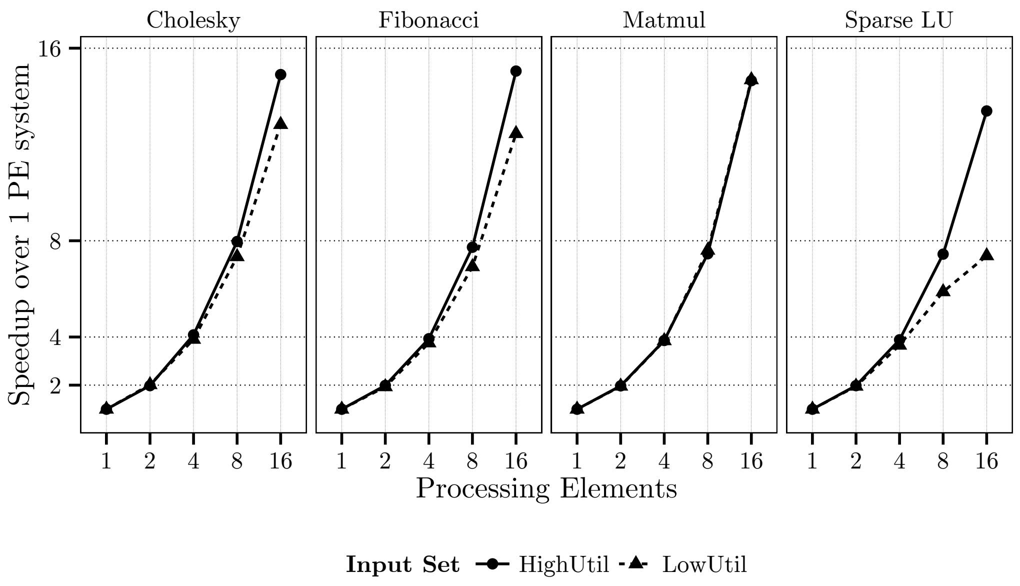


Figure 7.1: Speedup of the simulated benchmarks normalised to the execution time on a system with 1 PE.

Table 7.5: Data-flow benchmark characteristics.

	HighUtil			
	Fibonacci	Matmul	Sparse	Cholesky
# Thrds	5,168	1,728	4,005	7,536
# `tread`	56,841	1,369,153	90,499	200,505
# `twrite`	5,167	1,369,154	7,915	17,920
# `twriteP`	41,336	0	69,789	150,928
# `tdecrease`	25,835	0	57,235	133,248
# OWM reads	0	0	27,013,515	35,432,235
# OWM writes	0	0	9,052,526	11,699,867
	LowUtil			
	Fibonacci	Matmul	Sparse	Cholesky
# Thrds	68	217	701	1,208
# `tread`	741	175,753	12,772	27,645
# `twrite`	67	175,754	1,083	2,432
# `twriteP`	536	0	9,679	20,680
# `tdecrease`	335	0	7435	17,600
# OWM reads	0	0	3,108,481	4,507,306
# OWM writes	0	0	1,047,125	1,530,061

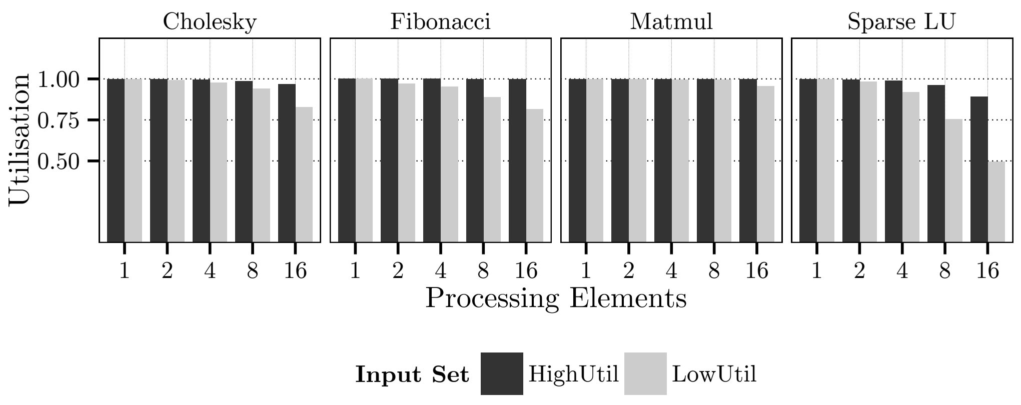


Figure 7.2: Utilisation of the systems with the *HighUtil* and *LowUtil* input sets.

Figure 7.1 presents the speedup of the non-redundant data-flow execution for the *HighUtil* and the *LowUtil* input sets normalised to the execution time on a system configuration with 1 PE. As expected, the *HighUtil* input set shows a higher scalability than the *LowUtil* input set. Particularly configurations with more PEs (8 and 16 PE systems) reveal that the scalability of the *LowUtil* input set significantly decreases with more available PEs in the system. Sparse LU saturates at 16 PEs, which means that the benchmark can effectively not exploit more than 16 PEs with this input set. The decreasing scalability for the *HighUtil* input set in general stems from the reduced sequential performance induced by the inherent overhead for parallel thread execution in the baseline architecture.

Figure 7.2 shows the utilisation for both input sets. The utilisation of the system is the average fraction of time at which the PEs execute data-flow threads. For the *HighUtil* input set, we can see high utilisation for all benchmarks and system configurations with up to 16 PEs. However, for Sparse LU and Cholesky, the utilisation decreases significantly on the 16 PE systems. For the *LowUtil* input set, it can be seen that the utilisation is significantly lower for all benchmarks. In particular, Sparse LU provides only an utilisation of $\approx 50\%$ for the 16 PE system.

7.2 Double Execution

This section presents results on the execution behaviour of double execution and thread restart recovery. First, the fault-free execution overhead and scalability of double execution compared to the ideal *Lockstep* system is presented. Second, the impact of the comparison latency on the execution time of double execution is discussed. Third, the average commit slack, which results from the asynchronous, decoupled redundant thread execution, is shown. Finally, the behaviour of double execution and the thread restart recovery mechanism with transient and permanent faults are discussed.

7.2.1 Execution Overhead of Double Execution

Ideal Comparison

This subsection presents the execution overhead of double execution with an ideal, zero-latency comparator.

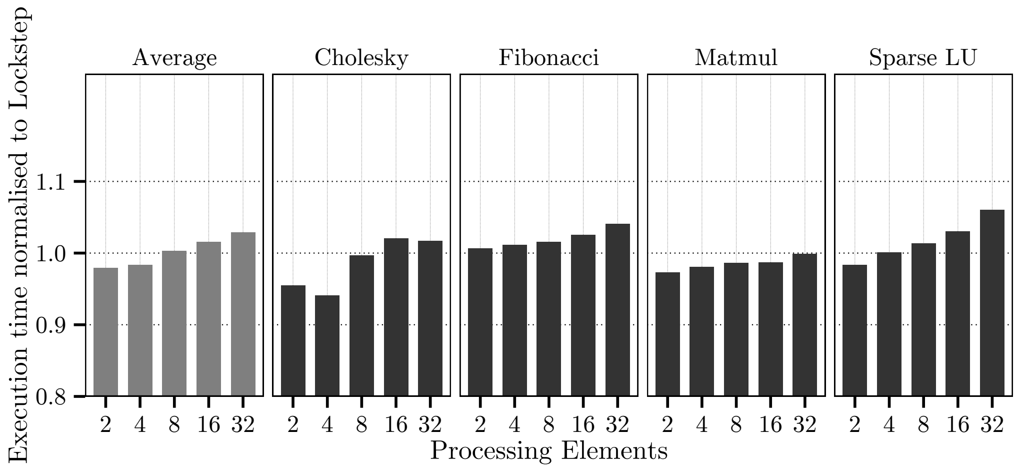


Figure 7.3: Execution time of double execution normalised to the execution time of *Lockstep* with the same number of PEs (*HighUtil* input set).

HighUtil Figure 7.3 shows the normalised execution time of double execution in relation to the execution time of *Lockstep*. It can be seen that double execution shows only a small difference compared to *Lockstep*. The average normalised execution time of double execution over all executed benchmarks reveals that double execution is faster for smaller system configurations (0.98 for the 2 PE system), while the execution time increases for larger systems (1.029 for the 32 PE system).

The main reason for faster execution of double execution compared to *Lockstep* stems from the decoupled execution of the leading and the trailing threads. Since the duplicated threads operate on the same physical memory, they can serve as prefetchers for each other. We found this effect for Matmul, Sparse LU, and Cholesky, where the average thread execution time is faster compared to *Lockstep*.

Sparse LU, Matmul, and Cholesky operate on relatively large input data compared to Fibonacci and are executed faster than *Lockstep* on the 2–4 PE system configurations. Matmul, which uses only the TF for communication between the data-flow threads, achieves a speedup over *Lockstep* for all simulated systems. We can see that the overhead of double execution increases for systems with more PEs for all benchmarks. This effect stems from the PE blocking time and the reduced parallelism due to the asynchronous thread execution and commit. Since Fibonacci can not benefit from the cache due to its small thread input data, its overhead increases for larger systems. For Sparse LU the overhead mainly results from the decreasing utilisation of systems with more PEs.

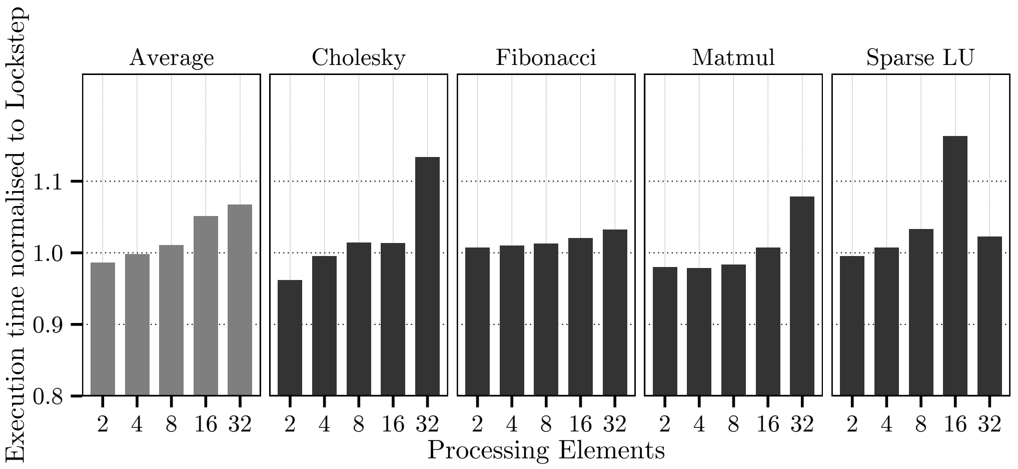


Figure 7.4: Execution time of double execution normalised to the execution time of *Lockstep* with the same number of PEs (*LowUtil* input set).

LowUtil Figure 7.4 depicts the normalised execution time of double execution for the *LowUtil* input set. Since the parallelism of the benchmarks for the *LowUtil* input set is smaller, the execution time of double execution increases compared to the *HighUtil* input set. One reason for the increasing execution time is the lower commit slack between the redundant threads. While leading threads are always started before trailing threads, it may happen that trailing threads are executed faster and therefore finish before their leading thread counterparts. In these cases, a trailing thread's PE is blocked until the leading thread has finished execution, too. This effect can be seen in larger systems for Cholesky, Matmul, and Sparse LU. The parallelism of the benchmarks is not able to utilise all PEs, which in turn reduces the start slack. Another reason is the reduced parallelism of the *LowUtil* input set, because subsequent threads can only be started when both redundant threads have finished execution. This effect has a strong impact on the execution of the *LowUtil* input set, since the parallelism of this input set is already low. We conclude that double execution can not benefit from underutilised PEs, in particular, the overhead for double execution increases with reduced parallelism of the benchmark. Another interesting effect can be seen for Sparse LU, where the overhead decreases for the 32 PE configuration. We assume here that the very low utilisation of the system ($\approx 50\%$) allows resolving thread dependencies earlier, which increases the system utilisation of double execution compared to *Lockstep*.

Non-Ideal Comparison

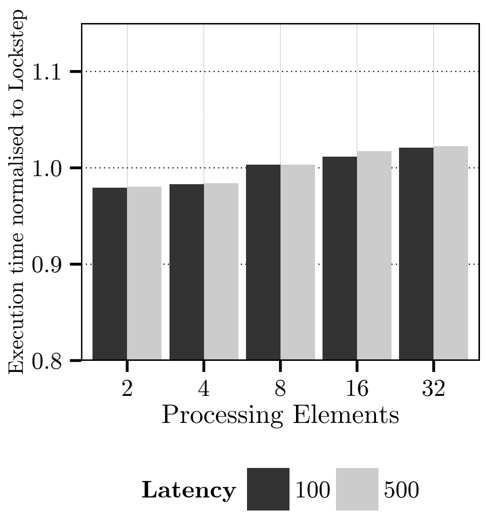
For the thread restart recovery, the trailing thread's PE must buffer the computational results until the comparison of the signatures of the redundant threads confirms the absence of any error. This requires blocking the PE of the leading thread for the time of the signature comparison. However, in Chapter 5 we proposed to exploit the commit slack between the leading and the trailing threads to transfer the leading thread's signature to the core probe of the trailing thread's PE. This can reduce the blocking time of the PEs, when the commit slack between the redundant threads is high enough. Table 7.6 depicts the effective PE blocking time for the simulated comparison latencies of 100 ns and 500 ns averaged over all configurations and benchmarks. For 100 ns most of the latency can be masked by the commit slack. If the comparison latency is increased to 500 ns, the average core blocking time can be still significantly reduced by the proposed technique.

The average PE blocking times for the 100 ns comparison latency are 6.6 ns (*HighUtil* input set) and 7.1 ns (*LowUtil* input set). The PE blocking times for the 500 ns comparison latency are 119.9 ns (*HighUtil* input set) and 120.1 ns (*LowUtil* input set). The *LowUtil* input set shows that the blocking time increases with lower parallelism of the benchmarks. This effect is caused by the lower commit slack, since the low parallelism of the *LowUtil* benchmarks requires starting trailing and leading threads with a smaller start slack, which at the end also leads to a smaller commit slack. Nevertheless, looking at all benchmarks, we can see that the comparison latency can be significantly reduced by the asynchronous redundant thread execution.

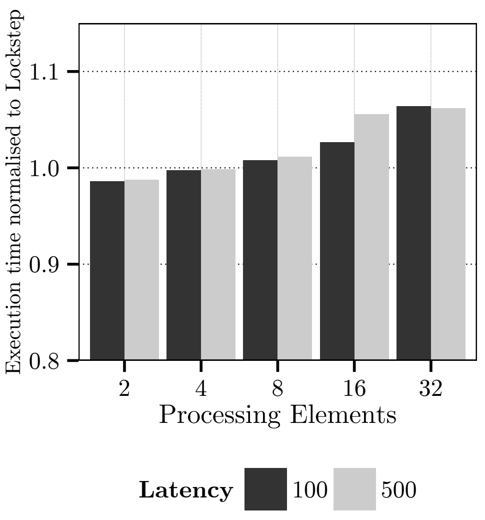
An alternative implementation to decrease the comparison latency would be the implementation of double buffering, which would allow subsequent threads to be started even when the signature comparison has not finished. However, double buffering would require the duplication of the buffer and port sizes for the write buffer and the OWM buffer, since the newly started thread's `twrites` and OWM writes must be kept in a second buffer of the same size.

Table 7.6: Effective PE blocking for comparison latencies of 100 ns and 500 ns.

Comparison Latency	100 ns	500 ns
HighUtil	6.6 ns	119.9 ns
LowUtil	7.1 ns	120.1 ns



(a) *HighUtil* input set



(b) *LowUtil* input set

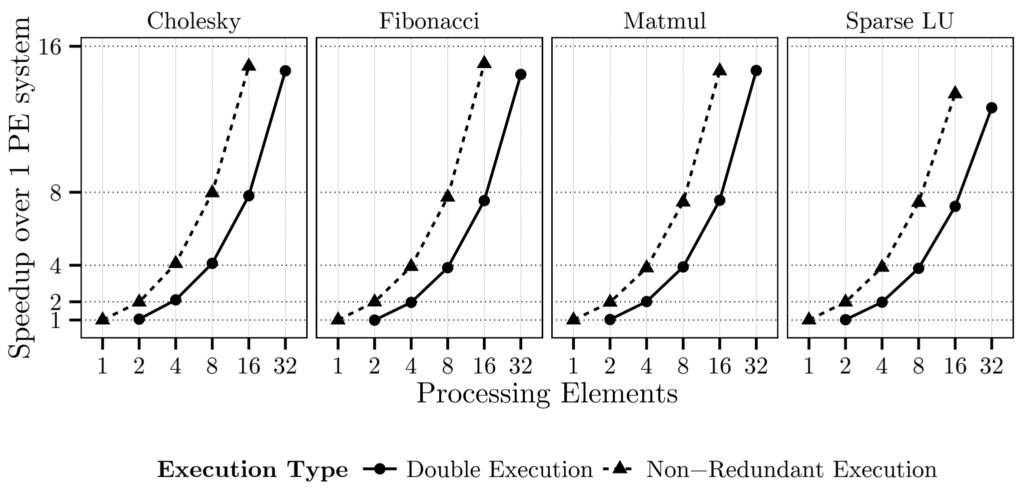
Figure 7.5: Average execution time overhead for all benchmarks with comparison latencies of 100 ns and 500 ns.

