An Adaptive Multi-Core Architecture for Runtime-Configurable Redundancy in Mixed-Criticality Systems

Zur Erlangung des akademischen Grades eines

DOKTORS DER INGENIEURWISSENSCHAFTEN (Dr.-Ing.)

von der KIT-Fakultät für
Elektrotechnik und Informationstechnik
des Karlsruher Instituts für Technologie (KIT)

angenommene

DISSERTATION

von

M. Sc. Fabian E.-W. Kempf
geb. in Stuttgart

Tag der mündlichen Prüfung:	21.06.2024
Hauptreferent:	Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferentin:	Prof. Dr.-Ing. Diana Göhringer

An Adaptive Multi-Core Architecture for Runtime-Configurable Redundancy in Mixed-Criticality Systems
First edition: April 2025

Abstract

Until today, Moore's Law has persevered. This continual advancement in technology permits a consistent integration of transistors within the same silicon footprint, resulting in the development of increasingly complex systems. For computational systems, this entails a surge in processor units, ultimately leading to the raise of multi-core processors. Although multi-core systems were omnipresent in consumer electronics, High-Performance Computing (HPC), and data warehousing, they have progressively become relevant for embedded systems. The raise of multi-core processors in the embedded domain has enhanced functional integration density within the same processor, consolidating functionalities of varying criticality levels onto a common hardware platform.

In the field of safety-critical applications, these systems must adhere to international standards such as IEC 61508 (Electrical/ Electronic/ Programmable Systems), DO-254 (avionic), and ISO 26262 (automotive), which outline the safety requirements to be met. ISO 26262 introduces the concept of Automotive Safety Integrity Level (ASIL). To fulfill ASIL and functional safety prerequisites, safety-critical applications must, at the very least, be fail-safe.

In the automotive industry, given the ongoing trend towards ever-increasing integration and centralization, it is foreseeable that functionalities of different criticality will be executed on a single control unit. This means that in the future, safety-critical applications such as

Advanced Driver Assistance Systems (ADAS), classified as ASIL-B, as well as highly safety-critical functions such as electric power steering control or the Anti-lock Braking System (ABS), each rated ASIL-D, can be executed alongside non-critical functions such as entertainment applications on the same processor.

Current systems employ a static design that limits the computational performance of non-critical applications to meet the safety requirements of critical applications. However, this design approach is inadequate for Mixed Criticality System (MCS) applications, especially in systems utilizing graceful degradation to ensure fail operation. In such cases, non-critical applications may be replaced by critical ones. Therefore, the MCS platform must provide the necessary functional safety for the application.

Taking inspiration from the concept of Adaptive Fault Tolerance (AFT), this thesis explores the concept of Adaptive Redundancy (AR) and its application to multi-core architectures. Runtime AR empowers systems to adapt their redundancy on the fly based on the safety requirements of the executed function. To realize the AR concept for MCS, two novel adaptive processor architectures are introduced: the Adaptive Lockstep Processor (ALP), which employs a fine-grained safety mechanisms, and the Adaptive Cache Checkpointing (ACCP) processor, which utilizes a coarse-grained approach to ensure system safety.

In the ALP, the processor cores provide the flexibility to reconfigure between split and lockstep execution modes during runtime. In split mode, cores independently execute program code, while in lockstep configuration, each instruction is redundantly processed on different processor cores, with results compared to detect faults. Detected faults trigger an automatic fault handling with the ability to recover from transient faults. The reconfiguration process from independent processor cores to a logical lockstep core and vice versa is entirely abstracted from the executed applications, managed by the digital hardware architecture.

The second adaptive processor architecture based on ACCP, focuses on a coarse-grained safety mechanism at the cache level. Similar to the ALP, the redundant ACCP is activated on-demand. The hardware autonomously generates checkpoints and detects faults during program code execution. Whenever a flaw is detected, the system reverts to the last checkpoint.

However, implementing ACCP involves time-consuming state replication, during which the processor cores must stall. To mitigate this stall time, an interleaved state transfer method is introduced, reducing the stall time of processor cores required for state replication. The processor state transfer occurs in parallel with regular program execution.

Zusammenfassung

Bis heute hat sich das Mooresche Gesetz bewahrheitet. Der kontinuierliche technologische Fortschritt ermöglicht eine konsequente Steigerung der Integration von Transistoren auf der gleichen Siliziumfläche, was zur Entwicklung immer komplexerer Systeme führt. Für Rechensysteme bedeutet dies einen Anstieg der Recheneinheiten, was schließlich zum Aufkommen von Multicore-Prozessoren geführt hat. Obwohl Multicore-Systeme in der Unterhaltungselektronik, im High-Performance Computing (HPC) und im Data Warehousing schon lange üblich sind, werden diese zunehmend auch für eingebettete Systeme relevant. Die Zunahme von Multicore-Prozessoren im eingebetteten Bereich hat die Dichte der Funktionsintegration innerhalb desselben Prozessors erhöht. Dadurch können Funktionen mit unterschiedlichen Kritikalitätsgraden auf einer gemeinsamen Hardwareplattform zusammengeführt werden.

Im Bereich von sicherheitskritischen Anwendungen müssen diese Systeme internationalen Normen wie IEC 61508 (elektrische/ elektronische/ programmierbare Systeme), DO-254 (Avionik) und ISO 26262 (Automotive) entsprechen, welche die zu erfüllenden Sicherheitsanforderungen umreißen. ISO 26262 führt das Konzept des Automotive Safety Integrity Level (ASIL) ein. Um ASIL und die Anforderungen an die funktionale Sicherheit zu erfüllen, müssen sicherheitskritische Anwendungen zumindest ausfallsicher sein.

In der Automobilindustrie ist es angesichts des anhaltenden Trends zur immer weiteren Integration und Zentralisierung absehbar, dass Funktionalitäten unterschiedlicher Kritikalität auf einem Steuergerät ausgeführt werden. Dies bedeutet, dass in Zukunft sicherheitsrelevante Anwendungen wie Fahrerassistenzsysteme (Advanced Driver Assistance Systems, ADAS), die mit ASIL-B klassifiziert sind, sowie höchst sicherheitskritische Funktionen wie die Steuerung der elektrischen Servolenkung oder das Antiblockiersystem (ABS), jeweils bewertet mit ASIL-D, neben unkritischen Funktionen wie beispielsweise Unterhaltungsanwendungen auf demselben Prozessor ausgeführt werden können.

Derzeitige Systeme verwenden ein statisches Design, das die Rechenleistung unkritischer Anwendungen begrenzt, um die Sicherheitsanforderungen kritischer Anwendungen zu erfüllen. Dieser Entwurfsansatz ist jedoch für Anwendungen mit gemischter Kritikalität (engl. Mixed Criticality System, MCS) unzureichend, insbesondere bei Systemen, die mit Graceful Degradation arbeiten, um einen störungsfreien Betrieb zu gewährleisten. In solchen Fällen können unkritische Anwendungen durch kritische Anwendungen ersetzt werden. Daher muss die MCS-Plattform die erforderliche funktionale Sicherheit für die Anwendung bereitstellen.

In Anlehnung an das Konzept der adaptiven Fehlertoleranz (engl. Adaptive Fault Tolerance, AFT) wird in dieser Arbeit das Konzept der adaptiven Redundanz (engl. Adaptive Redundancy, AR) und die Anwendung des Konzepts auf Multicore-Architekturen untersucht. Laufzeit-AR befähigt Systeme, ihre Redundanz im laufenden Betrieb auf der Grundlage der Sicherheitsanforderungen der ausgeführten Funktion anzupassen. Um das AR-Konzept für MCS zu realisieren, werden zwei neuartige adaptive Prozessorarchitekturen vorgestellt: der Adaptive Lockstep Processor (ALP), der feinkörnige Sicherheitsmechanismen einsetzt, und der Adaptive Cache Checkpointing (ACCP) Processor, der einen grobkörnigen Ansatz zur Gewährleistung der Systemsicherheit verwendet.

Im ALP bieten die Prozessorkerne die Flexibilität, während der Laufzeit zwischen Split- und Lockstep-Ausführungsmodi umzuschalten. Im Split-Modus führen die Kerne den Programmcode unabhängig voneinander aus, während in der Lockstep-Konfiguration jede Anweisung redundant auf verschiedenen Prozessorkernen verarbeitet wird, wobei die Ergebnisse jeder Instruktion zur Fehlererkennung verglichen werden. Erkannte Fehler lösen eine automatische Fehlerbehandlung, mit der Fähigkeit zur Wiederherstellung von transienten Fehlern, aus. Der Rekonfigurationsprozess von unabhängigen Prozessorkernen zu einem logischen Lockstep-Kern und umgekehrt ist vollständig von den ausgeführten Anwendungen abstrahiert und wird von der digitalen Hardwarearchitektur verwaltet.

Die zweite adaptive Prozessorarchitektur, die auf ACCP basiert, konzentriert sich auf einen grobkörnigen Sicherheitsmechanismus auf Cache-Ebene. Ähnlich wie der ALP wird die Redundaz des ACCP-Prozessors bei Bedarf aktiviert. Dieser erzeugt selbstständig Checkpoints und erkennt Fehler während der Programmcodeausführung. Sobald ein Fehler entdeckt wird, kehrt das System zum letzten gültigen Checkpoint zurück.

Die Implementierung des ACCP-Prozessors erfordert jedoch eine zeitaufwendige Zustandsreplikation, bei der die Prozessorkerne im Leerlauf arbeiten müssen. Um diese Zeit zu verkürzen, wird eine verschachtelte Zustandsübertragungsmethode eingeführt, welche die für die Zustandsreplikation erforderliche Zeit des Leerlaufs der Prozessorkerne reduziert. Die Übertragung des Prozessorstatuses erfolgt hierbei parallel zur regulären Programmausführung.

Preface

This dissertation originates from my time at the Karlsruhe Institute of Technology (KIT), where I had the privilege of working as a researcher at the Institute for Information Processing Technologies (ITIV). My time at KIT was both academically enriching and personally rewarding, and it has shaped me profoundly, both as a researcher and as an individual.

First and foremost, I would like to express sincere gratitude to my supervisor, Prof. Jürgen Becker. His guidance, trust, and continuous support allowed me to explore a wide range of exciting projects, from fundamental to applied research. This dissertation would not have been possible without his encouragement and the academic freedom he provided throughout this journey.

I am also sincerely thankful to Prof. Diana Göhringer from the Technical University of Dresden (TUD) for serving as the second examiner of this thesis.

A significant portion of this dissertation was developed within the ARoMA project. I would like to thank everyone involved in this project. In particular, I am grateful to Christoph Kühbacher, Christian Mellwig, and Prof. Theo Ungerer for the engaging discussions and productive collaboration, which had a lasting impact on the direction and content of this dissertation.

I also owe special thanks to my colleagues at ITIV. Your academic and non-academic conversations were a constant source of motivation

and inspiration. Rather than risk leaving someone out, let me extend my appreciation to all of you. The collegial and supportive atmosphere made my time at ITIV not only professionally meaningful but personally enjoyable. I truly valued every shared experience, whether at the institute, during conferences and business trips, or throughout the many informal moments we enjoyed together beyond work.

To my friends, thank you for your encouragement, your patience, and your belief in me. Your support meant more than I can put into words.

Last but by no means least, I want to thank my parents, my brother, and my entire family for their unwavering support throughout the years. Your encouragement has been a constant foundation in my life.

And finally, to my girlfriend Jana — thank you for your endless support, your patience, and your faith in me throughout this journey. Without you all, this work would not have been possible.

Fabian Kempf
April 2025

Contents

Chapter 1

Introduction

It is nearly impossible to envision a world without electronic devices, given their omnipresence and widespread usage. From vast warehouses and High-Performance Computing (HPC) centers to the tiniest embedded devices, the foundation of their existence lies in the steady advancement of semiconductor technology. The countless electronic devices that have become integral to our daily lives owe their existence and functionality to the continuous evolution of semiconductor technology.

Semiconductor technology has consistently advanced, maintaining the validity of Moore's Law to the present day. In 1965, Gordon Moore foresaw a doubling of the number of transistors in a dense integrated circuit every 18 to 24 months [1]. As illustrated in Figure 1.1, the semiconductor industry has successfully fulfilled this prediction up to the present days.

The historical increase in the number of transistors has traditionally been utilized to enhance the performance of integrated circuits. For several decades, the increment of operational frequency, driven by technological improvements, significantly boosted overall system performance. However, this frequency increase came at the cost of higher power consumption and increased thermal heat. The dissipation of thermal energy emerged as a limiting factor for modern integrated circuits.

The total power consumption, denoted as P_{total}, in a digital integrated circuit can be described as follows:

$$P_{total} = P_{stat} + P_{dyn}. \tag{1.1}$$

The power consumption of an integrated circuit is composed of two primary components: static power (P_{stat}) and dynamic power dissipation (P_{dyn}) [2; 3; 4]. Static power dissipation is primarily influenced by leakage and remains independent of workload. It is determined by factors such as leakage current and the supply voltage.

The dynamic power term can be expressed as follows:

$$P_{dyn} \sim C_{load} + V_{dd}^2 * f. \tag{1.2}$$

While static power remains unaffected by external factors, dynamic power is directly proportional to the operational frequency (f). In each clock cycle, the integrated circuit's capacitance (C_{load}) undergoes charging or discharging, a physical process that demands energy and power. Beyond capacitance and frequency, the supply voltage (V_{dd}) plays a essential role. However, scaling the supply voltage encounters inherent physical limitations.

The operational frequency of an integrated circuit has reached a saturation point, primarily constrained by limitations in heat dissipation and the proportional relationship with power consumption. This saturation is evident in both typical power consumption and the fre-

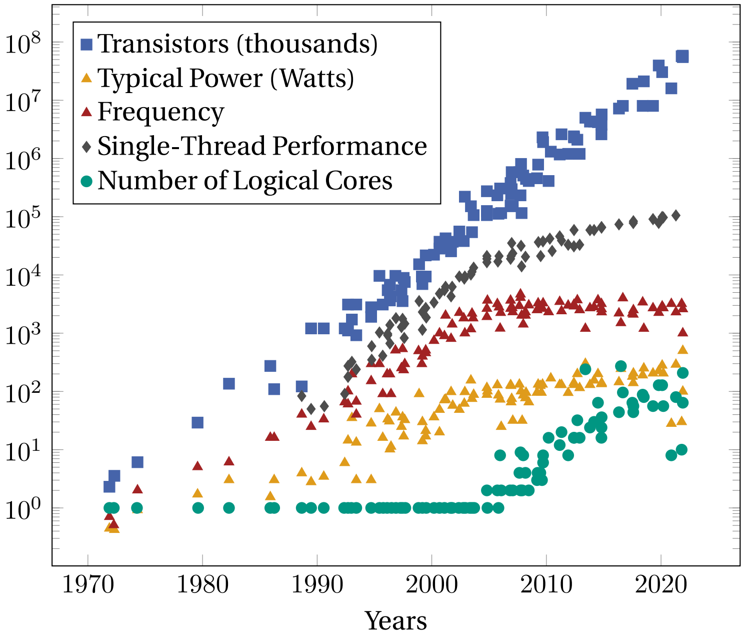


Figure 1.1: 50 Years of microprocessor trend. The data is taken from [5].

quency graph for real Commercial off-the-Shelf (COTS) devices, as illustrated in Figure 1.1.