Figure 7.5 shows that the PE blocking has only a small impact on the execution time of double execution. The average overhead for a comparison latency of 100 ns is not measurable in the simulator for the *HighUtil* input set and is 0.01% for the 500 ns latency. For the *LowUtil* input set, we found an overhead of 1.6% for a comparison latency of 100 ns and 2.4% for a comparison latency of 500 ns.

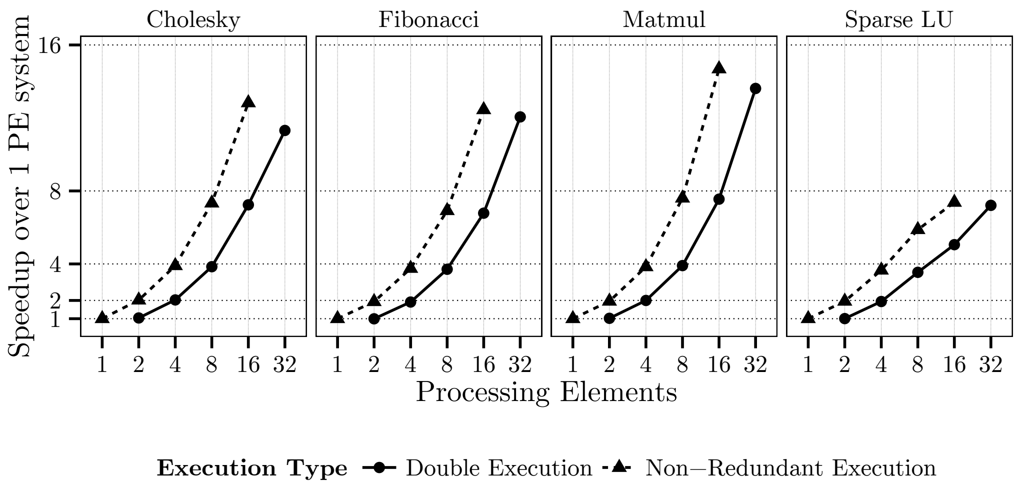
7.2.2 Scalability

Figure 7.6 compares the speedup of double execution with the speedup of the non-redundant execution on half of the PEs. It can be seen that in the best case, the speedup of double execution is equal to the speedup of the non-redundant execution. While double execution scales with the available PEs in the systems, we can also seen that double execution can not speedup the execution compared to the non-redundant execution of Cholesky, Fibonacci, and Sparse LU. Only Matmul reveals a small speedup, because of the cache effects and its equal execution speed of the threads.

The reason for the decreasing speedup of the other benchmarks stems from the fact that subsequent threads can only be started, when the previous redundant threads have finished execution and their signatures have been compared. This means that in the best case, the parallelism of the threads is equal to the non-redundant execution on half of the PEs. However, benchmarks with smaller parallelism reveal that the scalability of double execution can be significantly reduced compared to the non-redundant execution. This can be observed for both input sets. On average the speedup of the *HighUtil* input set is 5.8 for non-redundant and 5.7 for double execution, while for the *LowUtil* input set, the speed up further decreases for double execution. The speed up for the *LowUtil* input set is 5.1 for the non-redundant execution and 4.9 for the redundant execution.



(a) Scalability of non-redundant and double execution normalised to the non-redundant execution on one PE (*HighUtil* input set).

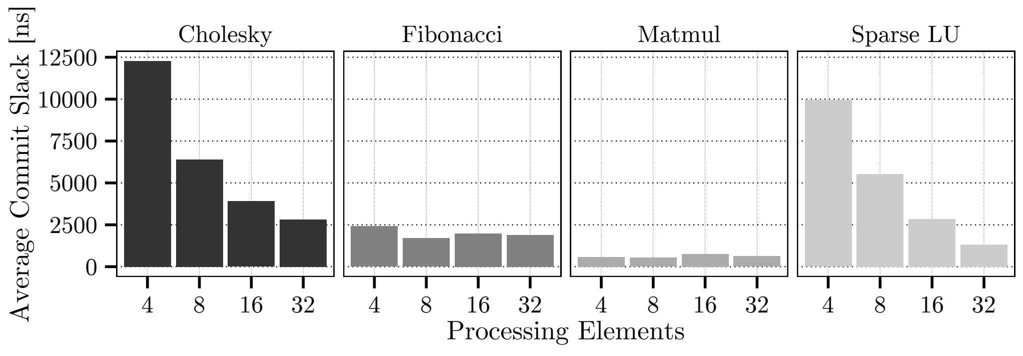


(b) Scalability non-redundant and double execution normalised to the non-redundant execution on one PE (*LowUtil* input set).

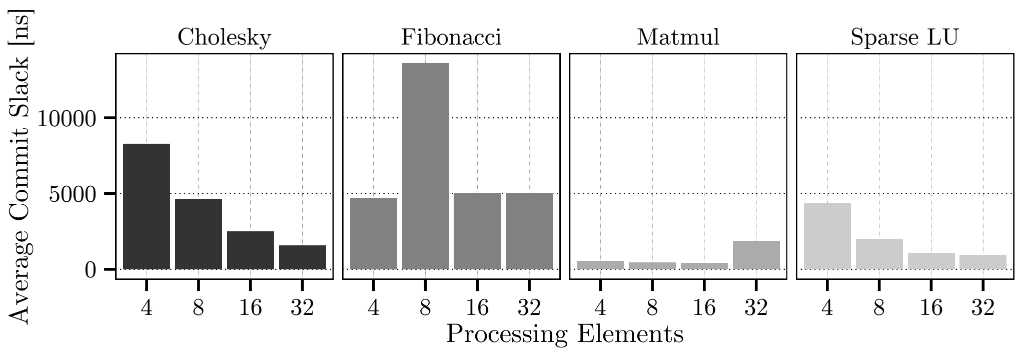
Figure 7.6: Scalability of non-redundant data-flow execution and double execution

7.2.3 Commit Slack

In contrast to *Lockstep*, double execution executes redundant threads completely decoupled from each other. While this has advantages for the flexible resource usage and the hardware implementation of double execution, it can also lead to asynchronous execution of the redundant threads. The asynchronous execution can be used to mask the comparison latency and exploit the leading thread as a prefetcher, however, it also increases the error detection latency and reduces the parallelism of the application, since subsequent data-flow threads can be only started and errors detected, when both threads have finished execution. Although the TSU prioritises waiting trailing threads over the leading threads to reduce the commit slack, we found that the decoupled thread execution can lead to a significant commit slack.



(a) Average commit slack of the *HighUtil* input set.



(b) Average commit slack of the *LowUtil* input set.

Figure 7.7: Commit slack of double execution.

Figure 7.7 shows the average commit slack of double execution for the *HighUtil* and *LowUtil* input sets.

HighUtil The results of the *HighUtil* input set (Figure 7.7(a) show that Cholesky and Sparse LU have a significant higher commit slack compared to Fibonacci and Matmul. This indicates that the leading and trailing threads are executed at different speed in Cholesky and Sparse LU, which is a result of the data sharing and the cache behaviour of these benchmarks. For Cholesky and Sparse LU, the commit slack decreases with higher parallelism of the system, which stems from the fact that in systems with more available PEs, the probability that a PE becomes available for the execution of a trailing thread is higher, which in turn leads to lower start and commit slacks. Matmul shows the smallest commit slack of all benchmarks. Interestingly, Matmul's commit slack remains constant even on systems with more available PEs. This behaviour stems from the thread characteristic of matmul, where all threads have a relative equal execution time.

LowUtil Figure 7.7(b) depicts the average commit slack of the *LowUtil* input set. For Cholesky, Matmul, and Sparse LU the commit slack is lower compared to the *HighUtil* execution. This is also an effect of the smaller start slack because of the lower parallelism of the *LowUtil* input set. By contrast, Fibonacci shows a higher average commit slack.

The commit slack of Fibonacci is strongly run time depended, which is caused by differing scheduling decisions of the TSU scheduler in the different system configurations. For Matmul and Sparse LU, the evaluation reveals a decreasing commit slack for larger system configuration, which is a consequence of the lower parallelism of the *LowUtil* input set.

7.2.4 Graceful Degradation under Permanent Faults

In order to evaluate the ability of the proposed mechanisms to cope with permanent faulty PEs, we simulated the baseline configuration with a certain number of deactivated PEs, as described in Section 7.1.4.

HighUtil The Figures 7.8 (4 and 8 PE system) and 7.9 (16 and 32 PE system) show the execution time of double execution normalised to the execution time on a non-faulty system, when 0–8 PEs are permanently deactivated. The figures depict only configurations, which are still functional and exclude configurations, which have no functional PEs left. Compared to systems without deactivated PEs, we can see that the systems' performance gracefully degrades with the number of the deactivated PEs. In particular, for system configurations with 16–32 PEs, double execution is flexible enough to compensate even higher numbers of permanent faulty PEs.

LowUtil Figure 7.10 (4 and 8 PE system) and 7.11 (16 and 32 PE system) depict the graceful degradation of double execution, when 0–8 PEs are permanently faulty for the *LowUtil* input set. Compared to the *HighUtil* input set, the systems can better compensate the deactivated PEs, since the benchmarks can not fully utilise all PEs, as the evaluation of the scalability of double execution has shown before and therefore the lost performance by the deactivated PEs has a smaller performance impact. In particular, this effect can be seen for the 32 PE configuration executing Fibonacci and Sparse LU. Due to the limited parallelism of both benchmarks, several PEs can be deactivated without performance impact. In both cases, the underutilised PEs serve as flexible spare PEs and can be used without significant performance impact.

Nevertheless, for both input sets the execution time of double execution gracefully degrades even when multiple PEs of a system are deactivated. We conclude that double execution can exploit all functional PEs of a system, even when some PEs are permanently deactivated and can gracefully reduce the performance of the system, while being still operational.

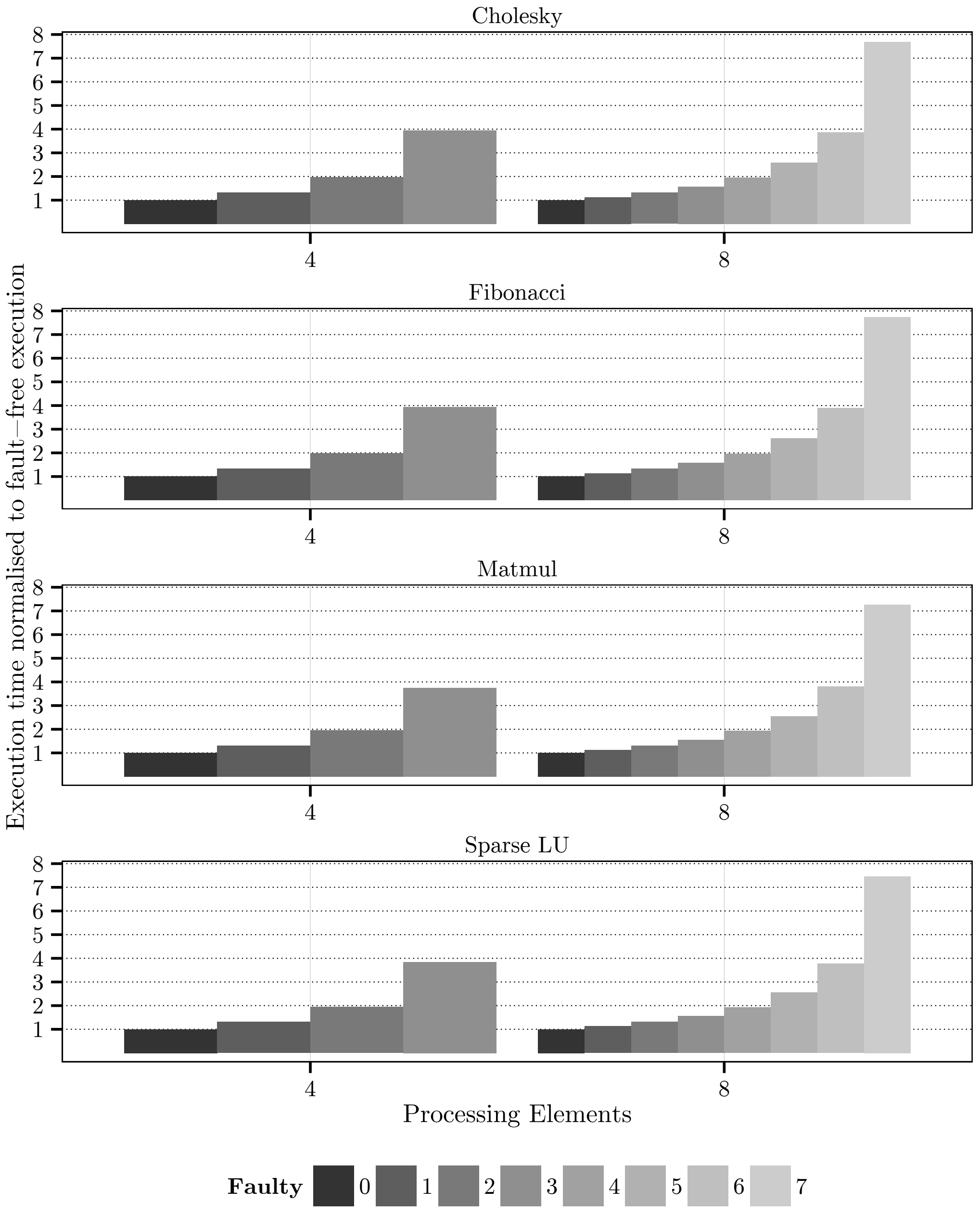


Figure 7.8: Graceful degradation of double execution, when 0–7 PEs are permanently faulty in the 4 and 8 PE systems (*HighUtil*).

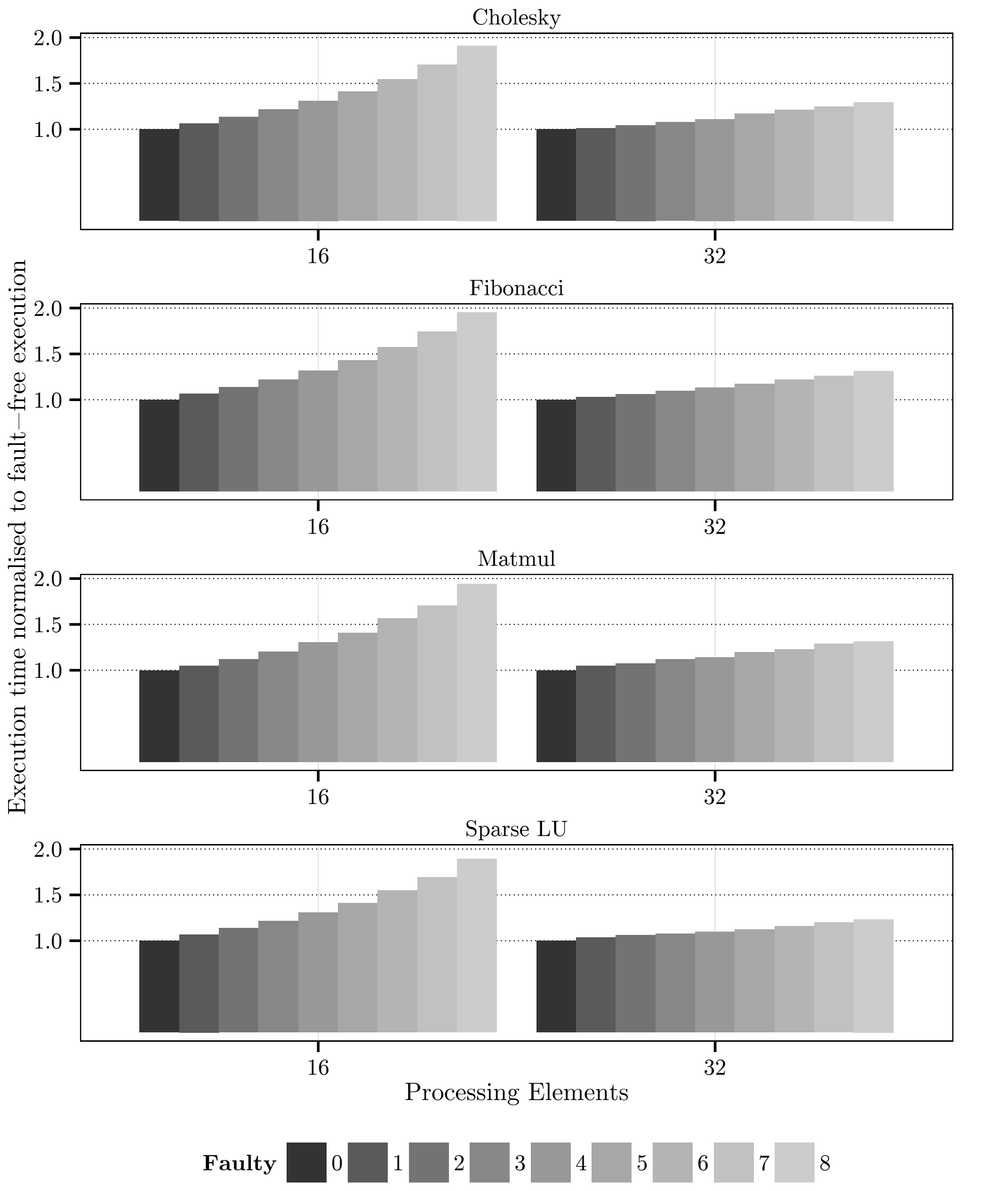


Figure 7.9: Graceful degradation of double execution, when 0–8 PEs are permanently faulty in the 16 and 32 PE systems (*HighUtil*).