To enhance the computational power of digital integrated circuits beyond these saturation points, the focus has shifted towards architectural improvements. However, boosting the performance of a single core has become increasingly challenging. Pollack's rule, which asserts that "performance increases roughly proportional to the square root of the increase in complexity" [6], still holds true. This rule remains applicable as the single-thread performance experiences gradual improvement due to new microarchitectures, as illustrated in Figure 1.1.

The physical limitations, including the inability to further increase the frequency of digital integrated circuits due to power density constraints and Pollack's rule, make achieving significant performance improvements for a single core highly challenging.

Instead of persistently focusing on enhancing the microarchitectures of individual processor cores, there has been a shift towards increasing the number of processor cores. Figure 1.1 illustrates the trend of increasing core counts after reaching a point where further major improvements were unattainable through frequency scaling. The integration of multiple cores in a single processor design is termed a multi-core architecture.

The multi-core architecture aims to improve performance through the parallelization of applications. Rather than executing a problem on a single core, the problem formulation is distributed across the many cores of the multi-core system. Amdahl's law describes how a workload or application can benefit from multiple execution units, such as processor cores [7]. The law quantifies the speedup achieved through parallel execution, providing a framework to describe the execution time of each workload.

$$T = t_S + t_P \tag{1.3}$$

The execution time, denoted as T, consists of two components: the sequential program aspects (t_S) and the parallelizable portion (t_P). It is important to note that only the parallelizable portion (t_P) of the application can benefits of multiple execution units. The speedup (S) is thus expressed as follows:

$$S = \frac{T}{t_S + \frac{t_p}{n_p}} \leq \frac{T}{t_S} = \frac{1}{1 - t_P} \tag{1.4}$$

The speedup (S) is primarily dependent on the parallelizable time (t_P) of the workload and the number of parallel execution units (n_p). The maximum achievable speedup is determined by the sequential part (t_S) of the workload, setting an upper limit.

A visual representation of Amdahl's law in Figure 1.2 illustrates the speedup of various workloads with varying degrees of parallelization. It demonstrates that beyond a certain number of cores, increasing the core count does not significantly enhance the speedup of a workload or functionality; instead, the speedup saturates. Multi-core systems stands out in improving latency speedup until a certain threshold is reached, at which point the sequential program section begins to dominate. To optimize the utilization of parallel systems, such as multi-core processors, it is more efficient to deploy multiple and diverse functionalities simultaneously on a shared system rather than executing a single workload or functionality.

1.1 Motivation

Amdahl's law demonstrates the inherent limitations of parallelization within a program's sequential components. While an increase in processor cores can enhance application performance, the contemporary trend in embedded multi-core systems extends beyond single-task execution. These systems integrate diverse functionalities into a single processor, each associated with varying levels of functional safety criticalities. Notably, the automotive industry is currently centralizing Electronic/Electrical (E/E) architecture, reducing Electronic Control Units (ECUs). This reduction leads to the integration of functions with distinct safety requirements, resulting in Mixed Criticality System (MCS) [8].

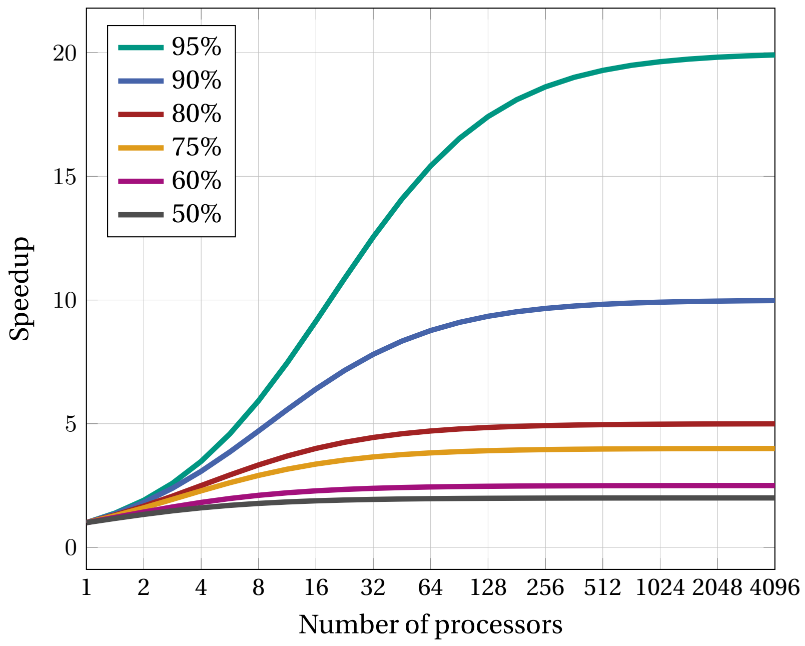


Figure 1.2: Speedup in accordance to Amdahl's law for different parallel portions. For example 90% parallel portion means that 90% of the program could be parallelized and 10% is the sequential part.

MCS research emphasizes scheduling challenges, with a key focus on ensuring the Worst-Case Execution Time (WCET) of critical tasks. However, existing research often overlooks and neglected the functional safety requirements inherent in such applications.

Safety-critical applications must conform to international standards such as IEC 61508, DO-254, and ISO 26262, which define safety such as Safety Integrity Level (SIL) and Automotive Safety Integrity Level (ASIL). These applications must, at a minimum, be fail-safe and should not cause any harm in the case of a failure.

When concurrently running highly safety-critical and non-critical applications on the same hardware, the functional safety requirements for the safety-critical application must be respected. Anticipating the future, scenarios where Advanced Driver Assistance Systems (ADAS) at ASIL-B and electric power steering at ASIL-D share a processor are foreseeable. These applications, with varying dependability and fault detection and mitigation needs, pose unique challenges. ASIL-D applications demand the highest fault detection capabilities and safety mechanisms, while ASIL-B applications prescribe fewer requirements. Furthermore, non-critical applications like entertainment may share the processor with safety-critical ones.

As technology advances, more sophisticated driving assistant systems with Scociety of Automotive Engineers (SAE) levels 4 and 5 emerge [9]. These levels range from simple Adaptive Cruise Control (ACC) (SAE level 1) to full driving automation (SAE level 5).

SAE mandates a fallback system not reliant on the driver for levels 4 and beyond. Until SAE level 3, the satisfied system design is fail-safe, with the driver as the fallback option. However, from SAE level 4 onwards, the vehicle must operate without driver involvement, even in case of failure. These systems must be fail-operational, functioning flawlessly at all times [9].

To ensure fail-operational capability, system designers employ techniques such as the simplex architecture [10; 11; 12; 13]. This architecture utilizes a fallback compute system with reduced computational power or application complexity, providing a subset of functionality sufficient to keep the system operational during failure, although with reduced comfort. This concept is known as graceful degradation.

For complex systems like autonomous vehicles, graceful degradation may involve more than one ECU. Hence, to maintain fail-operational functionality, these systems may dynamically migrate tasks and functionality between distributed ECUs, potentially replacing non-critical applications with critical ones.

The challenge lies in balancing graceful degradation, simple architecture, and MCS design. The MCS platform must not only adapt to new functionality but also ensure the required functional safety for the application. Compliance with non-functional safety requirements, adhering to international standards, is crucial.

Despite these challenges, existing hardware solutions predominantly rely on static redundancy concepts and time predictability, rendering them unsuitable for MCS applications. One processor architecture specific designed for MCS is the FlexPRET processor [14], which guarantees WCET and prevents interference between critical tasks and non-critical tasks by segregation but overlooks functional safety requirements.

This thesis addresses the limitations of static redundancy designs by proposing an adaptive multi-core architecture capable of runtime-configurable redundancy. The aim is to meet the demands and requirements of mixed-criticality systems. The contribution extends beyond static or application-level design, presenting a fine-grained and highly adaptable processor architecture capable of configuring redundancy at runtime on-demand, even at the sub-functional level.

1.2 Contribution

Over the past decades, extensive research and literature has delved into the elaborateness of MCS, largely overlooking critical aspects such as fault tolerance, detection, and mitigation of transient faults. However, transient faults, as highlighted by various studies [15; 16; 17], stand out as the most common fault type in embedded systems.

This work aims to address this neglected aspects for mixed-criticality systems, focusing on both transient and permanent faults from a hardware perspective. The investigation and research centers around a flexible and runtime-adaptable processor architecture designed to overcome the limitations inherent in static fault-tolerant processor designs.

The key contribution lies in the incorporation of an on-demand reconfiguration mechanism that seamlessly switches between performance and reliability modes. Importantly, the reconfiguration process, which dynamically adjusts the system's reliability, remains hidden and abstracted from the executed user application.

This work contributes to the following aspects to achieve the on demand reliability configuration:

- Concept of on-demand Adaptive Redundancy (AR) for multi-core architecture
- On-demand hardware-based pipeline synchronization and runtime maintenance
- On-demand hardware-based processor state transfer and replication
- Adaptive Lockstep (AL) architecture (fine-grained redundancy)
- Embedding the Adaptive Lockstep (AL) architecture into a tile-based processor architecture
- Adaptive on-demand Cache Checkpointing (coarse-grained redundancy)
- Advanced state transfer methodology by interleaving the state transfer with the regular program execution

The fundamental of this research lies in the implementation of on-demand Adaptive Redundancy (AR) within multi-core systems, based on the broader concept of Adaptive Fault Tolerance (AFT) and configurable redundancy. Configurable redundancy, a central and underlying element, is used to precisely manage system reliability and dependability. Two runtime adaptive redundancy hardware methodologies are explored based on this principle of AR.

The first methodology is utilized in the Adaptive Lockstep Processor (ALP), a runtime-configurable processor architecture that introduces on-demand fault detection safety mechanisms at the instruction level. This approach employs fine-grained redundancy, where each instruction undergoes a comparison process. The ALP architecture requires hardware-based synchronization and state replication mechanisms, forming the foundation for both the fine-grained Adaptive Lockstep Processor (ALP) and coarse-grained the Adaptive Cache Checkpointing (ACCP) architectures.

To validate the feasibility and scalability of the ALP architecture, it is seamlessly integrated into a tile-based multi-core architecture. The coarse-grained ACCP architecture employs a comparison mechanism that assesses the results of multiple instructions. Consequently, any modifications to the system state resulting from executed instructions are tracked and, in the event of a fault, instantly reverted after detection.

A further contribution of this work affects to state transfer. Replicating the processor state from one core to another introduces potential timing overhead. To mitigate this, a state transfer methodology is investigated, interleaving the state transfer with regular code execution on the involved processor cores.

1.3 Outline

The organizational structure of this work outlined as follows. In Chapter 2, the foundational aspects are introduced, precisely defined and explained. Following this, Chapter 3 provides a comprehensive review of the current state-of-the-art and related work, encompassing diverse techniques aimed at enhancing system reliability, spanning both software, hardware, and hybrid approaches.

The introduction of the on-demand Adaptive Redundancy (AR) concept for multi-core architectures takes center stage in Chapter 4. This chapter begins by presenting and discusses on the principles of adaptive redundancy. It culminates with an exploration of the essential system states necessary for program migration and replication between two processor cores. This encompasses internal processor states, such as processor registers, and the program state in memory, characterized by the dynamic state of the stack and heap.

Expanding on the AR concept for multi-core architectures, Chapter 5 introduces and discusses the principles of the Adaptive Lockstep (AL) processor architecture. To implement an efficient safety mechanism for the target processor architecture, a thorough failure mode analysis of the pipeline architecture is conducted. This analysis probes into faults leading to Silent Data Corruption (SDC) and Control Flow Error (CFE).

The chosen processor architecture for this study is the open-source LEON3 processor from Frontgrade Gaisler AB[18]. This chapter leverages the integration of the safety mechanism into the pipeline design based on the previously conducted failure mode analysis. Before the safety mechanism can be operationalized, two or more processor cores must execute the same instructions concurrently, requiring both the synchronization of processor pipelines and the replication of the processor state. Pipeline synchronization is entirely managed by the hardware architecture, and, as discussed in the previous chapter, the replication process remains abstracted from the application.

The embedded fault detection and mitigation mechanism operates within the architecture. In reliable mode, the mechanism automatically detects faults and efforts to mitigate them by re-executing the flawed instruction. A detailed discussion of both aspects is conducted in this chapter.

The chapter concludes with an evaluation of the VHSIC (Very High Speed Integrated Circuits) Hardware Description Language (VHDL) implementation of the processor architecture. The ALP undergoes assessment for its fault detection behavior in both simulation and as an Field Programmable Gate Array (FPGA) prototype. For the fault evaluation of the FPGA prototype implementation, a customized hardware-based fault injection unit is designed and integrated. Furthermore, the design is evaluated in terms of its required resources and the introduced runtime overhead. The evaluation concludes with an investigation of the needed reconfiguration times to switch between performance and reliable mode.

In Chapter 6, the ALP is seamlessly integrated into a multi-core system based on a compute tile architecture. To achieve this, a hybrid network adapter is introduced, designed for the specific requirements of MCS, focusing on fault segregation of compute tiles and analyzability. The network adapter architecture merges two design philosophies with an optimization for distributed dataflow execution in mind.

Based on the designed network adapter, a Runtime Environment (RTE) dataflow is deployed to the tile-based architecture. The utilized RTE is briefly introduced, and deployment details are presented. The RTE supports fault-tolerant execution, incorporating both software and hardware-based execution dependent on the design of the ALP. The evolution of the network adapter and the deployed RTE conclude the chapter, along with a comparison of software and hardware-based adaptive redundancy. Additionally, a combination of hardware and software redundancy is investigated.

While the ALP employs a fine-grained redundancy concept, this approach is not universally applicable. Therefore, Chapter 7 introduces the more coarse-grained Adaptive Cache Checkpointing (ACCP) concept. The processor architecture, based on the principles of ACCP, incorporates loosely coupled processor cores. The chapter begins with the introduction of the ACCP concept's principles, followed by a discussion of different design aspects, including cache reconfiguration and the introduction of a checkpoint-capable register file for the processor core. The presented architecture automatically compares the processor states before creating any checkpoints. Each time a fault is detected by a divergence of two processor states, the changes since the last checkpoint are undone, restoring the last created checkpoint.

The chapter concludes with an evaluation of a VHDL implementation of the ACCP processor architecture. The evaluation considers hardware resources and the runtime behavior of the presented processor architecture. The ACCP is synthesized and implemented for an FPGA prototype.

The ACCP involves the replication of the complete processor state before two cores can execute the same program redundantly. The state replication process dominates the time needed to synchronize two cores. Therefore, in Chapter 8, an interleaved state transfer methodology is investigated. Instead of stalling the processor core to replicate the processor state, hardware features are introduced to replicate the processor state while simultaneously executing the program on both cores. The chapter explores different aspects of interleaved state replication, considering the requirements of an interleaved state transfer and hardware aspects. Based on the previous analyses, various state transfer strategies are investigated and evaluated. The VHDL implementation of the ACCP processor architecture is modified to support interleaved state transfer.