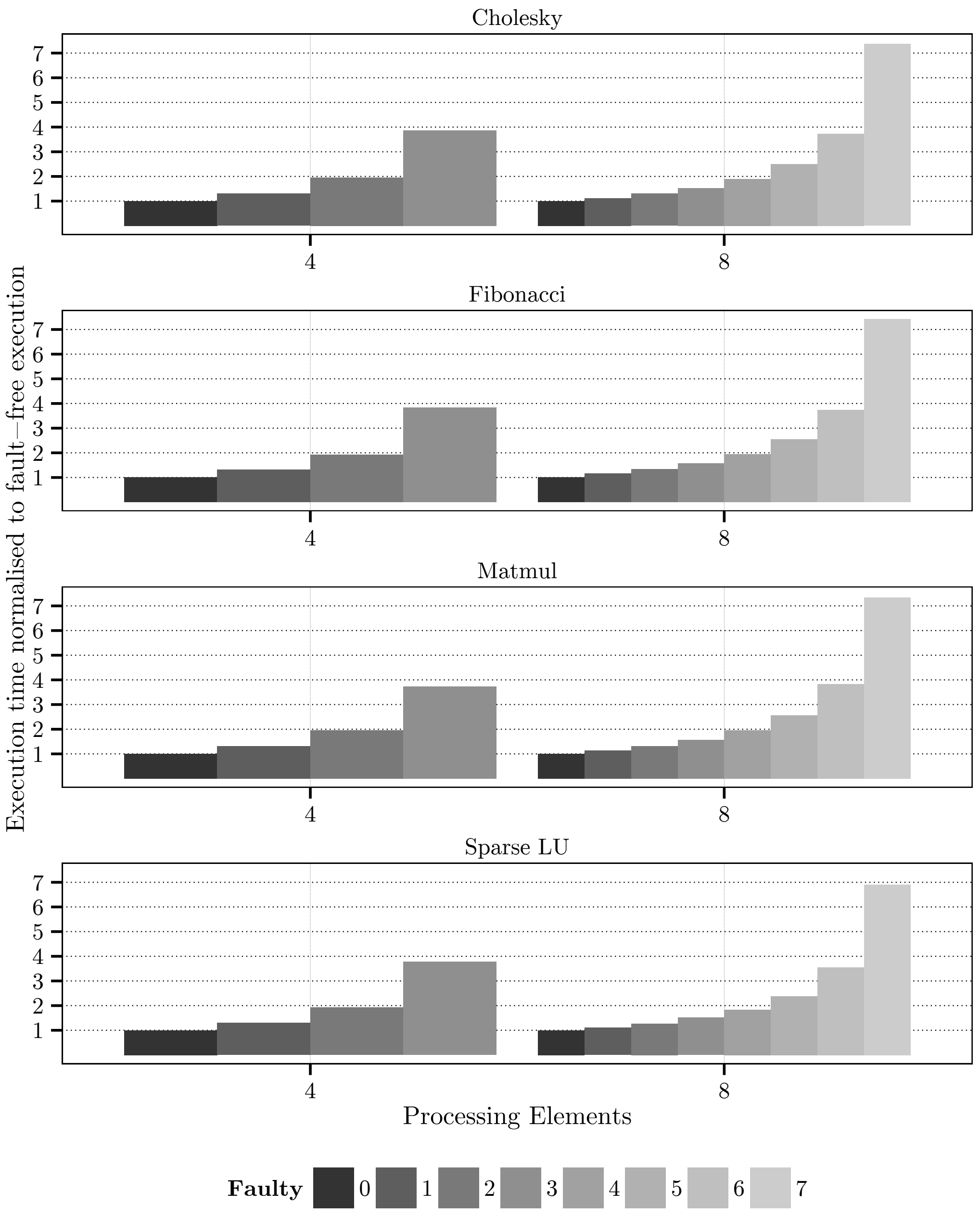


Figure 7.10: Graceful degradation of double execution, when 0–7 PEs are permanently faulty in the 4 and 8 PE systems (*LowUtil*).

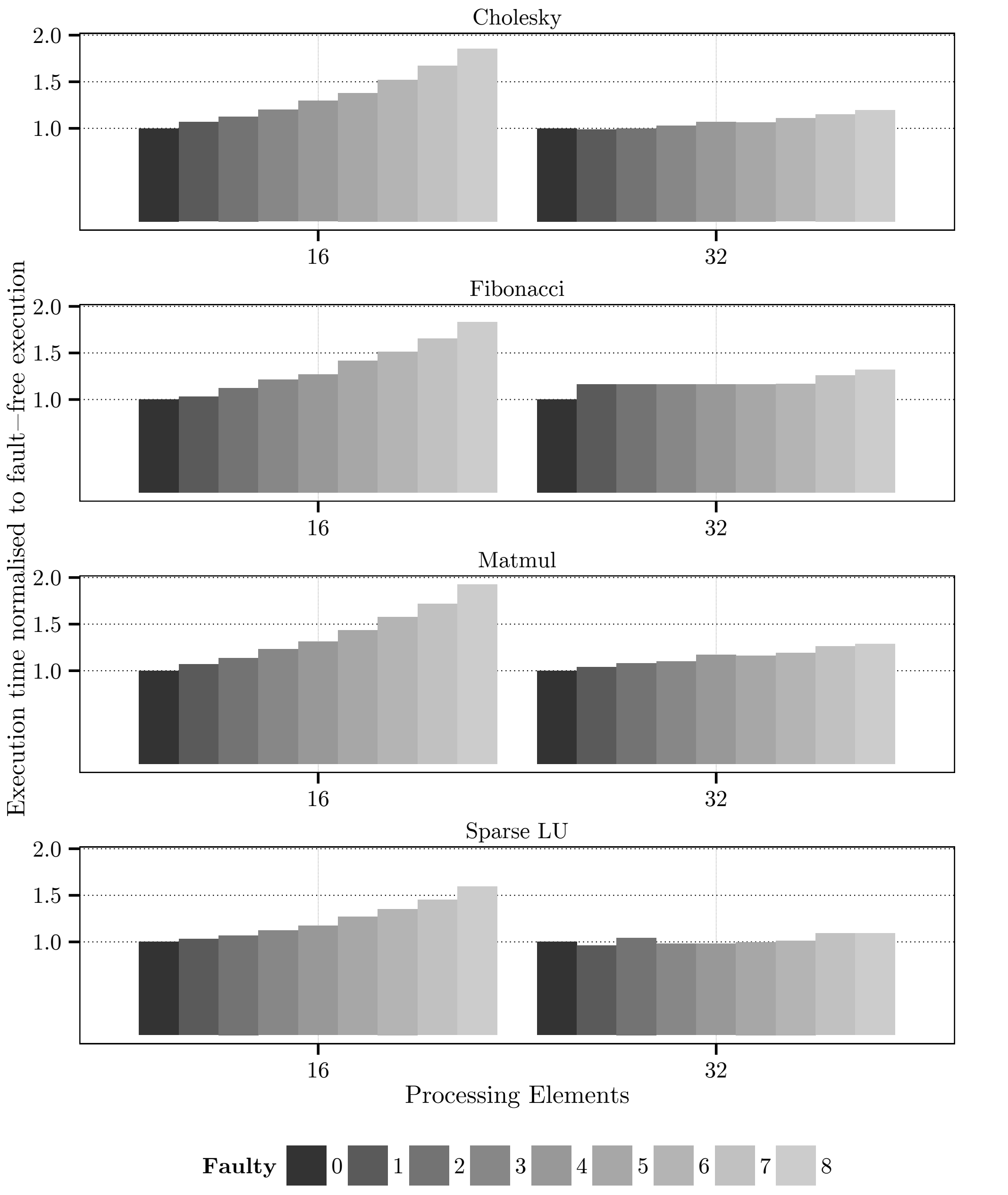


Figure 7.11: Graceful degradation of double execution, when 0–8 PEs are permanently faulty in the 16 and 32 PE systems (*LowUtil*).

7.2.5 Execution Overhead under Transient Faults

Since thread restart recovery promises low-overhead recovery, we also evaluate the execution overhead with transient faults. We inject faults at a fixed time interval of 10^5 ns. After a fixed amount of faults, the system stops the injection and proceeds without faults. We investigate the impact of 5 and 10 faults on the simulated system configurations.

The results for the *HighUtil* shown in Figure 7.12, depict that thread restart recovery has a negligible effect on the execution time of double execution in case of transient faults. This is not surprising, since in case of the *HighUtil* input set, the benchmarks execute several thousand threads.

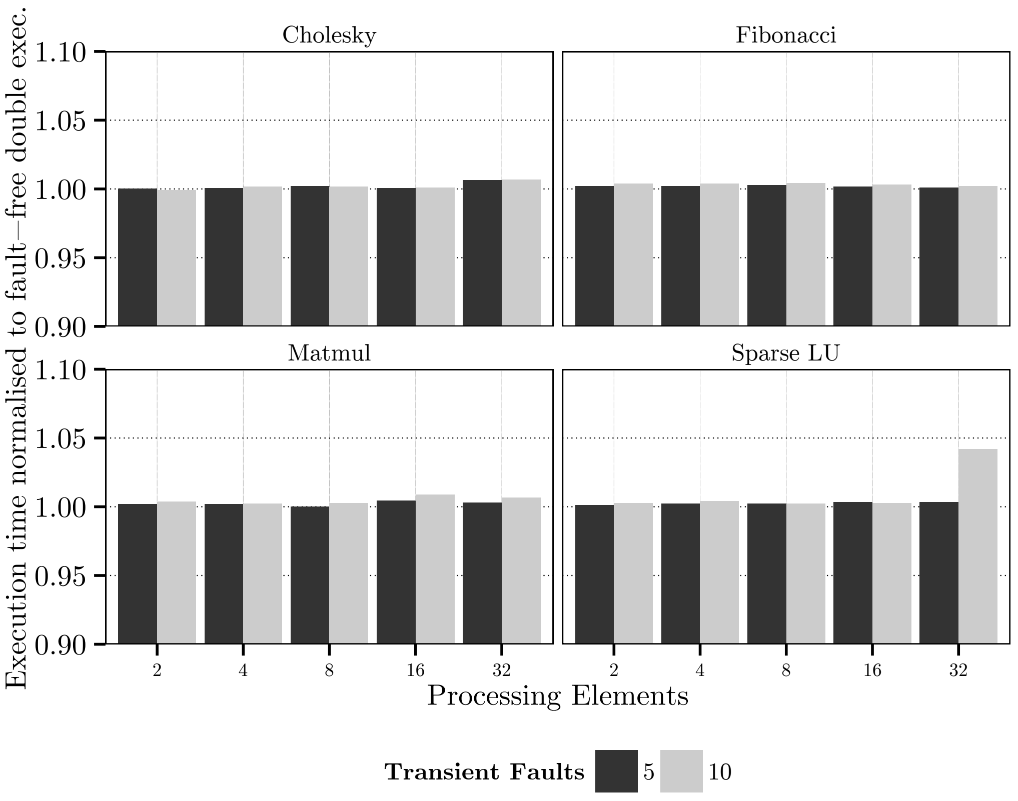


Figure 7.12: Execution time of double execution normalised to double execution without faults, when 5 and 10 faults occur (*HighUtil*).

By contrast, the evaluation of the *LowUtil* input set, depicted in Figure 7.13, shows a measurable impact of the thread restart recovery, since for the *LowUtil* input set the benchmark execute a significant smaller number of data-flow threads, which in turn increases the relative execution time overhead for a thread restart. In particular, the Fibonacci benchmark, which executes only 68 threads in case of the *LowUtil* input set, suffers from a significant overhead, in particular when 10 threads are restarted. Nevertheless, we conclude that thread restarts can be used for efficient recovery, even under the occurrence of multiple faults.

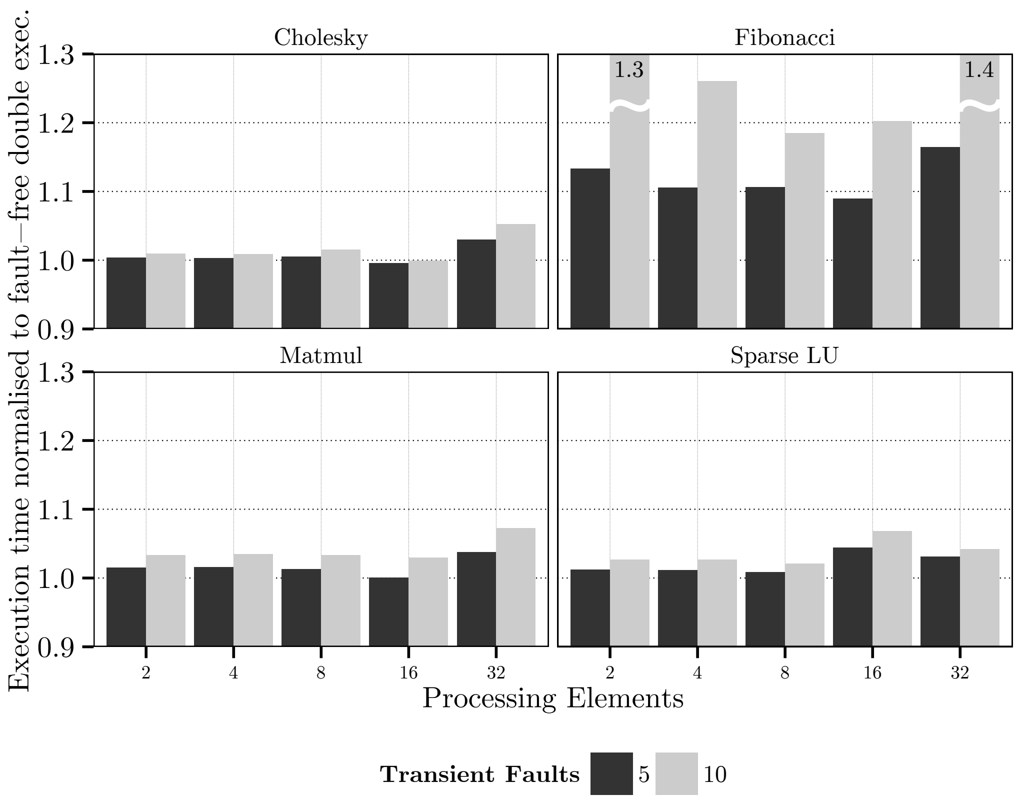


Figure 7.13: Execution time of double execution normalised to double execution without faults, when 5 and 10 faults occur (*LowUtil*).

7.3 Optimistic Double Execution

This section presents evaluation results on optimistic double execution and data-flow based global checkpointing. First, the execution overhead of optimistic double execution without checkpointing is presented. Second, the execution overhead of optimistic double execution with data-flow based global checkpointing with different checkpoint intervals is shown. Third, the log sizes of data-flow based global checkpointing in relation to the log sizes of the *LockstepCheck* system are discussed. Finally, the impact of permanent and transient faults on the execution time of optimistic double execution and data-flow based global checkpointing are presented.

The performance of double execution can be reduced with a higher comparison latency, since the output comparison latency is on the critical path of double execution. By contrast, optimistic double execution completely removes the comparison latency from the critical path by immediately committing the leading thread. This instantaneous commit decouples the thread commit from the signature comparison and does not require PE blocking until signature comparison. Therefore, we do not discuss different comparison latencies in this section.

Furthermore, to simplify the simulation, we assume that in case of optimistic double execution, the DMA transfer for duplicated OWM sections is implemented over an additional interconnect and causes no performance overhead.

7.3.1 Execution Overhead without Checkpoints

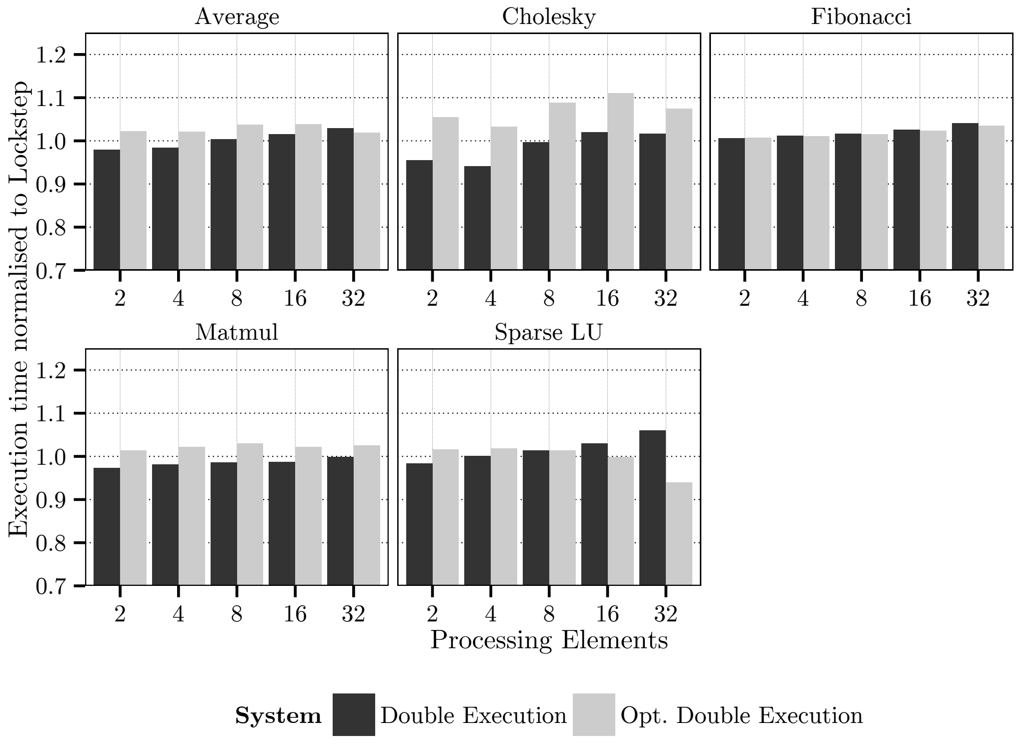


Figure 7.14: Execution time of double execution and optimistic double execution normalised to *Lockstep* (*HighUtil*)

HighUtil Figure 7.14 depicts the execution time of optimistic double execution normalised to *Lockstep*. Compared to double execution, optimistic double execution introduces additional overhead for applications, which transfer larger data blocks between threads (Cholesky, Matmul, and Sparse LU). One reason for this behaviour is the reduced prefetching effect compared to double execution. For optimistic double execution the trailing and the leading threads have different physical OWM addresses, such that the redundant threads can not serve as prefetchers for the OWM sections. Additionally, the leading and the trailing threads are executed with a higher commit slack, since the TSU prioritises leading over trailing threads to increase the parallelism (see Subsection 7.3.4). While the Matmul benchmark does not use the OWM region, the asynchronous execution of the redundant threads can reduce the temporal locality in the caches. Since the small parallelism of Sparse LU can not exploit all PEs of the 32 PE system, optimistic double execution can already benefit from the underutilised PEs to speed up the execution.