Each of these chapters delves into one or more of the previously described contributions of this work.

- **Chapter 4**: Concept of on-demand Adaptive Redundancy (AR) for multi-core architecture
- **Chapter 5**: On-demand pipeline synchronization and maintaining, processor state transfer and replication, and an Adaptive Lockstep (AL) architecture (fine-grained redundancy)
- **Chapter 6**: Embedding the Adaptive Lockstep (AL) architecture into a tile-based processor architecture
- **Chapter 7**: Adaptive Cache Checkpointing (coarse-grained redundancy)
- **Chapter 8**: Advanced state transfer interleave the state transfer with the regular program execution

This thesis is accomplished in Chapter 9, where a comprehensive comparison is performed between the presented and discussed architectural solutions and their counterparts in related work. Subsequently, the chapter ends with an outlook outlook on potential directions for future research and development.

Chapter 2

Fundamentals

2.1 Processor Architecture

Processor architectures are composed of multiple subsystems and interfaces. The following section introduces the most relevant subsystems and interfaces of a processor system, as utilized in this thesis. The fundamentals are explained, and relevant definitions are provided.

2.1.1 Instruction Set Architecture

The **Instruction Set Architecture (ISA)** serves as the interface between software and hardware, providing a set of instructions that enables the control of the processor by the software in the form of program code. Each instruction within the ISA induces a change in the state of the processor or the main memory. The architecture of an ISA allows for considerable flexibility, and depending on the complexity of the instructions, they are categorized as either Complex Instruction Set Computer (CISC) or Reduced Instruction Set Computer (RISC).

Complex Instruction Set Computer (CISC) instructions, exemplified by the x86 ISA, encompass a large set of instructions capable of performing complex operations. Furthermore, these instructions may vary in size for encoding.

Conversely, **Reduced Instruction Set Computer (RISC)** defines a small and simplified set of instructions, each dedicated to a specific task. The design of RISC instructions prioritizes efficient hardware implementation. Examples of RISC ISAs include RISC-V, Armv8 series, and Oracle's Scalable Processor Architecture (SPARC) v8 ISA.

Typically a RISC instruction is one of the following types:

- Load or store
- Arithmetic, logical, or shift
- Control transfer
- Read or write control register.

Load and store instructions are designed to access the main memory, often serving as the exclusive instructions for interacting with the main memory, contingent upon the memory model in use.

Arithmetic, logical, or shift instructions are utilized for data manipulation operations. In most RISC architectures, these instructions involve one or two source registers and an immediate value — a constant encoded within the instruction. The computed result is then stored in a designated destination register.

Control transfer instructions, encompassing jumps and branches, play a essential role in changing the program's control flow. These instructions can either be relative to the current instruction's location, as indicated by the Program Counter (PC), or independent of the PC. The execution of a control transfer may depend on the satisfaction or non-satisfaction of a condition. For instance, the SPARC v8 ISA employs an Integer Condition Code (ICC) to determine whether a control transfer is executed, with the ICC value being influenced by preceding arithmetic and logical instructions.

Furthermore, processors typically include instructions for configuring and reading special control registers.

Additionally, the ISA may provided some special instructions like:

- Floating-point operate
- Coprocessor operate.

These instructions play a crucial role in controlling hardware extensions and accelerators, with a common example being floating-point operations. Consequently, processor designers often integrate a dedicated Floating Point Unit (FPU) directly into the processor design.

Moreover, an ISA extends beyond the definition of instructions; it encompasses a memory model and the registers provided by the hardware implementation of the processor. These general-purpose registers store temporary results or serve as storage for passing function arguments during program execution.

The design of the provided general-purpose registers allows for significant flexibility. Typically, RISC architectures include 32 general-purpose registers. The SPARC v8 ISA adopts a distinctive approach to general-purpose registers, requiring more physical registers than are directly accessible. Programs can only access a subset of the 32 general-purpose registers, and the currently available registers are determined by the register window.

The program can manipulate the register window by issuing special instructions (e.g., SAVE or RESTORE), which increment or decrement the Current Window Pointer (CWP). In Figure 2.1, the SPARC v8 reg-

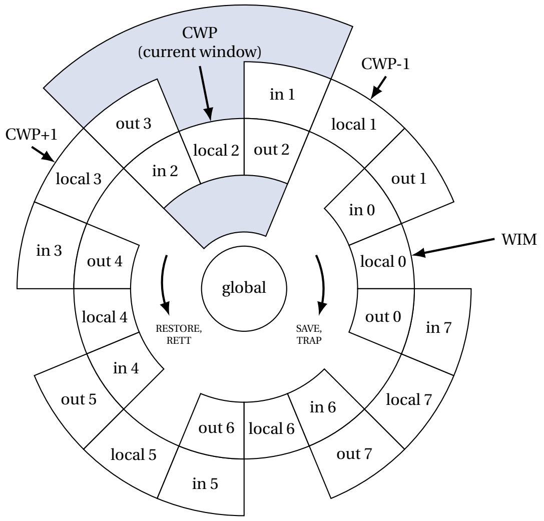


Figure 2.1: SPARCv8 Register Wheel (refer to [19])

ister wheel with eight windows is illustrated. Two adjacent windows share their *outs* and *ins registers*, with the *in registers* of the current window serving as the *outs* of the next window. Each window also has its own set of *local registers*. This combination of *in* and *out registers* is designed to efficiently handle function parameters and results. In addition to the 24 window-dependent registers, the SPARC v8 ISA defines eight globally accessible registers that are independent of the current window.

2.1.2 Processor Pipeline

At the heart of a processor is the processor core, taking over the responsibility of orchestrating the complex execution of instructions outlined by the ISA.

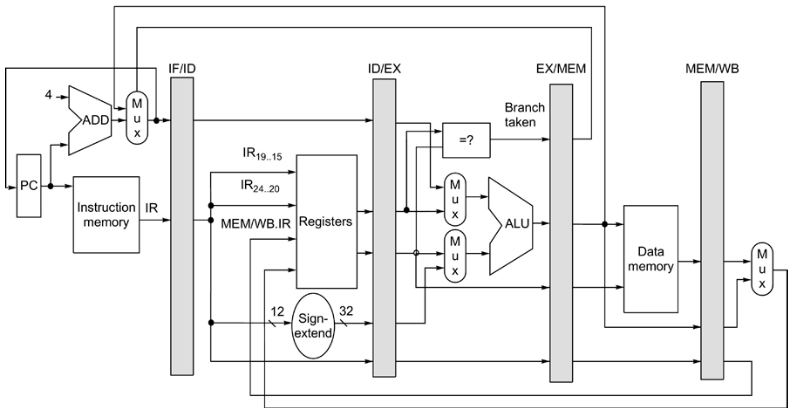


Figure 2.2: Five stage RISC processor design (from [4])

To enhance computational performance, processor cores leverage a pipeline architecture. Processor pipelines utilizes the power of Instruction-level Parallelism (ILP), where the straightforward approach of executing one instruction per clock cycle is transformed. Instead, each instruction is segmented into sub-tasks, each assigned to a specific pipeline stage. This architectural choice facilitates the parallel and simultaneous execution of multiple instructions, allowing subtasks from different instructions to be executed at the same time. Each sub-task performs a dedicated, smaller function, enhancing simplicity and speed in execution. Consequently, this parallelism significantly boosts the throughput of instructions.

A classical five stage RISC architecture following a DLX-Pipeline design is shown in Figure 2.2. The pipeline consist of the following stages each responsible for one dedicated task:

- Instruction fetch (IF)
- Instruction decode (ID)
- Execute (EX)

- Memory access (MEM)
- Write Back (WB).