For Fibonacci, we can see a small speedup compared to double execution. While this is already an effect of the increasing parallelism with optimistic double execution, the already high parallelism of Fibonacci reduces the possibility to greater exploit underutilised PEs. The results show that optimistic double execution can trade higher parallelism against reduced single thread performance, since the temporal and spatial locality in the caches decreases with the higher commit slack of optimistic double execution and the additional costs for the OWM address mapping. However, in case of the *HighUtil* input set, which provides enough parallelism to utilise all system configurations, optimistic double execution leads to increasing execution time. This overhead is evident for benchmarks, which benefit from prefetching effects in double execution.

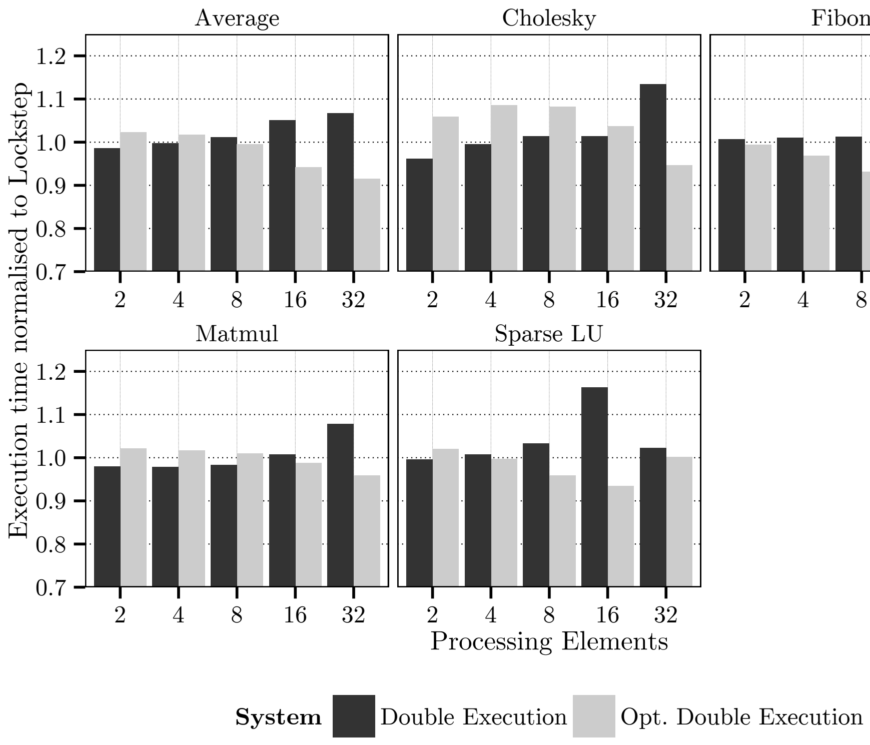
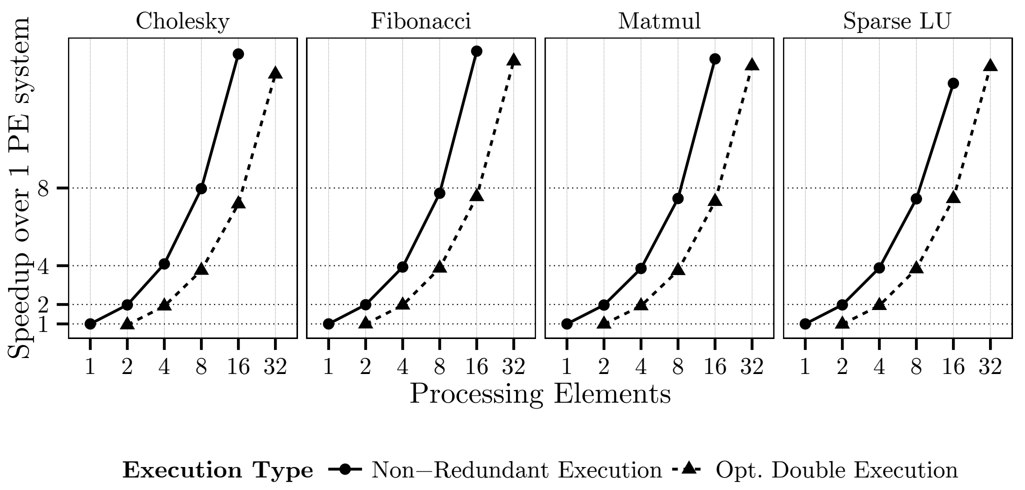


Figure 7.15: Execution time of double execution and optimistic double execution normalised to *Lockstep* (*LowUtil*)

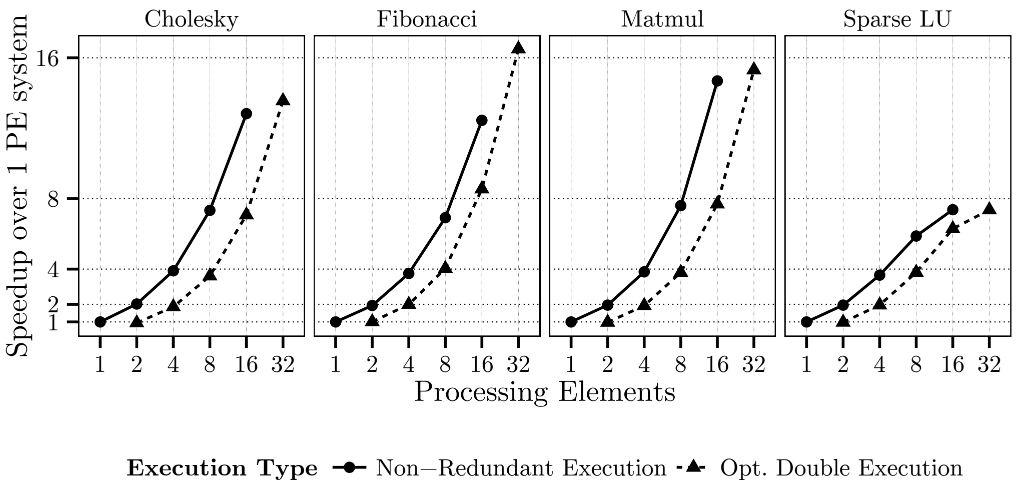
LowUtil The *LowUtil* input set, depicted in Figure 7.15, shows the capability of optimistic double execution to exploit underutilised resources to speed up the redundant execution. Optimistic double execution exploits the effect that all thread dependencies have already been resolved, when the redundant threads are moved to the RQ. This means that all trailing threads can be executed in parallel, since their dependencies have already been resolved and the inputs are available for execution. The evaluation of the *LowUtil* input set reveals that all benchmarks can better utilise the available PEs. The best speedup compared to *Lockstep* is seen for Fibonacci, which is ≈ 25% faster than *Lockstep* and ≈ 28% faster than double execution. Another interesting effect can be seen for Sparse LU. While the execution time of optimistic double execution for Sparse LU is decreasing until 16 PEs, the execution times of optimistic double execution and double execution for the 32 PE system are relatively equal. This effect results from the very low parallelism of Sparse LU for the *LowUtil* input set, which means that the utilisation of the system is low enough such that scheduling decisions for both double execution variants are quite similar, leading to the same overhead. For the 32 PE system, the *LowUtil* input set is executed ≈ 8% faster than *Lockstep* and ≈ 15% faster than double execution on average.

7.3.2 Scalability

Figure 7.16(a) depicts the scalability of optimistic double execution for the *HighUtil* input set. Optimistic double execution can increase the parallelism of double execution and benefits from underutilised PEs. This is particular the case for the *LowUtil* input set with its lower parallelism. Compared to the non-redundant execution on half of the PEs, we can see that the scalability of optimistic double execution is relatively similar to double execution. However, when the utilisation of the system decreases, which is the case for the 32 PE system executing Sparse LU, the speedup of optimistic double execution exceeds the non-redundant execution. This shows that optimistic double execution can exploit underutilised PEs in the system to speed up redundant thread execution.



(a) Scalability of non-redundant and optimistic double execution in relation to the non-redundant execution on one PE (*HighUtil* input set).



(b) Scalability of non-redundant and optimistic double execution in relation to the non-redundant execution on one PE (*LowUtil* input set).

Figure 7.16: Scalability of non-redundant execution and optimistic double execution.

Figure 7.16(b) shows that this effect increases for the *LowUtil* input set with its lower parallelism. Especially Cholesky, Fibonacci, and Matmul reveal an increasing speedup, when the parallelism of the benchmarks can not utilise all PEs of the system. Sparse LU achieves also a speedup compared to the non-redundant execution for system configurations up to 16 PEs. However, for the 32 PE system the speedup saturates. As explained before, this effect is a result of the very low utilisation of Sparse LU.

7.3.3 Execution Overhead with Checkpoints

Section 7.3.1 presented results of optimistic double execution without checkpointing. However, optimistic double execution can not recover from errors without a checkpoint mechanism. In this subsection, the evaluation results of optimistic double execution with the data-flow based global checkpointing are presented. The results of optimistic double execution with checkpointing are compared with *LockstepCheck*, which always creates the same number of checkpoints as the data-flow based checkpointing mechanism. As mentioned before, we assume a static overhead per checkpoint for *LockstepCheck* of 0.1 µs.

HighUtil Figure 7.17 shows the execution time overhead of optimistic double execution with checkpoint intervals of 10 000 ns, 100 000 ns and 1 000 000 ns for the *HighUtil* input set. At these intervals, the checkpoint creation is triggered. However, when a checkpoint is triggered, all unverified, already committed leading threads must be checked for errors before the new checkpoint can be established. When a checkpoint is triggered, the TSU scheduler prioritises the currently waiting trailing threads to establish the checkpoint as fast as possible. As a consequence, the checkpointing interval influences also the commit slack between the leading and trailing threads (see Subsection 7.3.4). The results show that checkpoint creation introduces only a small overhead compared to the execution without checkpoints for the *HighUtil* input set and can be reduced with longer checkpoint intervals, as seen for Sparse and Cholesky. Compared to *LockstepCheck* we found that optimistic double execution has performance advantages for smaller checkpoint intervals. Interestingly, Matmul is executed faster with checkpoint intervals of 10 000 ns and 100 000 ns. This behaviour results from the reduced asynchronous execution between the leading and the trailing threads.

LowUtil Figure 7.18 depicts the execution time of optimistic double execution for the *LowUtil* input set. Compared to the *HighUtil* input set, checkpointing has a stronger influence on the execution time. For Cholesky and Fibonacci the execution time of optimistic double execution can be significantly increased by the checkpoint creation overhead, eliminating the speedup of optimistic double execution. In case of Cholesky and Fibonacci, the overhead is induced by the low parallelism of the benchmarks and therefore underutilisation of the PEs, while waiting for new checkpoints to be established. For all benchmarks, the overhead induced by the checkpoint creation is reduced with longer checkpoint intervals. However, Cholesky, Matmul, and Sparse LU are still faster than *LockstepCheck* for the 10 000 ns and 100 000 ns checkpoint intervals.

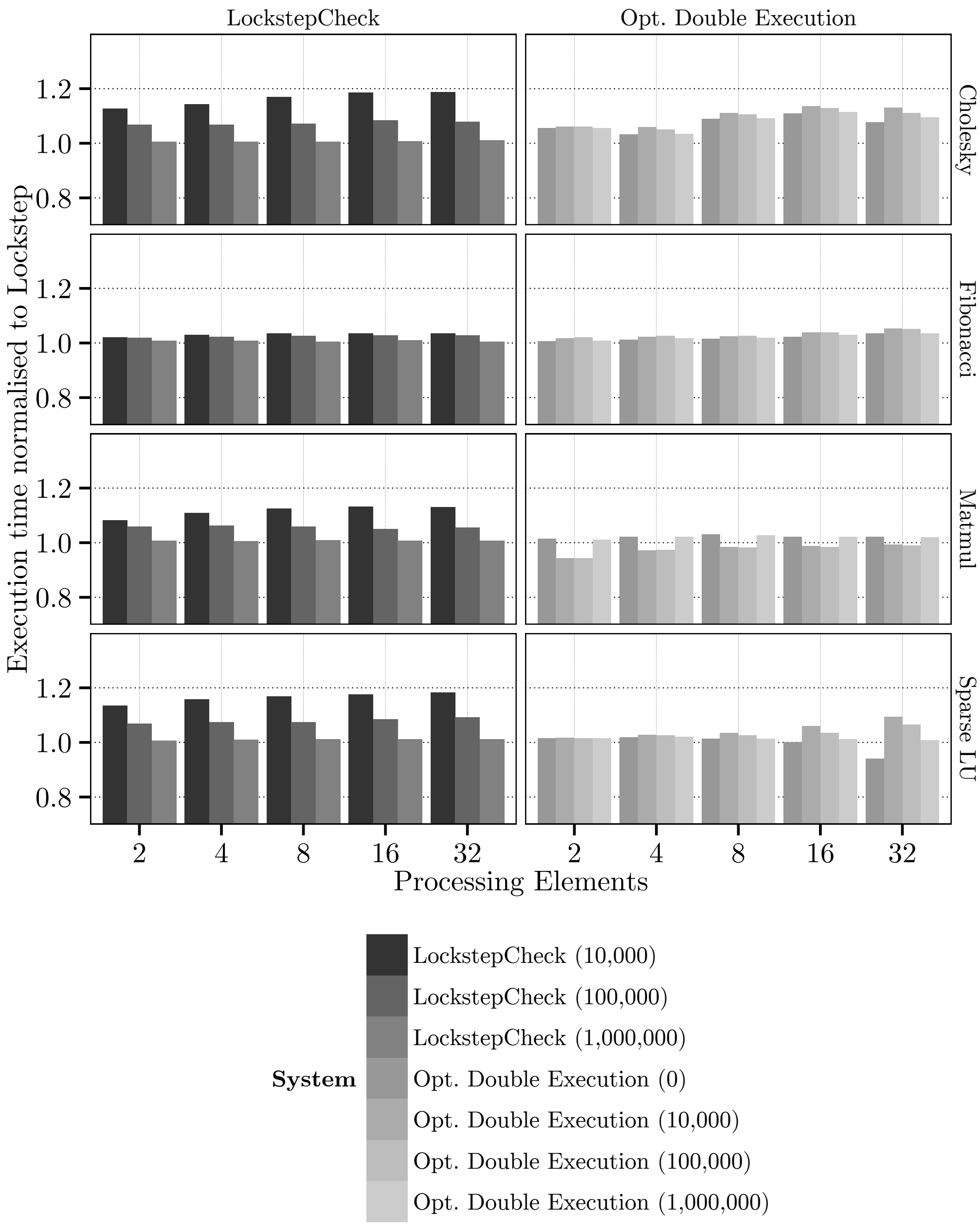


Figure 7.17: Execution time of *LockstepCheck* and optimistic double execution normalised to *Lockstep* (*HighUtil*).