The instruction fetch stage, the first in the pipeline, is responsible for the continuously providing of new instructions to the subsequent stages. This stage utilizes the PC to fetch instructions from the instruction memory continuously. The decode stage follows, configuring necessary internal control signals based on the fetched instruction, including general-purpose register selection. The execute stage handles arithmetic, logical, or shift operations based on input register values and control signals, encompassing branch target addresses or address calculations for memory accesses. Memory accesses are executed by the memory stage, which reads from or writes to the memory. Non load or store instructions simply passes thi stage. The read data from the data memory or the result of the Arithmetic Logic Unit (ALU) is written to the general-purpose registers by the write-back stage.

The use of a pipeline design introduces multiple hazards, arising from the partial execution of instructions. As only a sub-task of an instruction is completed, subsequent instructions may depend on the results of a previous instruction. Three classes of pipeline hazards are defined:

- Structural hazards
- Control hazards
- Data hazards.

A **structural hazard** arises from resource conflicts, where the hardware cannot execute all possible combinations of instructions concurrently. For instance, when two or more instructions in the pipeline need simultaneous access to the same resource, and the resource cannot provide parallel service, a structural hazard occurs.

A **control hazard** manifests when control flow instructions are executed, leading to a change in the PC. The example in Figure 2.2 illustrates a pipeline design where control flow instructions are executed in the execute stage. However, the next instruction must be fetched from the new PC, resulting in a control hazard, which needs to be resolved.

A **data hazard** emerges from a data dependency between two instructions. The current instruction relies on the result of a previous instruction, leading to a **Read-After-Write (RAW)** hazard. This occurs, for example, when the previous instruction has not completed the write-back stage, and an operand of the current instruction requires this result value. **Data forwarding** provides a solution to this hazard by supplying the current instruction with results present in the pipeline but not yet written to the target register. The data forwarding logic detects dependencies between instructions and resolves them.

Nowadays, CISC instructions internally utilize a microarchitecture based on RISC. To execute a CISC instruction, it is translated into one or more µ-Operations. These µ-Ops are then executed by the processor pipeline. µ-Ops bear similarity to RISC instructions and are better suited for pipeline architectures.

2.1.3 Memory Subsystem

The memory subsystem of a processor manages the access and provision of data to the processor pipeline, with a essential and central unit being the cache subsystem.

A cache optimizes data access latency by buffering data close to its computational unit, enhancing data locality by bringing it closer to the processing element. However, a cache cannot store all data; instead, it temporarily stores and buffers a subset of the complete main memory data. Consequently, the cache must identify if the requested data is already present.

A **cache hit** occurs when the processing element accesses data already present in the cache, whether it be for a read or write operation, allowing the cache to instantly execute the operation.

In the event that the data is not already cached, a **cache miss** occurs. The accessed data is not present in the cache and needs firstly to be fetched from the main memory. Once the cache allocates the data, the requested read or write operation is executed.

Caches utilize an organizational structure, where the smallest entry and atomic element is a **cache line**. Each cache line consists of multiple data words, which are sequentially located in the main memory. A cache line maps to words in the main memory, with the cache placement policies defining the strategy. The **cache associativity** determines how many possible locations a word in the main memory can be copied to and placed at the cache. One extreme is the **direct-mapped cache**, where one entry in the main memory can only be mapped to one cache line. The other extreme is **fully associative**, where each entry in the main memory can be mapped to any cache line. Figure 2.3 illustrates a direct-mapped cache, where one entry in the main memory can only be mapped to one cache line.

Between the extremes of the direct-mapped and fully associative caches lies the **n-way associative cache**. This design allows one entry in the main memory to be mapped to N lines in the cache. These N cache lines collectively form a **cache set**, ensuring that one entry in the main memory is always mapped to one of the cache lines within a set. Figure 2.3 illustrates a 2-way associative cache, where two cache lines are logically clustered to form a cache set.

The cache depends on additional information to determine whether a cache access is a hit or miss. This information is stored in the **cache**

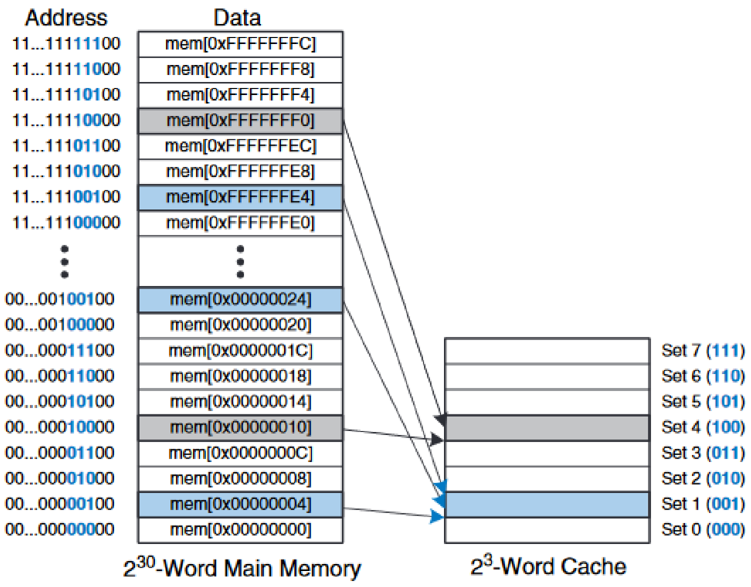


Figure 2.3: Direct-mapped cache (from [20])

tags. Each cache line features an associativity cache tag containing details about the cache line's validity (valid tag) or the associated memory address in the main memory (address tag). The valid tag is crucial for indicating cache lines that have not been fetched yet. An invalid cache line always results in a cache miss. The decision between a cache hit or miss is based on the accessed requested memory address, the validity of the accessed cache line, and the content of the address tag.

Caches can implement various write policies, dictating how the cache responds to a write access. When the processor core writes data to the

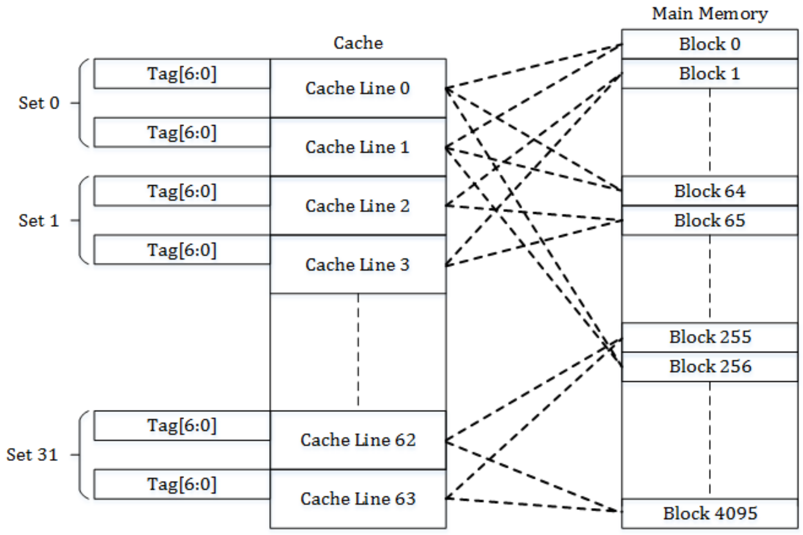


Figure 2.4: 2-way associative cache (from [20])

cache, it needs to control when data is written to the main memory. Two main write policies exist:

- The **write-through** policy writes the data directly to the main memory every time. The data is synchronously written to both the cache and the main memory.
- The **write-back** policy, on the other hand, follows an asynchronous strategy. Only the cache data is updated by the write, and writes are buffered and delayed by the cache until the cache needs to replace the data due to a cache miss. Before the data is replaced, the modified cache data is written back to the main memory.