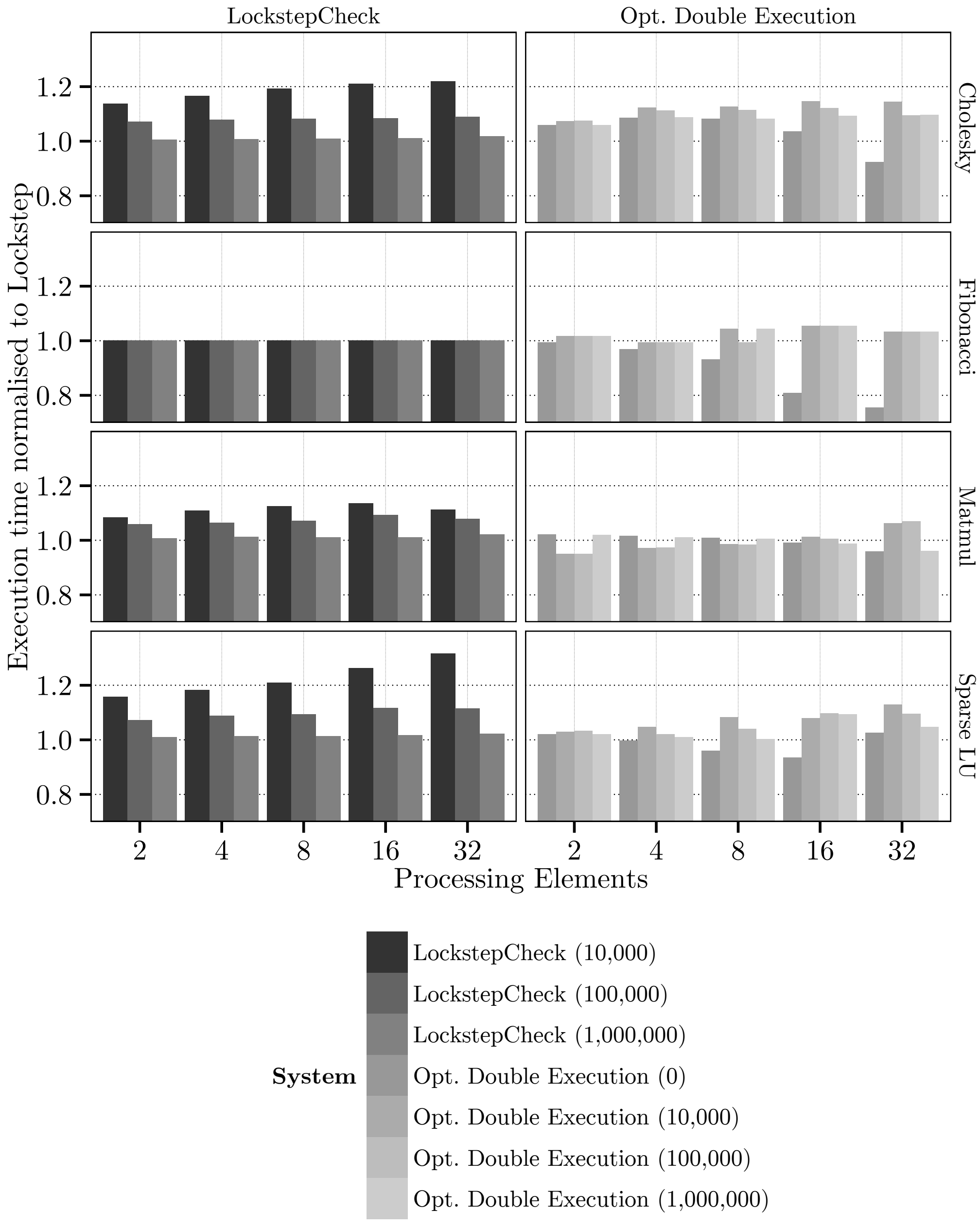


Figure 7.18: Execution time of *LockstepCheck* and optimistic double execution normalised to *Lockstep* (*LowUtil*).

7.3.4 Commit Slack

During optimistic double execution, the leading and trailing threads can be executed completely decoupled from each other. However, this decoupling can lead to a high commit slack between the leading and the trailing threads.

The Figures 7.19 and 7.20 depict the commit slack of optimistic double execution for the checkpoint intervals of 10 000 ns, 100 000 ns, and 1 000 000 ns in case of the *HighUtil* and the *LowUtil* input sets. As expected, the commit slack correlates with the checkpointing interval for both input sets, since a new checkpoint can only be created, when all outstanding trailing threads have been finished and compared. As a consequence, the commit slack for shorter checkpoint intervals is obviously smaller. Similar to double execution, the commit slacks decreases on systems with more PEs. However, for all benchmarks and all simulated system configurations we can state that the commit slack was orders of magnitude higher compared to double execution.

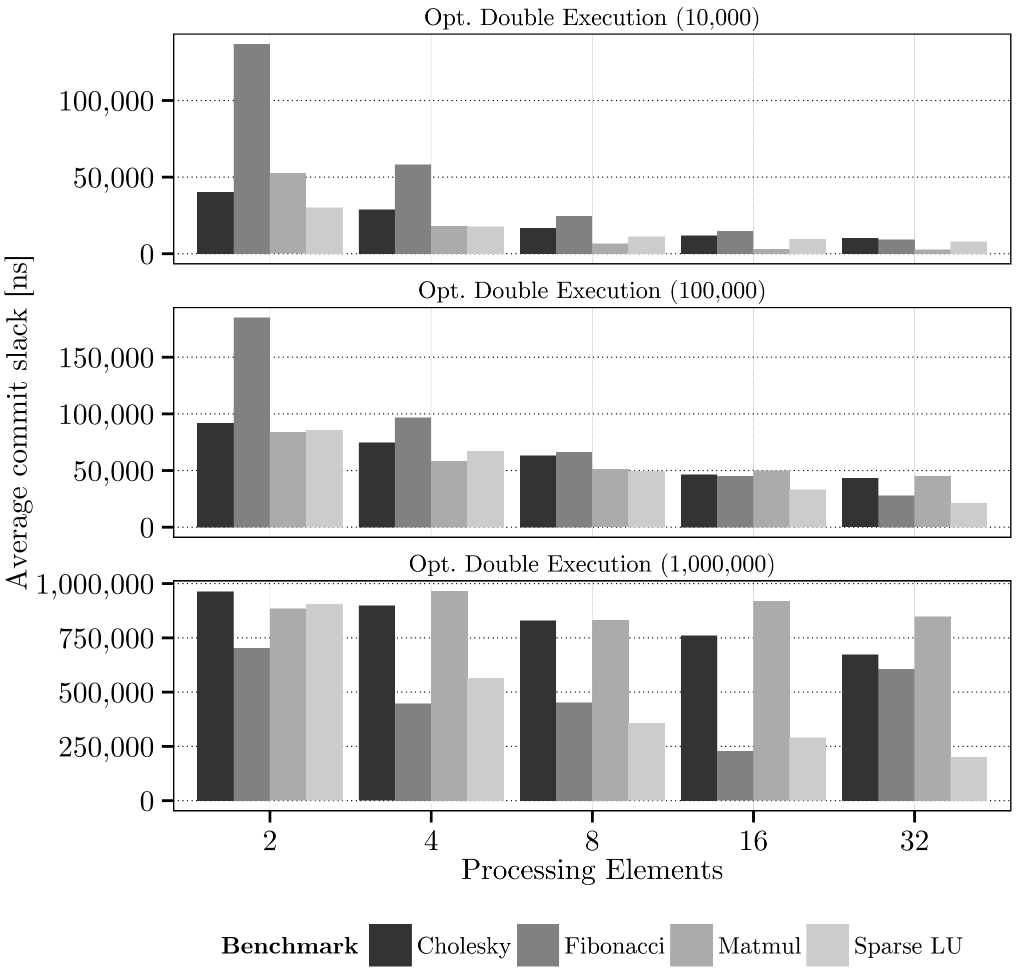


Figure 7.19: Commit slack of optimistic double execution (*HighUtil*).

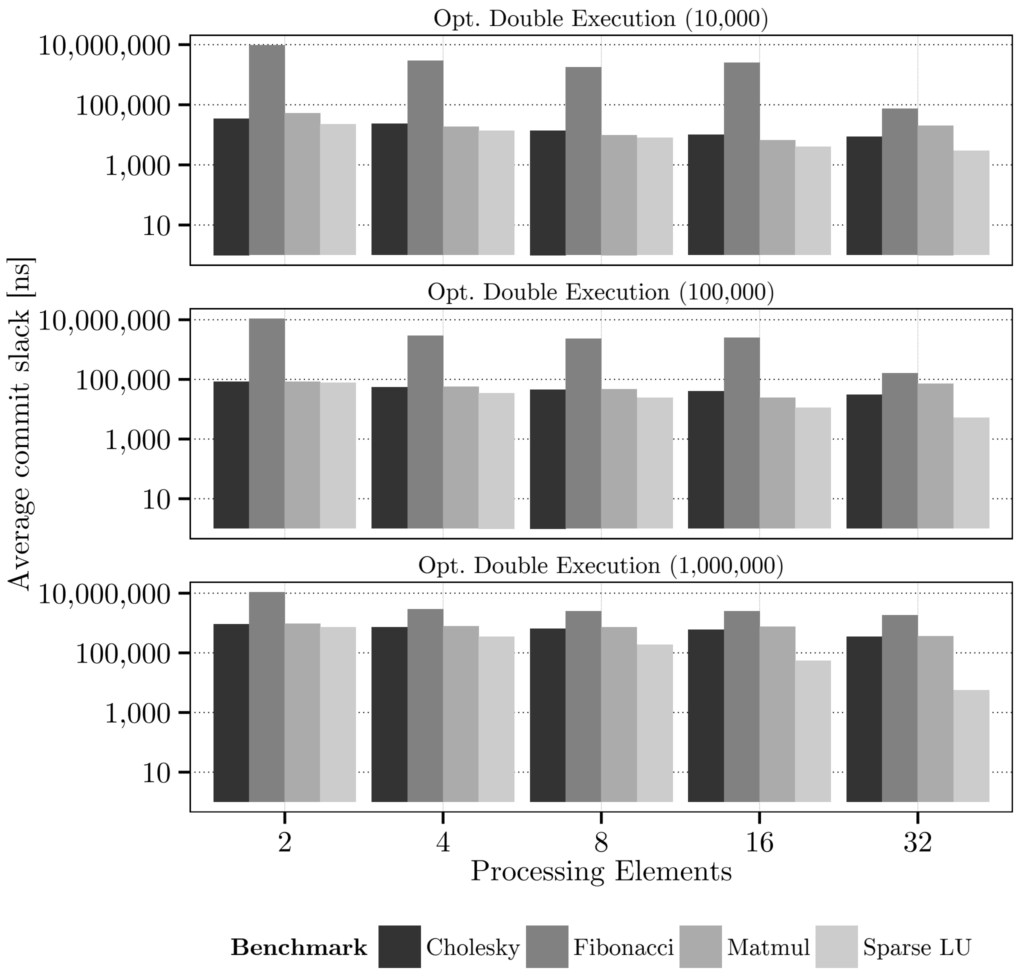


Figure 7.20: Commit slack of optimistic double execution (*LowUtil*).

7.3.5 Log Size of Global Checkpoints

Data-flow based checkpointing can reduce the necessary memory log size, since only the thread frame region and the OWM region must be incorporated in the log. This represents a significant advantage over non-data-flow checkpointing, like *LockstepCheck*, which requires logging of all changes to the main memory. The Figures 7.21 and 7.22 depict the average log size of the data-flow based checkpointing normalised to the average log size of *LockstepCheck*. It can be seen that data-flow based checkpointing reduces the log size requirements for both input sets to only a small fraction compared to *LockstepCheck*. We assume that the increasing normalised log size for larger systems stems from the fact that the data-flow based logging does not use the caches, while *LockstepCheck* creates only new log entries, when data is written back to the main memory. However, the cache size and therefore the storage capabilities of the caches increases with more PEs.

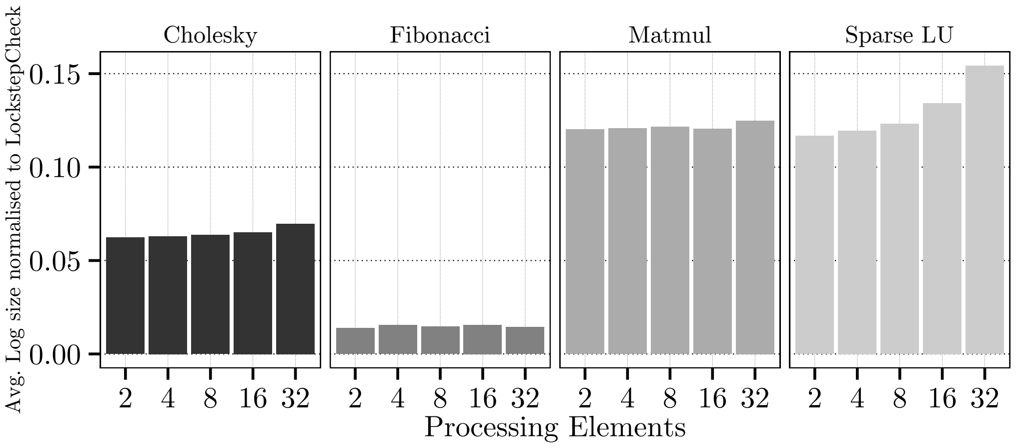


Figure 7.21: Average log size per checkpoint normalised to the log size of *LockstepCheck* (*HighUtil*).

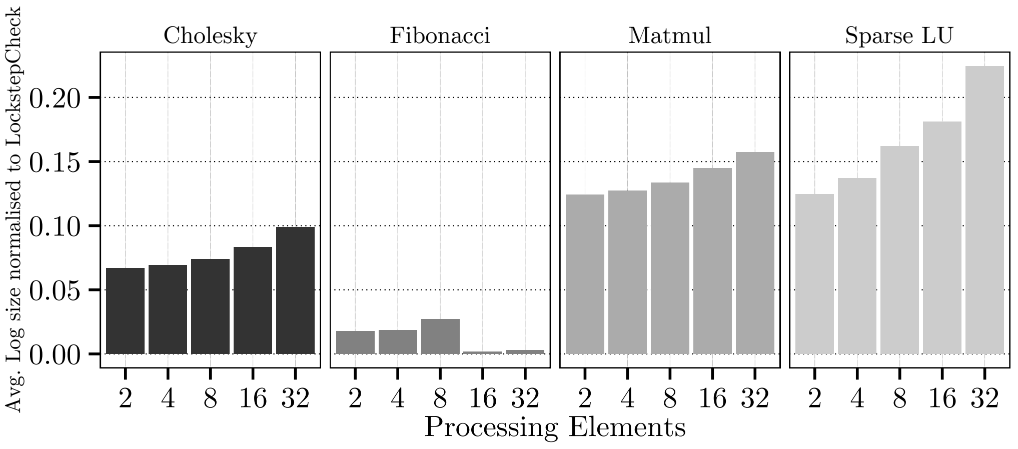


Figure 7.22: Average log size per checkpoint normalised to the log size of *LockstepCheck* (*LowUtil*).

7.3.6 Graceful Degradation under Permanent Faults

The Figures 7.23, 7.24, 7.25, and 7.26 depict the impact of permanent faults on the execution time of optimistic double execution for the *HighUtil* and *LowUtil* input sets. The simulation uses a checkpoint interval of 10 000 ns. Similar to double execution, optimistic double execution can compensate permanent broken PEs and gracefully adapt its performance in case of permanent faulty PEs for all simulated system configurations.

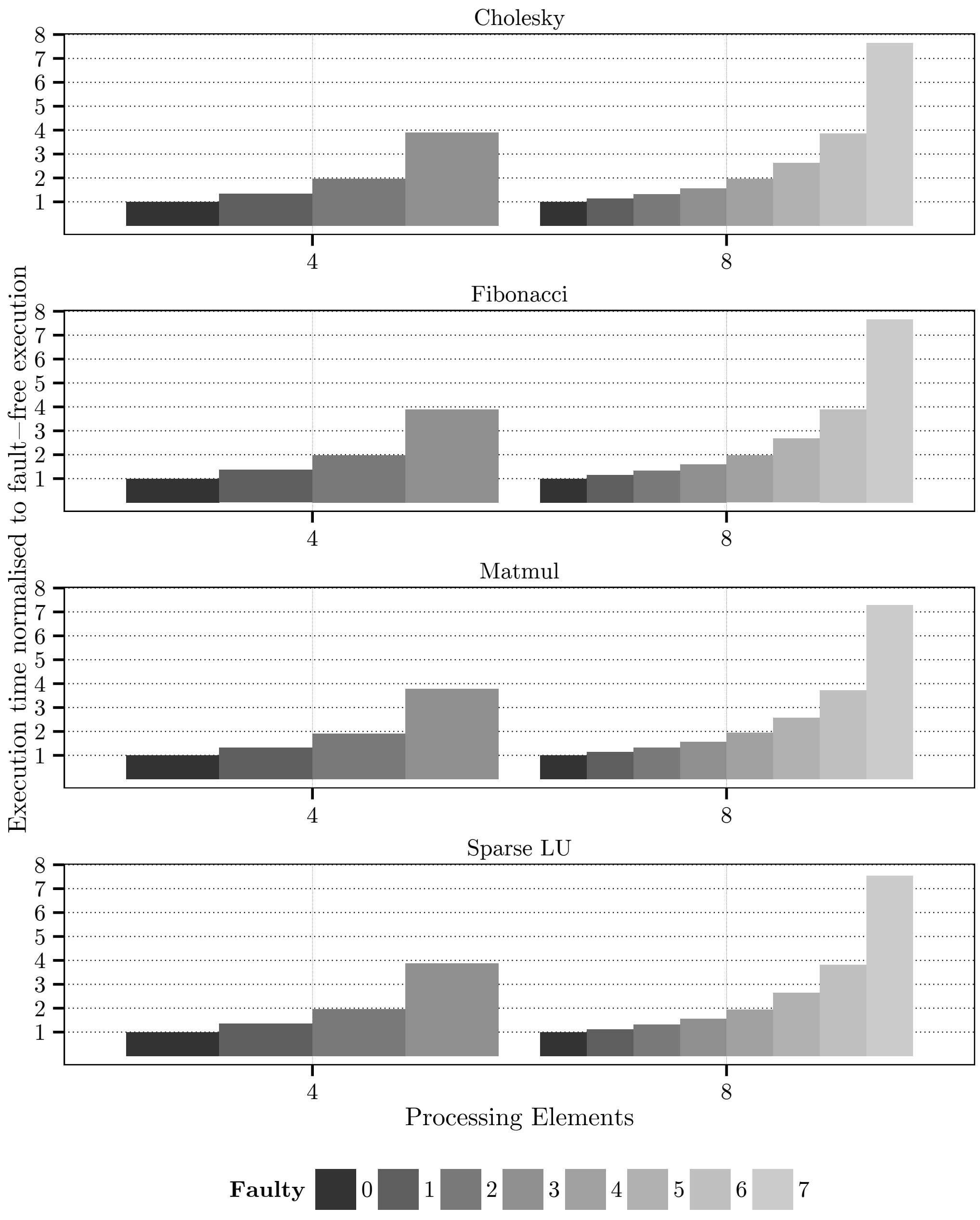


Figure 7.23: Graceful degradation of optimistic double execution, when 0–7 PEs are permanent faulty in the 4 and 8 PE systems (*HighUtil*).