Cache coherency becomes relevant in multi-core architectures, ensuring that copies of the same data stored in different caches are consistent. A system with cache coherence updates all cached data with the same address when data in one cache is modified.

2.1.4 Failure Modes of a Processor

A fault occurring during computation results in a **Data-flow Error (DFE)** due to a miscalculation in the data path. An undetected DFE can result into undetected **Silent Data Corruption (SDC)** — deviations between expected and actual stored data. In addition to faults within the data path, the memory itself is susceptible to SDCs. Faults at unprotected memory cells can directly lead to an SDC.

A true **Detected Unrecoverable Error (DUE)** results in a crash of the executed workload [21]. Such a crash can occur due to a processor hang or a processor trap, both of which prevent the program from progressing, leaving behind the processor unresponsive and halting normal execution. Processor hangs may arise from deadlock conditions or when the processor enters an infinite loop due to a fault.

Traps are designed to handle unexpected behavior by interrupting the current program execution and transferring control to a predefined software exception-handling routine. Traps may be triggered by arithmetic errors, such as division by zero, invalid memory access attempts to execute privileged instructions, or invalid instructions. Unexpected traps resulting from a **CFE** can lead to a processor crash, preventing further program progress.

False DUEs are detected and mitigated CFEs. A previously undetected SDC can later lead to a CFE.

2.2 Dependability

2.2.1 Fault, Error, Failure

Fault, error, and failure are commonly used terms in our language that, in the context of technical systems, describe a cause-and-effect chain. In this chain, a fault is the starting point, and the failure is the resulting outcome. It is crucial to understand and distinguish between the source and the observable result to prevent misbehavior, especially in digital systems. Digital computer systems, being among the most complex systems designed by humans, require a deep understanding of the correlation between a fault and its consequences. This knowledge is fundamental for providing countermeasures within the cause-and-effect chain to prevent fatal consequences.

To introduce the terms fault, error, and failure, it is essential to define a system. These three terms cannot be adequately defined without introducing a comprehensive system definition. A **system** is an entity that interacts with other entities, which can be other systems, humans, or the physical world. The encompassing systems form the **environment** of the given system, with the **system boundary** serving as the interface between the system and its environment.

Computing systems possess fundamental properties. Functional properties describe the system's intended function based on functional requirements. The implementation of these functional properties is the system's behavior, which can be described as a sequence of states. The system states of a computational system encompass computation, informational storage, and physical conditions, providing a comprehensive description of the entire system condition. The behavior of the system is described by state transitions, defined by the system's structure. A system comprises a set of components that form its structure. Each component either builds a further system or is atomic, with atomic components having no further discernible internal structure. Each system delivers a **service** to its user(s), where users can be other systems or human beings perceiving the behavior of the provided service [22].

Non-functional properties of a system describe how the system's functionality is delivered, including aspects such as cost, system performance, and dependability, which are pertinent non-functional requirements for computer systems.

A **fault** serves as the origin of a failure, impacting the system state. Various sources can trigger faults, categorized as systematic or random faults. Systematic faults arise during the system's development phase, introduced by human actions in the design process. In contrast, random faults emerge during operations and can be either internal or external. Internal random faults arises from variations in the fabrication process, packaging materials, and device deterioration due to aging. External faults, originating from the system's environment, result from interactions with the physical world beyond specified interfaces at the system boundary. Factors like temperature and energetic particles in the environment influence the electrical behavior of digital Complementary Metal-Oxide-Semiconductor (CMOS) systems, leading to unpredictable and spontaneous faults [23; 24; 25; 26; 27]. Typically, a fault results in a system error.

An **error** manifests as a deviation between the actual system state and the expected state, arising from a fault. Errors may propagate to a system failure.

A **failure** occurs when the delivered system services deviate from the expected services, resulting into the system unable to provide the correct service to the user(s). Service failures may be caused by user faults, and the cascade initiated by a fault can lead to the failure of the entire system. Figure 2.5 illustrates the creation and manifestation mecha-

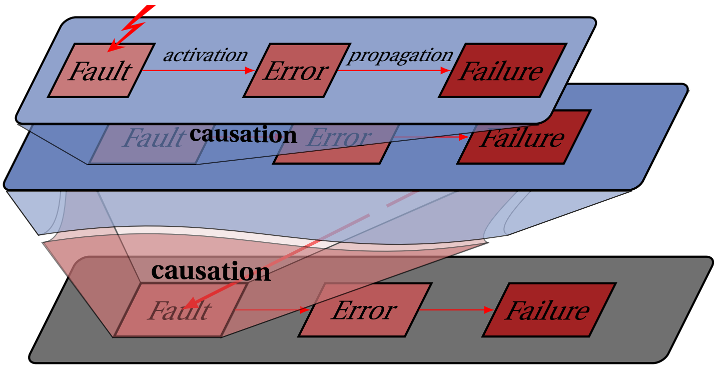


Figure 2.5: Relationship between fault, error and failure. A fault triggers an error which propagates to an failure. The failure of a component can cause a hierarchical fault of a further component.

nism of fault, error, and failure. The system consists of various hierarchical components, demonstrating how a fault propagates through the hierarchy.

It's important to note that faults may not propagate linearly through systems, as previously illustrated. Instead, faults can simultaneously influence multiple components. When multiple failures trace back to a common fault, they are referred to as **Common Cause Failures (CCF)**.

2.2.2 The Source of a Fault

The source of a fault is multifaceted, with fault sources categorized into static and random faults.

A **systematic fault** affects the designed system itself and arises without an external event. It manifests during the system's operating phase but originates during the design phase. Design flaws in both the software and hardware components of a computer system can result in abnormal conditions that lead to a fault.

A **random fault** emerges from phenomena in the physical world, occurring spontaneously and unpredictably. These phenomena particularly impact the behavior of transistors in digital computer systems based on CMOS technology.

Defects or **upsets** are abnormal conditions resulting in a deviation between the implemented system behavior and its intended behavior. The consequence of a physical defect or upset is a fault, serving as a logical abstraction of the underlying issue.

The downsizing of transistors to nanometer size and the reduction of the operating voltage make transistors more sensitive to the physical environment, particularly in terrestrial applications where radiation problems are increasing. The predominant issue is **Single Event Effects (SEEs)** caused by a single radiation event. An ionized particle striking the silicon can create transient ionization, resulting in a voltage pulse. This pulse can be either destructive or non-destructive. A destructive event, such as a **Single Event Latchs (SELs)**, occurs when the operating current exceeds device specifications, potentially leading to thermal device destruction. A parasitic thyristor created by an ionized particle can cause SELs, correctable by a power cycle or system reset.

A non-destructive event is transient, leading to a soft error [28]. A **soft error** is a temporary error correctable by the logic itself. The radiation effect resulting in a SEE can be primarily attributed to two dominant particle interactions.