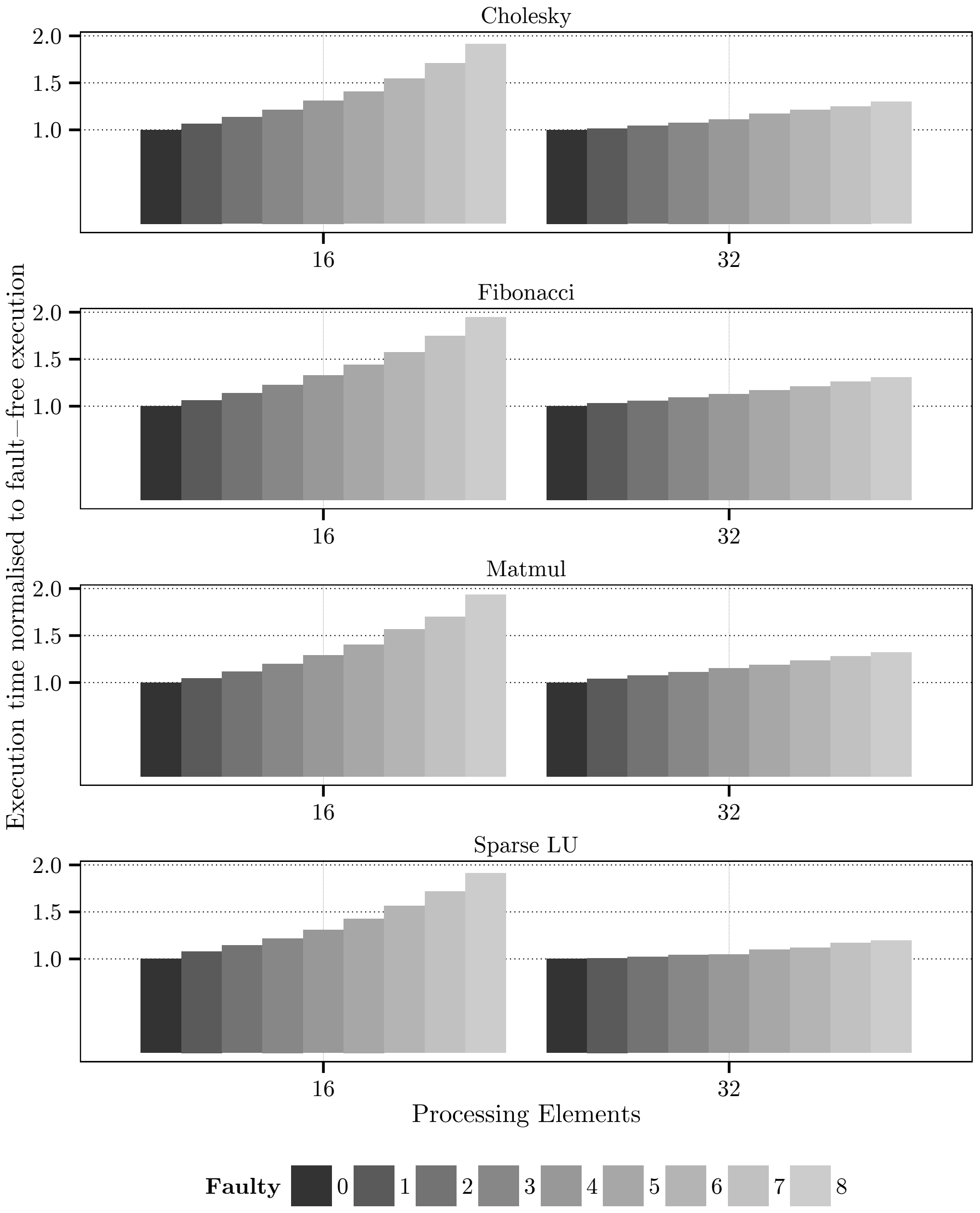


Figure 7.24: Graceful degradation of optimistic double execution, when 0–8 PEs are permanent faulty in the 8 to 16 PE systems (*HighUtil*).

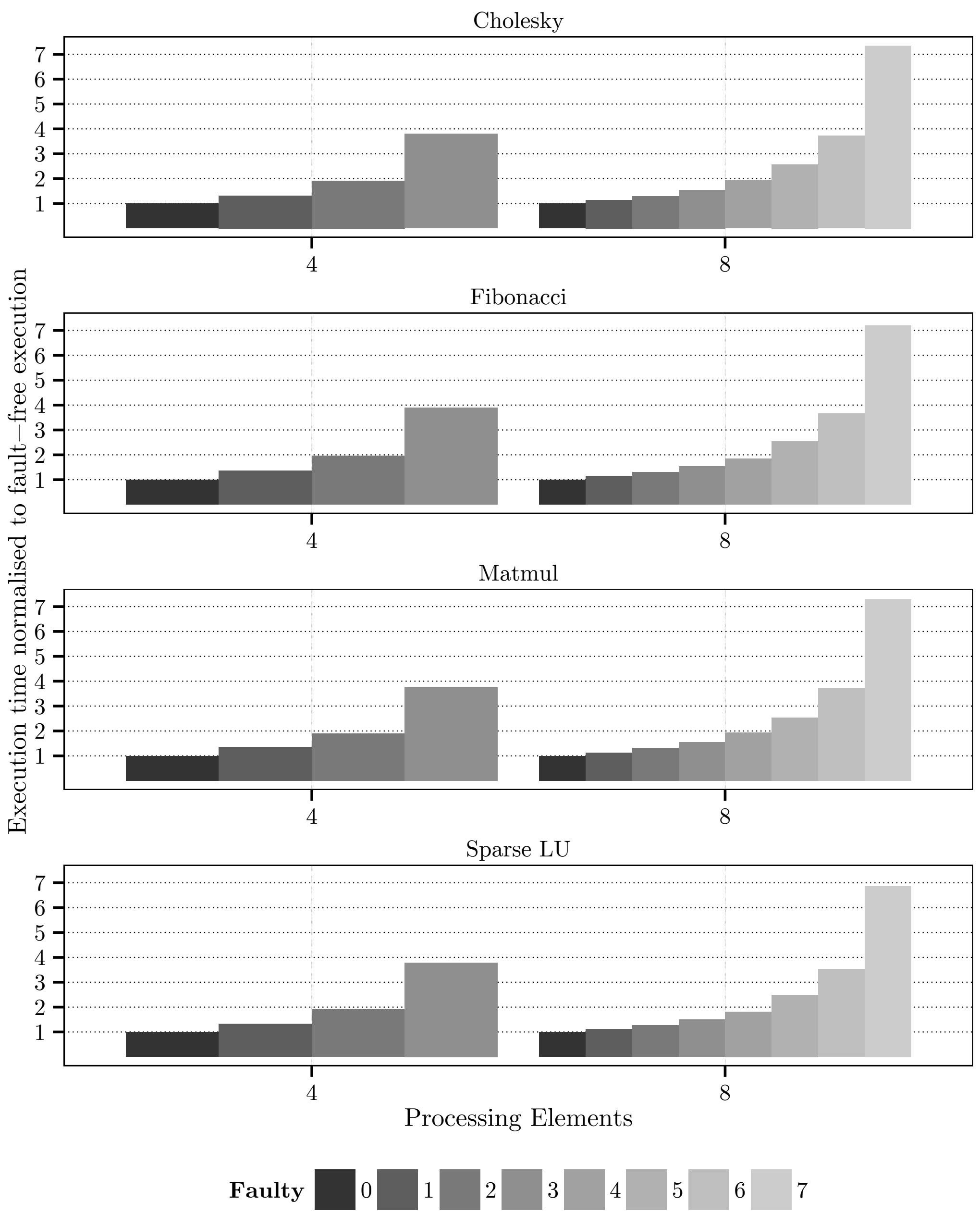


Figure 7.25: Graceful degradation of optimistic double execution, when 0–7 PEs are permanent faulty in the 4 and 8 PE systems (*LowUtil*).

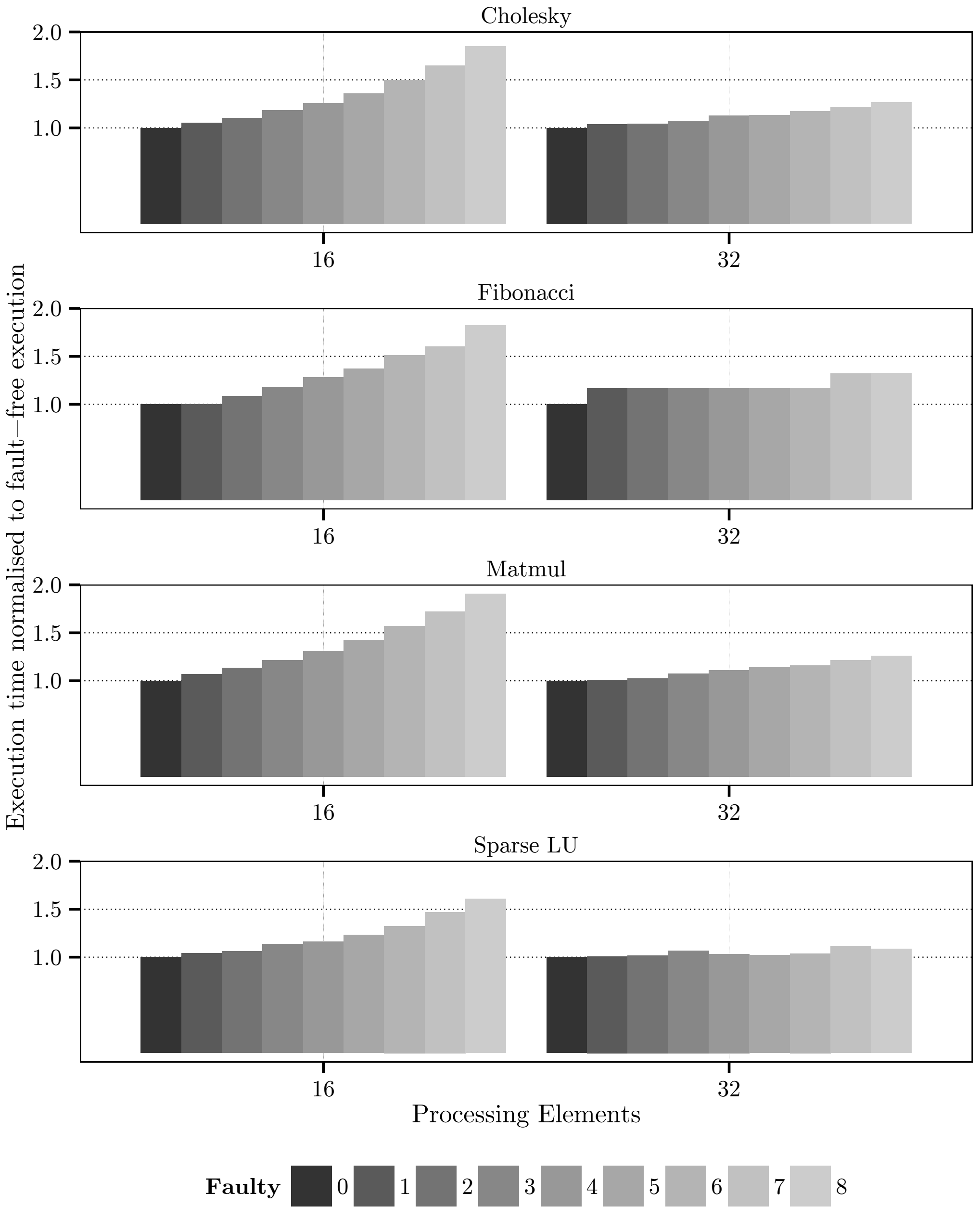


Figure 7.26: Graceful degradation of optimistic double execution, when 0–8 PEs are permanent faulty in the 16 and 32 PE systems (*LowUtil*).

7.3.7 Execution under Transient Faults

Similar to double execution, we also evaluate the performance impact of transient faults on optimistic double execution with global checkpointing. For all simulations, a checkpoint interval of 10 000 ns is used. Errors are injected at a fixed time interval of 10^5 ns. We compare the execution time of optimistic double execution with 5 and 10 faults. We also depict the overhead of the thread restart recovery in case of double execution, which suffered from the same number of faults. The execution time is normalised to the execution time of double execution without faults for all simulated configurations in this subsection. The Figures 7.27 (*HighUtil*) and 7.28 (*LowUtil*) show that global checkpointing has, as expected, a higher impact on the execution time than thread restart recovery. Additionally, it can be seen that in case of the *LowUtil* input set, the overhead of global recovery increases for Matmul and Fibonacci for systems with 16 and 32 PEs.

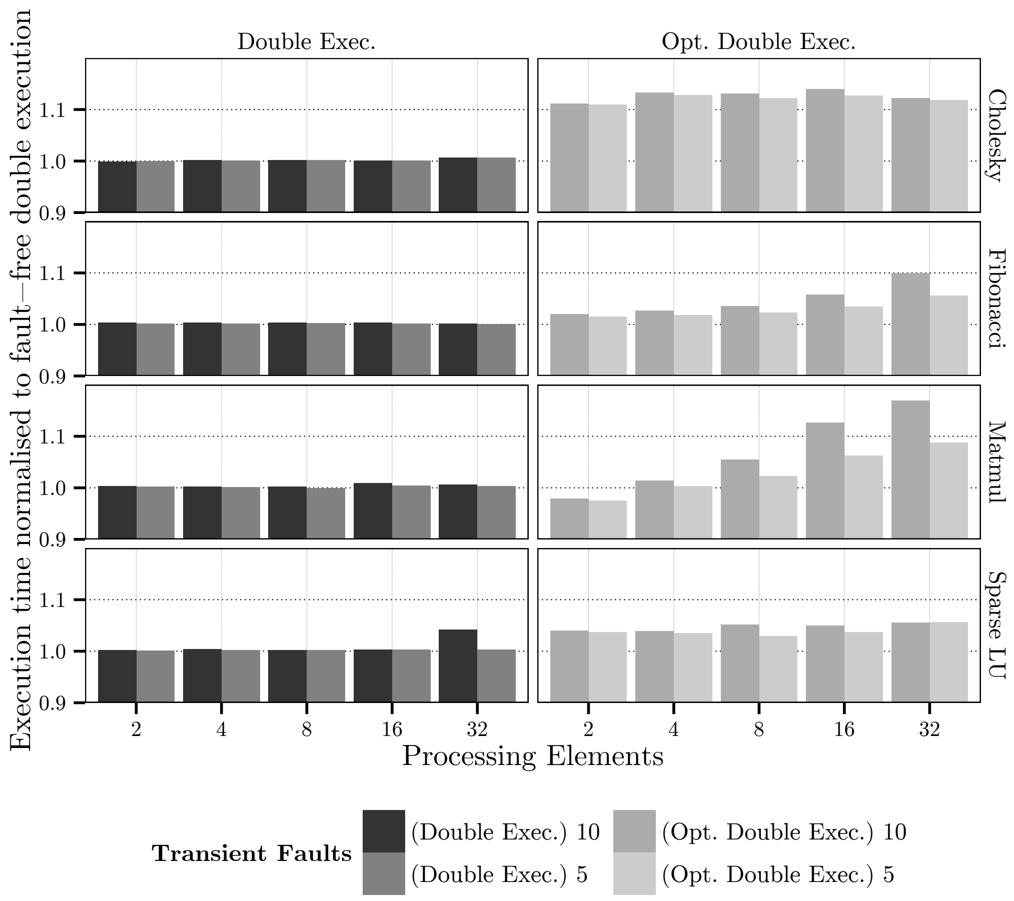
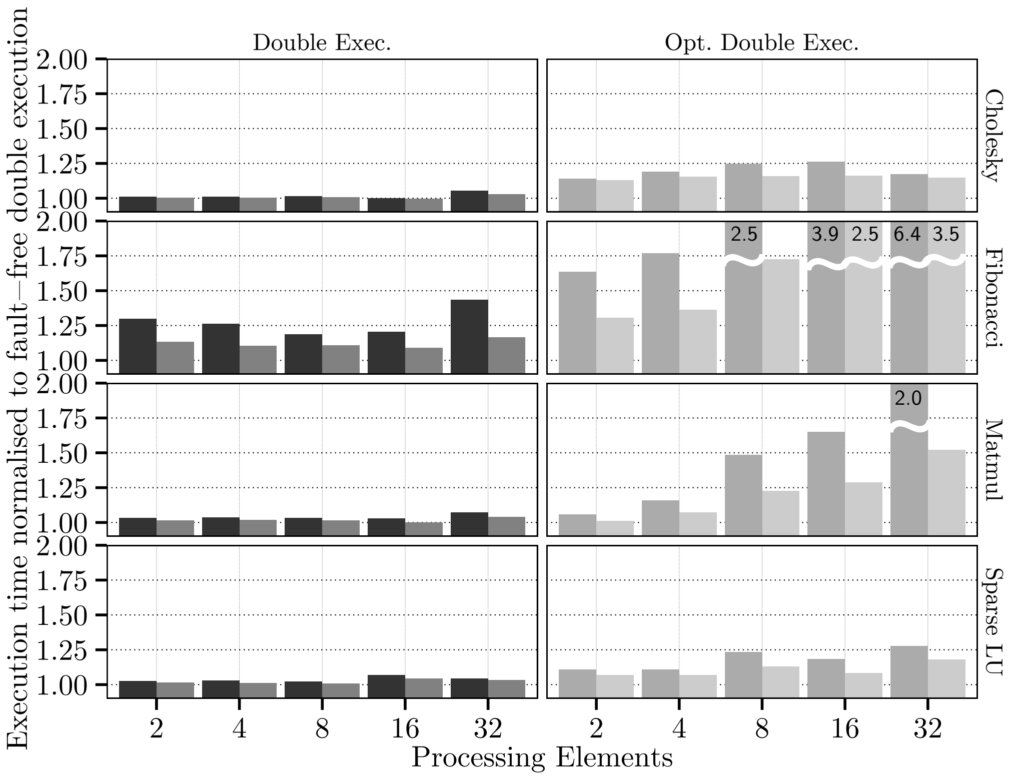


Figure 7.27: Execution time of double execution and optimistic double execution when 5 and 10 transient faults occur (*HighUtil*)



Transient Faults ▇ (Double Exec.) 10 ▇ (Double Exec.) 5 ▇ (Opt. Double Exec.) 10 ▇ (Opt. Double Exec.) 5

Figure 7.28: Execution time of double execution and optimistic double execution when 5 and 10 transient faults occur (*LowUtil*).

7.4 Summary

This section presented evaluation results of double execution, thread restart recovery, optimistic double execution, and data-flow based global checkpointing. The proposed mechanisms were integrated in the COTSon multi-core simulator and evaluated with four data-flow benchmarks.

Compared with an ideal lockstep machine, double execution induces only a small overhead for systems with 16–32 PEs and is faster for the 2–8 PE systems. The impact of the comparison latency on double execution can be significantly reduced by the commit slack of the redundant threads, such that even a high comparison latency of 500 ns shows only a small execution time overhead. Furthermore, the evaluation results confirmed that double execution can not efficiently exploit underutilised PEs to speed up the redundant execution. While this does not lead to a significant overhead for the *HighUtil* benchmarks, the *LowUtil* input set suffers from a significant overhead compared to the ideal lockstep system. It was also shown that double execution can induce a significant commit slack, leading to a high error detection latency in the worst case. Nevertheless, the evaluation also reveals that the commit slack can be decreased for larger systems with more available PEs. The evaluation of the system behaviour under permanent faults confirms that double execution systems are able to gracefully adapt their performance, even when an uneven number of PEs is permanently broken. Furthermore, the evaluation of double execution and thread restart recovery under transient faults reveals that thread restart recovery has only a small impact on the execution time of double execution, even when up to 10 transient faults occur.

The evaluation of optimistic double execution without global checkpoint creation showed that the OWM replication and asynchronous execution of the redundant threads can lead to overhead. In particular for the *HighUtil* input set, where optimistic double execution can not exploit underutilised PEs to speed up the redundant execution, we found a significant overhead compared to double execution. However, for the *LowUtil* input, the execution of optimistic double execution achieves a significant speedup. This performance advantage stems from the higher parallelism of the optimistic double execution. It was further shown that optimistic double execution improves the scalability of the redundant thread execution and is able to speed up the execution over the non-redundant execution by exploiting underutilised PEs. This finding is of particular interest for future parallel systems which may host more PEs than the simulated system configurations used in thesis. However, the simulation of optimistic double execution with global checkpointing also showed that the checkpoint creation overhead reduces the execution time advantages for some benchmarks. Furthermore, the commit slack of optimistic double execution is significant higher compared to double execution and strongly influenced by the checkpoint interval, since all unverified data-flow threads must be compared before a new checkpoint can be established. Compared to the ideal lockstep machine with conventional checkpointing, we still found execution time advantages of optimistic double execution and data-flow based checkpointing for relatively short checkpoint intervals. Additionally, the memory log size can be significantly reduced by the data-flow based global checkpointing mechanism, since not all memory accesses must be incorporated in the log during a checkpoint. Similar to double execution, optimistic double execution is also able to gracefully adjust its performance, when multiple PEs are permanently faulty. However, under the occurrence of transient faults, the overhead of the global recovery mechanism is significantly higher compared to the thread restart recovery.

8 Summary and Future Work

This chapter summarises the results of this thesis and presents ideas on future work on data-flow based fault-tolerant execution.

8.1 Summary

This thesis presented fault-detection, recovery, and diagnosis techniques for a coarse-grained, threaded data-flow execution model to support scalable fault-tolerance in a multi-core architecture. Double execution, the loosely-coupled redundant execution of data-flow threads was integrated in the baseline data-flow execution model and data-flow specific solutions for input replication, output comparison, and redundant thread management were described in detail.

Based on double execution, a thread restart mechanism was proposed, which requires that redundant data-flow threads can only commit, when their results have been compared. Double execution and thread restart recovery were used to implement a fault diagnosis mechanism to localise and identify permanent faulty processing elements in the system. The fault detection unit keeps track of detected PE errors and is able to localise faulty PEs by majority voting. In case of a permanent or intermittent fault, double execution and thread restart recovery support the shutdown of multiple PEs and are able to gracefully reduce the performance of the system, even when multiple PEs must be deactivated.

It was further shown that the data-flow execution principle, leveraged by double execution and thread restart recovery, has several advantages for the implementation of redundant execution, since result comparison is only necessary for data which is consumed by subsequent data-flow threads. Additionally, redundant data-flow threads can be executed completely decoupled from each other, which allows to re-use the data-flow thread scheduler to manage the redundant thread execution. Finally, input replication, output comparison, and the synchronisation between redundant threads is simplified by the data-flow execution principle.

To increase the parallelism of the redundant thread execution, an optimistic variant of double execution was presented, which speculatively allows succeeding data-flow threads to be started before the result comparison has verified the correctness of the thread execution. This allows the leading thread to make forward progress, while the data dependencies between the trailing threads have already been resolved by the leading threads. However, the speculative thread commit requires a mechanism to guarantee input replication for OWM regions. The proposed solution for input replication of the OWM memory uses DMA transfers to copy the OWM sections of the leading thread before the OWM region is required by the trailing thread.

Since optimistic double execution allows committing unverified and potentially erroneous results, the thread restart mechanism can not be used for recovery with optimistic double execution. Therefore, a data-flow based global recovery mechanism was proposed. In case of global checkpointing, the data-flow execution principle reduces the checkpoint creation effort, since the valid execution state of the system is only stored in the main memory and the TSU, while the execution contexts of the PEs can be considered as unsafe, recoverable state. This reduces the checkpointing costs, since for checkpoint creation, only the TSU state and a log, incorporating the thread frame and the OWM region, are required. Compared to checkpointing solutions for shared memory multi-cores, like [Sorin et al., 2002; Prvulovic et al., 2002], this can reduce the checkpointing complexity, since no architectural contexts of the PEs, the cache states, and the coherence protocol must be checkpointed or recovered.

Finally, the evaluation results of the proposed fault-tolerance mechanism showed that double execution has only a small overhead compared to an ideal lockstep system, while the execution is even faster for some system configurations and benchmarks. Particularly, benchmarks, which take advantages of the shared TF and OWM region between the redundant threads achieve a speedup compared to the ideal lockstep execution. Double execution has very low overhead when transient faults occur and is able to gracefully adapt its execution speed, when multiple PEs are deactivated due to permanent faults. However, double execution shows also limited scalability compared to the non-redundant execution, which reduces the exploitation of underutilised PEs. Compared to double execution, optimistic double execution shows a significant overhead for applications with high parallelism, but can speed up the execution of benchmarks, which are not able to fully utilise all PEs. The global checkpointing required for optimistic double execution leads to additional overhead for the periodic checkpoint creation and exposes increasing execution time when transient faults occur. Nevertheless, compared to a conventional checkpointing scheme for a shared memory multi-core, optimistic double execution and data-flow based checkpointing show still reasonable overhead, in particular, for shorter checkpoint intervals.

Finally, we conclude that the data-flow based fault-tolerance mechanisms presented in this thesis support the integration of flexible and scalable redundant execution and checkpointing mechanisms in a parallel architecture. In particular, the loosely-coupled execution of data-flow threads does not require strict cycle-by-cycle synchronisation and therefore allows the use of complex out-of-order cores and shared resources between the redundant execution units. Furthermore, the redundant threads can be executed decoupled from each other, improving the flexible resource sharing in a multi-core architecture. We think that the advantages of the data-flow execution principle for the construction of redundant execution and checkpointing mechanisms can help to solve reliability issues in future massively parallel architectures.

8.2 Future Work

This section discusses possible future research ideas and applications of the data-flow based fault tolerance mechanisms presented in this thesis.

Although fault-tolerance mechanism are state-of-the-art for safety-critical systems, most commercial fault-tolerant techniques in the embedded domain focus on lockstep execution of sequential applications [Infineon Technologies AG, 2015; STMicroelectronics, 2014]. However, future advanced embedded applications, like advanced driver assistance systems (ADAS) or autonomous driving, require high computational performance. While this performance can be usually satisfied by parallel applications and embedded multi-core systems, the systems must also give safety guarantees, such that hardware faults do not lead to a catastrophic behaviour of the system. This means that the fault detection and recovery mechanisms proposed in this thesis may also be used for the implementation of high-performance safety-critical systems. However, functional safety does not only demand fault-tolerant execution, but also predictable timing behaviour of the embedded systems, which would require a worst case execution time (WCET) and schedulability analysis of double execution and thread restart recovery.

The development of high-availability lockstep server systems requires expensive special-purpose processors with complex adaptions to the microarchitecture. This makes the development and the fabrication of high-availability server systems expensive. Therefore, some fault-tolerant systems rely on cheaper software solutions, like virtual machines [VMware, inc., 2009], to implement redundant execution for high-availability systems on general-purpose multi-cores, however, software based fault-tolerance solutions may induce high execution time overhead, limit the fault coverage, and complicate the software development. By contrast, a data-flow based general purpose processor, which implements the fault-tolerance mechanisms described in this thesis, can support parallel, high-performance execution, while the fault-tolerant execution could be activated on demand. This would enable the construction of high-performance systems for both the general-purpose and high-availability domain, while the purpose of the system could be dynamically selected by the customer.

Bibliography

[Ahmed et al., 1990] R. Ahmed, R. Frazier and P. Marinos. 'Cache-aided rollback error recovery (CARER) algorithm for shared-memory multiprocessor systems'. In: *Proceedings of the 20th International Symposium on Fault-Tolerant Computing Systems (FTCS).* 1990, pages 82–88. DOI: 10.1109/FTCS.1990.89338.

[Aggarwal et al., 2007] 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 International Symposium on Computer Architecture (ISCA).* 2007, pages 470–481. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250720.

[Agarwal et al., 2011] R. Agarwal, P. Garg and J. Torrellas. 'Rebound: Scalable Checkpointing for Coherent Shared Memory'. In: *Proceedings of the 38th International Symposium on Computer Architecture (ISCA).* 2011, pages 153–164. ISBN: 978-1-4503-0472-6. DOI: 10.1145/2000064.2000083.

[Arvind et al., 1988] Arvind and R. A. Iannucci. 'Two fundamental issues in multiprocessing'. In: *Parallel Computing in Science and Engineering.* Volume 295. 1988, pages 61–88. ISBN: 978-3-540-18923-7. DOI: 10.1007/3-540-18923-8_15.

[Agrawal et al., 1993] V. Agrawal, C. Kime and K. Saluja. 'A tutorial on built-in self-test. Part 1: Principles'. In: *IEEE Design Test of Computers* 10.1 (1993), pages 73–82. DOI: 10.1109/54.199807.

[Avizienis et al., 1986] A. Avizienis and J.-C. Laprie. 'Dependable computing: From concepts to design diversity'. In: *Proceedings of the IEEE* 74.5 (1986), pages 629–638. DOI: 10.1109/PROC.1986.13527.

[Alves et al., 2014] T. Alves, S. Kundu, L. Marzulo and F. Franca. 'Online error detection and recovery in dataflow execution'. In: *Proceedings of the 20th International On-Line Testing Symposium (IOLTS).* 2014, pages 9–104. DOI: 10.1109/IOLTS.2014.6873679.

[AMD, 2013] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming, Publication No. 24593, Rev. 3.23.* 2013.

[K. Arvind et al., 1990] K. Arvind and R. S. Nikhil. 'Executing a Program on the MIT Tagged-Token Dataflow Architecture'. In: *IEEE Transactions on Computers* 39.3 (1990), pages 300–318. DOI: 10.1109/12.48862.

[Arandi et al., 2011] S. Arandi, C. Kyriacou, G. Michael, G. Mathaios, N. Masrujeh, P. Trancoso and P. Evripidou. *D6.2 - Advanced TERAFLUX Architecture.* Deliverable. 2011. URL: http://www.teraflux.eu/sites/teraflux.eu/files/TERAFLUX-D62-v6.pdf (visited on 30/09/2016).

[Argollo et al., 2009] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero and D. Ortega. 'COTSon: infrastructure for full system simulation'. In: *ACM SIGOPS Operating Systems Review* 43.1 (2009), pages 52–61. DOI: `http://doi.acm.org/10.1145/1496909.1496921`.

[Austin, 1999] T. Austin. 'DIVA: a reliable substrate for deep submicron microarchitecture design'. In: *Proceedings of the 32th International Symposium on Microarchitecture (MICRO)*. 1999, pages 196–207. DOI: `10.1109/MICRO.1999.809458`.

[Avizienis et al., 2004] A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr. 'Basic Concepts and Taxonomy of Dependable and Secure Computing'. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pages 11–33. DOI: `10.1109/TDSC.2004.2`.

[Bernick et al., 2005] D. Bernick, B. Bruckert, P. Vigna, D. Garcia, R. Jardine, J. Klecka and J. Smullen. 'NonStop® Advanced Architecture'. In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 2005, pages 12–21. DOI: `10.1109/DSN.2005.70`.

[Borkar, 2005] S. Borkar. 'Designing Reliable Systems from Unreliable Components: the Challenges of Transistor Variability and Degradation'. In: *IEEE Micro* 25.6 (2005), pages 10–16. DOI: `10.1109/MM.2005.110`.

[Bower et al., 2005] F. Bower, D. Sorin and S. Ozev. 'A mechanism for online diagnosis of hard faults in microprocessors'. In: *Proceedings of the 38th International Symposium on Microarchitecture (MICRO)*. 2005, pages 12–22. DOI: `10.1109/MICRO.2005.8`.

[Constantinescu, 2003] C. Constantinescu. 'Trends and challenges in VLSI circuit reliability'. In: *IEEE Micro* 23.4 (2003), pages 14–19. DOI: `10.1109/MM.2003.1225959`.

[Cummings, 2009] D. Cummings. *Dataflow-Based Rollback Recovery in Distributed and Multi-Core Systems: A Novel Software Approach for Building Highly Reliable Distributed and Multi-Core Systems*. VDM Verlag, 2009. ISBN: 978-3639210194.

[Dell, 1997] T. J. Dell. 'A white paper on the benefits of chipkill-correct ECC for PC server main memory'. In: *IBM Microelectronics Division* (1997), pages 1–23.

[Dennis et al., 1975] J. B. Dennis and D. P. Misunas. 'A preliminary architecture for a basic data-flow processor'. In: *Proceedings of the 2nd International Symposium on Computer Architecture (ISCA)*. 1975, pages 126–132. DOI: `10.1145/642089.642111`.

[Etsion et al., 2010] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta and M. Valero. 'Task Superscalar: An Out-of-Order Task Pipeline'. In: *Proceedings of the 43th International Conference on Microarchitecture (MICRO)*. 2010, pages 89–100. DOI: `10.1109/MICRO.2010.13`.

[Fu et al., 2014] J. Fu, Q. Yang, R. Poss, C. Jesshope and C. Zhang. 'A fault detection mechanism in a Data-flow scheduled Multithreaded processor'. In: *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2014, pages 1–4. DOI: `10.7873/DATE.2014.076`.

[Garbade, 2014] A. Garbade. 'Fehlerlokalisierung in prozessorinternen Kommunikationsnetzen für Vielkern-Prozessoren'. PhD thesis. 2014. URL: `http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:384-opus4-27434` (visited on 30/09/2016).

[Gautier et al., 2007] T. Gautier, X. Besseron and L. Pigeon. 'KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors'. In: *Proceedings of the International Workshop on Parallel Symbolic Computation (PASCO)*. 2007, pages 15–23. ISBN: 978-1-59593-741-4. DOI: `10.1145/1278177.1278182`.

[Giorgi et al., 2014a] R. Giorgi and P. Faraboschi. In: *International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*. 2014, pages 60–65. DOI: `10.1109/SBAC-PADW.2014.30`.

[Giorgi et al., 2014b] R. Giorgi, R. M., F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R., A. Garbade, R. Gayatri, S. Girbal, D. Goodman, B. Khan, S. Koliaï, J. Landwehr, N. Minh, F. Li, M. Lujàn, A. Mendelson, L. Morin, N. Navarro, T. Patejko, A. Pop, P. Trancoso, T. Ungerer, I. Watson, S. Weis, S. Zuckerman and M. Valero. 'TERAFLUX: Harnessing dataflow in next generation teradevices'. In: *Microprocessors and Microsystems* 38.8 (2014), pages 976–990. DOI: `10.1016/j.micpro.2014.04.001`.

[Giorgi, 2012] R. Giorgi. 'TERAFLUX: exploiting dataflow parallelism in teradevices'. In: *Proceedings of the 9th Conference on Computing Frontiers (CF)*. 2012, pages 303–304. ISBN: 978-1-4503-1215-8. DOI: `10.1145/2212908.2212959`.

[Gurd et al., 1985] J. R. Gurd, C. C. Kirkham and I. Watson. 'The Manchester Prototype Dataflow Computer'. In: *Communications of the ACM* 28.1 (1985), pages 34–52. DOI: `10.1145/2465.2468`.

[Gold et al., 2006] B. T. Gold, J. C. Smolens, B. Falsafi, J. C. Hoe and 177958. 'The Granularity of Soft-Error Containment in Shared-Memory Multiprocessors'. In: *Proceedings of the Workshop on System Effects of Logic Soft Errors (SELSE)*. 2006. URL: `http://infoscience.epfl.ch/record/135947` (visited on 30/09/2016).

[Gomaa et al., 2003] M. Gomaa, C. Scarbrough, T. Vijaykumar and I. Pomeranz. 'Transient-fault recovery for chip multiprocessors'. In: *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*. 2003, pages 98–109. DOI: `10.1109/ISCA.2003.1206992`.

[Giorgi et al., 2007] R. Giorgi, Z. Popovic and N. Puzovic. 'DTA-C: A Decoupled Multi-Threaded Architecture for CMP Systems'. In: *Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2007, pages 263–270. ISBN: 978-0-7695-3014-7. DOI: `10.1109/SBAC-PAD.2007.27`.

[Gaudiot et al., 1985] J.-L. Gaudiot and C. Raghavendra. ‘Fault-Tolerance and Data-Flow Systems.’ In: *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. 1985, pages 16–23.

[G. Gupta et al., 2011] G. Gupta and G. S. Sohi. ‘Dataflow execution of sequential imperative programs on multicore architectures’. In: *Proceedings of the 44th International Symposium on Microarchitecture (MICRO)*. 2011, pages 59–70. ISBN: 978-1-4503-1053-6. DOI: 10.1145/2155620.2155628.

[S. Gupta et al., 2008] S. Gupta, S. Feng, A. Ansari, J. Blome and S. Mahlke. ‘The StageNet Fabric for Constructing Resilient Multicore Systems’. In: *Proceedings of the 41st International Symposium on Microarchitecture (MICRO)*. 2008, pages 141–151. ISBN: 978-1-4244-2836-6. DOI: 10.1109/MICRO.2008.4771786.

[Hammond et al., 2004] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis and K. Olukotun. ‘Transactional Memory Coherence and Consistency’. In: *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*. 2004, pages 102–113. ISBN: 0-7695-2143-6. DOI: 10.1145/1028176.1006711.

[Hammarlund et al., 2014] P. Hammarlund, A. Martinez, A. Bajwa, D. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. Osborne, R. Rajwar, R. Singhal, R. D’Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza and T. Burton. ‘Haswell: The Fourth-Generation Intel Core Processor’. In: *IEEE Micro* 34.2 (2014), pages 6–20. DOI: 10.1109/MM.2014.10.

[Haring et al., 2012] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist and C. Kim. ‘The IBM Blue Gene/Q Compute Chip’. In: *IEEE Micro* 32.2 (2012), pages 48–60. DOI: 10.1109/MM.2011.108.

[Hunt et al., 1987] D. B. Hunt and P. N. Marinos. ‘A general purpose cache-aided rollback error recovery (CARER) technique’. In: *Proceedings of the 17th International Symposium on Fault-Tolerant Computing Systems (FTCS)*. 1987, pages 170–175.

[Hum et al., 1995] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryk, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. S. Nemawarkar, P. Panangaden, X. Xue and Y. Zhu. ‘A design study of the EARTH multiprocessor’. In: *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*. 1995, pages 59–68. ISBN: 0-89791-745-6.

[Infineon Technologies AG, 2015] Infineon Technologies AG. *Infineon Aurix - 32-bit Microcontrollers for Automotiv and Industrial Applications*. 2015. URL: http://www.infineon.com/dgdl/Infineon-Tricore+Family+BR_2015-BC-v01_00-EN.pdf?fileId=db3a30431f848401011fc664882a7648 (visited on 30/09/2016).

[ITRS, 2013] ITRS. *International Technology Roadmap for Semiconductors*. Website. Retrieved August 14, 2015 from http://www.itrs.net. 2013.

[Iyer et al., 2005] R. Iyer, N. Nakka, Z. Kalbarczyk and S. Mitra. 'Recent advances and new avenues in hardware-level reliability support'. In: *IEEE Micro* 25.6 (2005), pages 18–29. DOI: 10.1109/MM.2005.119.

[Jafar et al., 2005] S. Jafar, T. Gautier, A. Krings and J.-l. Roch. 'A Checkpoint/Recovery Model for Heterogeneous Dataflow Computations Using Work-Stealing'. In: *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*. Volume 3648. 2005, pages 675–684. DOI: 10.1007/11549468_74.

[Janssens et al., 1994] B. Janssens and W. Fuchs. 'The performance of cache-based error recovery in multiprocessors'. In: *IEEE Transactions on Parallel and Distributed Systems* 5.10 (1994), pages 1033–1043. DOI: 10.1109/71.313120.

[Kavi et al., 2001] K. Kavi, R. Giorgi and J. Arul. 'Scheduled dataflow: execution paradigm, architecture, and performance evaluation'. In: *IEEE Transactions on Computers* 50.8 (2001), pages 834–846. DOI: 10.1109/12.947003.

[Koren et al., 2007] I. Koren and C. M. Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., 2007. ISBN: 978-0120885251.

[Kranitis et al., 2005] N. Kranitis, A. Paschalis, D. Gizopoulos and G. Xenoulis. 'Software-based self-testing of embedded processors'. In: *IEEE Transactions on Computers* 54.4 (2005), pages 461–475. DOI: 10.1109/TC.2005.68.

[Lehtonen et al., 2005] T. Lehtonen, J. Plosila, J. Isoaho et al. *On fault tolerance techniques towards nanoscale circuits and systems*. Turku Centre for Computer Science, 2005. ISBN: 952-12-1596-8.

[LaFrieda et al., 2007] C. LaFrieda, E. Ipek, J. Martinez and R. Manohar. 'Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor'. In: *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN)*. 2007, pages 317–326. DOI: 10.1109/DSN.2007.100.

[Li et al., 2012] F. Li, A. Pop and A. Cohen. 'Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs'. In: *Micro, IEEE* 32.4 (2012), pages 19–31. DOI: 10.1109/MM.2012.49.

[Mukherjee et al., 2002] S. Mukherjee, M. Kontz and S. Reinhardt. 'Detailed design and evaluation of redundant multi-threading alternatives'. In: *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*. 2002, pages 99–110. DOI: 10.1109/ISCA.2002.1003566.

[Mendelson et al., 2000] A. Mendelson and N. Suri. 'Designing high-performance and reliable superscalar architectures-the out of order reliable superscalar (O3RS) approach'. In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 2000, pages 473–481. DOI: 10.1109/ICDSN.2000.857578.

[Mukherjee et al., 2003] 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 International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society. 2003, pages 29–40. DOI: `10.1145/859526.859529`.

[Mukherjee, 2008] S. Mukherjee. *Architecture Design for Soft Errors.* Morgan Kaufmann Publishers Inc., 2008. ISBN: 978-0123695291.

[Najjar et al., 1990] W. Najjar and J.-L. Gaudiot. ‘A data-driven execution paradigm for distributed fault-tolerance’. In: *Proceedings of the 4th ACM SIGOPS European Workshop.* 1990, pages 1–6. DOI: `10.1145/504136.504166`.

[Nguyen-tuong et al., 1996] A. Nguyen-tuong, A. S. Grimshaw and M. Hyett. ‘Exploiting Data-Flow for Fault-Tolerance in a Wide-Area Parallel System’. In: *Proceedings of the 15th International Symposium on Reliable and Distributed Systems (RELDIS).* 1996, pages 2–11. DOI: `10.1109/RELDIS.1996.559687`.

[Portero et al., 2012] A. Portero, A. Scionti, Z. Yu, P. Faraboschi, C. Concatto, L. Carro, A. Garbade, S. Weis, T. Ungerer and R. Giorgi. ‘Simulating the Future kilo-x86-64 core Processors and their Infrastructure’. In: *Proceedings of the 45th Annual Simulation Symposium.* 2012, 9:1–9:7. ISBN: 978-1-61839-784-3.

[Pradhan, 1996] D. K. Pradhan. *Fault-tolerant Computer System Design.* Edited by D. K. Pradhan. Prentice-Hall, Inc., 1996. ISBN: 0-13-057887-8.

[Prvulovic et al., 2002] M. Prvulovic, Z. Zhang and J. Torrellas. ‘ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors’. In: *Proceedings of the 29th International Symposium on Computer Architecture (ISCA).* 2002, pages 111–122. DOI: `10.1109/ISCA.2002.1003567`.

[Rashid et al., 2008] M. Rashid and M. Huang. ‘Supporting highly-decoupled thread-level redundancy for parallel programs’. In: *Proceedings of the 14th International Symposium on High Performance Computer Architecture (HPCA).* 2008, pages 393–404. DOI: `10.1109/HPCA.2008.4658655`.

[Ray et al., 2001] J. Ray, J. Hoe and B. Falsafi. ‘Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery’. In: *Proceedings of the 34th International Symposium on Microarchitecture (MICRO).* 2001, pages 214–224. ISBN: 0-7695-1369-7.

[Reinhardt et al., 2000] S. K. Reinhardt and S. Mukherjee. ‘Transient Fault Detection via Simultaneous Multithreading’. In: *Proceedings of the 27th International Symposium on Computer Architecture (ISCA).* 2000, pages 25–36. DOI: `10.1145/339647.339652`.

[Rotenberg, 1999] E. Rotenberg. ‘AR-SMT: a microarchitectural approach to fault tolerance in microprocessors’. In: *Proceedings of the 29th International Symposium on Fault-Tolerant Computing.* 1999, pages 84–91. DOI: `10.1109/FTCS.1999.781037`.

[Robic et al., 2000] B. Robic, J. Silc and T. Ungerer. ‘Beyond dataflow’. In: *Journal of Computing and Information Technology* 8.2 (2000), pages 89–102. DOI: `10.2498/cit.2000.02.01`.

[Sánchez et al., 2009] D. Sánchez, J. Aragón and J. Garcıa. 'Extending SRT for parallel applications in tiled-CMP architectures'. In: *Proceedings of the International Symposium on Parallel Distributed Processing (IPDPS)*. 2009, pages 1–8. DOI: `10.1109/IPDPS.2009.5160902`.

[Sánchez et al., 2010] D. Sánchez, J. Aragón and J. Garcıa. 'A log-based redundant architecture for reliable parallel computation'. In: *Proceedings of the International Conference on High Performance Computing (HiPC)*. 2010, pages 1–10. DOI: `10.1109/HIPC.2010.5713183`.

[Stavrou et al., 2005] K. Stavrou, P. Evripidou and P. Trancoso. 'DDM-CMP: Data-Driven Multithreading on a Chip Multiprocessor'. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Volume 3553. 2005, pages 364–373. ISBN: 978-3-540-26969-4. DOI: `10.1007/11512622_39`.

[Shivakumar et al., 2002] 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: *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 2002, pages 389–398. DOI: `10.1109/DSN.2002.1028924`.

[Slegel et al., 1999] 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'. In: *IEEE Micro* 19.2 (1999), pages 12–23. DOI: `10.1109/40.755464`.

[Smolens et al., 2004] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe and A. G. Nowatzyk. 'Fingerprinting: bounding soft-error detection latency and bandwidth'. In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2004, pages 224–234. ISBN: 1-58113-804-0. DOI: `10.1145/1024393.1024420`.

[Smolens et al., 2005] J. C. Smolens, J. Kim, J. Hoe and B. Falsafi. 'Understanding the performance of concurrent error detecting superscalar microarchitectures'. In: *Proceedings of the 5th International Symposium on Signal Processing and Information Technology (ISSPIT)*. 2005, pages 13–18. DOI: `10.1109/ISSPIT.2005.1577062`.

[Smolens et al., 2006] J. C. Smolens, B. T. Gold, B. Falsafi and J. C. Hoe. 'Reunion: Complexity-Effective Multicore Redundancy'. In: *Proceedings of the 39th International Symposium on Microarchitecture (MICRO)*. 2006, pages 223–234. ISBN: 0-7695-2732-9. DOI: `10.1109/MICRO.2006.42`.

[Sorin et al., 2002] D. J. Sorin, M. M. K. Martin, M. D. Hill and D. A. W. J. 'SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery'. In: *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*. 2002, pages 123–134. DOI: `10.1109/ISCA.2002.1003568`.

[Sorin, 2009] D. J. Sorin. *Fault Tolerant Computer Architecture*. Morgan and Claypool Publishers, 2009. ISBN: 978-1598299533.

[Sundaramoorthy et al., 2000] K. Sundaramoorthy, Z. Purser and E. Rotenberg. 'Slipstream processors: improving both performance and fault tolerance'. In: *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* 2000, pages 257–268. DOI: `10.1145/378993.379247`.

[Srinivasan et al., 2004] J. Srinivasan, S. Adve, P. Bose and J. Rivers. 'The impact of technology scaling on lifetime reliability'. In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN).* 2004, pages 177–186. DOI: `10.1109/DSN.2004.1311888`.

[Siewiorek et al., 1998] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems (3rd Ed.): Design and Evaluation.* A. K. Peters, Ltd., 1998. ISBN: 1-56881-092-X.

[STMicroelectronics, 2014] STMicroelectronics. *SPC56EL60x, SPC56EL54x, SPC564L60x, SPC564L54x - 32-bit Power Architecture® microcontroller for automotive SIL3/ASILD chassis and safety applications.* datasheet DocID15457 Rev 12. 2014. URL: `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00070691.pdf` (visited on 30/09/2016).

[Tullsen et al., 1995] D. Tullsen, S. Eggers and H. Levy. 'Simultaneous multithreading: Maximizing on-chip parallelism'. In: *Proceedings of 22nd International Symposium on Computer Architecture (ISCA).* 1995, pages 392–403.

[Timor et al., 2010] A. Timor, A. Mendelson, Y. Birk and N. Suri. 'Using Underutilized CPU Resources to Enhance Its Reliability'. In: *IEEE Transactions on Dependable and Secure Computing* 7.1 (2010), pages 94–109. DOI: `10.1109/TDSC.2008.31`.

[VMware, inc., 2009] VMware, inc. *VMware vSphere™ 4 Faul Tolerance: Architecture and Performance.* White Paper Revision: 20090811. 2009. URL: `https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/perf-vsphere-fault_tolerance.pdf` (visited on 30/09/2016).

[Vijaykumar et al., 2002] T. Vijaykumar, I. Pomeranz and K. Cheng. 'Transient-fault recovery using simultaneous multithreading'. In: *Proceedings of the 29th International Symposium on Computer Architecture (ISCA).* 2002, pages 87–98. DOI: `10.1109/ISCA.2002.1003565`.

[Weis et al., 2011a] S. Weis, A. Garbade, J. Wolf, B. Fechner, A. Mendelson, R. Giorgi and T. Ungerer. 'A Fault Detection and Recovery Architecture for a Teradevice Dataflow System'. In: *First Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM).* 2011, pages 38–44. DOI: `10.1109/DFM.2011.9`.

[Weis et al., 2011b] S. Weis, A. Garbade, S. Schlingmann and T. Ungerer. 'Towards Fault Detection Units as an Autonomous Fault Detection Approach for Future Many-Cores'. In: *ARCS 2011 Workshop Proceedings.* 2011, pages 20–23. ISBN: 978-3-8007-3333-0.

[Weis et al., 2016] S. Weis, A. Garbade, B. Fechner, A. Mendelson, R. Giorgi and T. Ungerer. 'Architectural Support for Fault Tolerance in a Teradevice Dataflow System'. In: *International Journal of Parallel Programming* 44.2 (2016), pages 208–232. DOI: `10.1007/s10766-014-0312-y`.

[Wu et al., 1990] K. L. Wu, W. K. Fuchs and J. H. Patel. 'Error Recovery in Shared Memory Multiprocessors Using Private Caches'. In: *IEEE Transactions on Parallel and Distributed Systems* 1.2 (1990), pages 231–240. DOI: `10.1109/71.80134`.

[Wittenbrink et al., 2011] C. Wittenbrink, E. Kilgariff and A. Prabhu. 'Fermi GF100 GPU Architecture'. In: *IEEE Micro* 31.2 (2011), pages 50–59. DOI: `10.1109/MM.2011.24`.

[Yazdanpanah et al., 2013] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez and Y. Etsion. 'Hybrid Dataflow/Von-Neumann Architectures'. In: *IEEE Transactions on Parallel and Distributed Systems* 25.6 (2013), pages 1489–1509. DOI: `10.1109/TPDS.2013.125`.

[Yeh, 1996] Y. Yeh. 'Triple-triple redundant 777 primary flight computer'. In: *Proceedings of the Aerospace Applications Conference.* Volume 1. 1996, pages 293–307. DOI: `10.1109/AERO.1996.495891`.

[Yalcin et al., 2013] G. Yalcin, O. S. Unsal and A. Cristal. 'Fault Tolerance for Multi-Threaded Applications by Leveraging Hardware Transactional Memory'. In: *Proceedings of the International Conference on Computing Frontiers.* 2013, 4:1–4:9. DOI: `10.1145/2482767.2482773`.

[Zuckerman et al., 2011] S. Zuckerman, J. Suetterlein, R. Knauerhase and G. R. Gao. 'Using a "codelet" program execution model for exascale machines: position paper'. In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era.* 2011, pages 64–69. ISBN: 978-1-4503-0708-6. DOI: `http://doi.acm.org/10.1145/2000417.2000424`.

List of Figures

List of Tables

List of Algorithms