The primary effects arise from the interaction of alpha particles or neutrons with the silicon of the transistor. Alpha particles, ionized radiation particles emitted from packaging material, can generate electron-

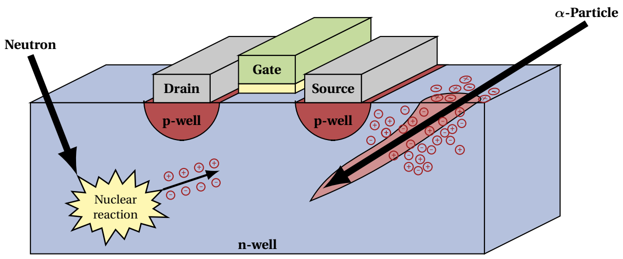


Figure 2.6: Alpha particle and neutron interacts with transistor material which creates electron-hole pairs. (refer to [29])

hole pairs (Electron-Hole Pairs (EHP)) when penetrating the transistor's silicon. Conversely, neutrinos from cosmic rays can indirectly induce single-event effects (SEEs). Neutrinos react with the transistor material's nucleus, generating secondary particles, such as protons, alpha particles, and heavy ions, which become the source of the soft error observed [29]. Charged particles, either alpha particles or those resulting from a nuclear reaction with neutrons, create EHP on their path and deposit charges. Both effects are illustrated in Figure 2.6. Alpha particles directly generate EHP within the transistor material, while neutrons first produce secondary particles through nuclear reactions.

The generated charges drift and diffuse towards the transistor's drain, where they accumulate. This accumulation eventually leads to a soft error [30].

Soft errors can be distinguished between a Single Event Upset (SEU) and a Single Event Transient (SET). Despite sharing the same physical origin, the distinction lies in the location of the event. If the fault occurs in a memory element, such as a latch, Flip-Flop (FF), Static Random-Access Memory (SRAM) cell, or Dynamic Random-Access Memory (DRAM) cell [26; 31], it is an **SEU**. The corruption of the mem-

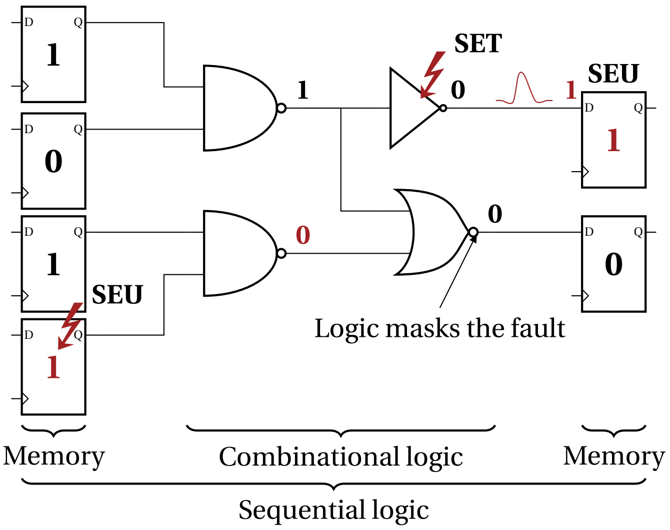


Figure 2.7: Safety relevant time intervals defined by the ISO26262 standard.

ory state is not permanent and can be corrected and overwritten by a subsequent write access. On the other hand, if the fault arises within the combinatorial logic, such as an *AND* or *OR gate*, it is a **Single Event Transient (SET)**. The SET can propagate through the combinational logic until it reaches a memory element. If a SET is sampled by a latch or FF, it becomes the source of an SEU [31]. Figure 2.7 illustrates the locations of SEUs and SETs and the relationship between the two. The SEU of the input register results in a bit flip from '0' to '1', and the SET of the inverter flips the output from '0' to '1'. The erroneous '1' is then sampled by the register, causing an SEU and rendering the SET persistent. Without sampling the erroneous value, the SET would not propagate and would be masked by the sequence logic. Another way logic masks an error is seen in combined logic, where, under specific conditions, the logic, such as an *OR gate*, masks the SEU of the FF.

SEU and SET characterize a **transient fault**. This fault is a single, spontaneous event that occurs briefly and then vanishes. Its impact is transient, existing only for a short duration within the component's lifetime. Transient faults fall into the category of random faults.

In contrast, when a fault repeats intermittently before disappearing, it is termed an **intermittent fault**. Cross talks within the device, such as signal interference on signal line cross-sections, can be a root cause of intermittent faults. These faults result from suboptimal system design and are preventable during the development phase, categorizing them as systematic faults.

Alternatively, a fault may persist continuously, constituting a **permanent fault**. Various factors contribute to permanent faults, with manufacturing defects and aging being primary causes. Manufacturing processes, susceptible to variations, may introduce defects deeply embedded in the device structure before the operational phase [32; 33]. Aging, occurring throughout the device's operational phase, involves continuous degradation of transistors, leading to undesirable behavior. Long-term ionization contributes to the Total Ionizing Dose (TID), degrading the transistor's performance. Negative Bias Temperature Instability (NBTI) emerges as a dominant aging effect, gradually slowing down PMOS transistors over time. Slower transistor speeds can result in delay errors. Some EHP generated by ionized particles are annihilated, and the remaining EHP may fall into traps at the Si-SiO2 interface, generating a negatively biased gate-source voltage. Holes generated during this process contribute to interface traps and oxide-fixed-charge through reactions with Si-H bonds [31; 34; 35; 36].

2.2.3 Reliability and Availability

When developing technical systems, it is necessary to measure individual properties to facilitate a comprehensive comparison of various solutions. For safety-critical systems, metrics in regard to reliability and availability assume high importance alongside the usual performance metrics. Reliability and availability, though closely related, describe distinct aspects of a system.

Reliability serves as a metric measuring the capability of a system or component to function as intended. In contrast, **availability** quantifies the duration during which a system is capable of delivering its intended service. While these measures share an intrinsic connection, they describe different facets of system properties.

In the context of introducing faults and failures, two essential observations emerge: firstly, a fault can lead to a failure, and secondly, faults are inherently unpredictable. A fault, being inherently unobservable, manifests as a malfunction resulting from a system failure caused by the fault. Locating and predicting the exact occurrence of a fault is impossible due to the stochastic nature of random faults, which transpire spontaneously. However, failures can be modeled using probability theory, leveraging the stochastic characteristics of fault processes.

The expression of a system's reliability often relies on the probability that the system will provide the expected service without encountering a failure. A common approach involves using the **Mean Time to Failure (MTTF)** definition, which denotes the average time until a system fails. This metric is grounded in a probabilistic system description, with reliability values ranging between zero and one. Over a sufficiently extended operating period, every system or device is bound to experience failure. As time progresses, the probability of a system failure increases, and reliability converges to zero [37].

The **reliability function** $R(t)$ offers a means to describe the probability of survival over time, denoted by t. Constrained between zero and one, the reliability function is generally defined by the count of failure-free units $n(t)$ up to time t relative to the initial quantity of units n. Conversely, the **failure distribution function** $F(t)$ complements the reliability function $R(t)$, representing the probability that a system will fail before reaching time t. The definitions of the reliability function $R(t)$ and the associated failure distribution function $F(t)$ are as follows:

$$R(t) = \frac{n(t)}{n_0} = e^{-\lambda(t)t}, F(t) = 1 - R(t). \tag{2.1}$$

To articulate the reliability function, an exponential function and a time-dependent failure rate $\lambda(t)$ are employed. The failure distribution function of the system, denoted as $F(t)$, is complemented by its temporal derivative, yielding the failure density function $f(t)$. The **probability density function** $f(t)$ must be normalized to adhere to probabilistic principles. Consequently, $f(t)$ is defined as follows:

$$f(t) = \frac{dF(t)}{dt} = -\frac{dR(t)}{dt}, \int_0^\infty f(t)\,dt = 1. \tag{2.2}$$

The time-dependent **failure rate** $\lambda(t)$ represents the probability that a system, which has not failed up to time $t + dt$, will experience a failure within this infinitesimally small time interval dt. It quantifies the average number of units failing at a specific moment in time and cannot be measured for an individual unit. The expression for the time-dependent failure rate is as follows:

$$\lambda(t) = \frac{f(t)}{R(t)} = -\frac{1}{R(t)} * \frac{dR(t)}{dt}. \tag{2.3}$$

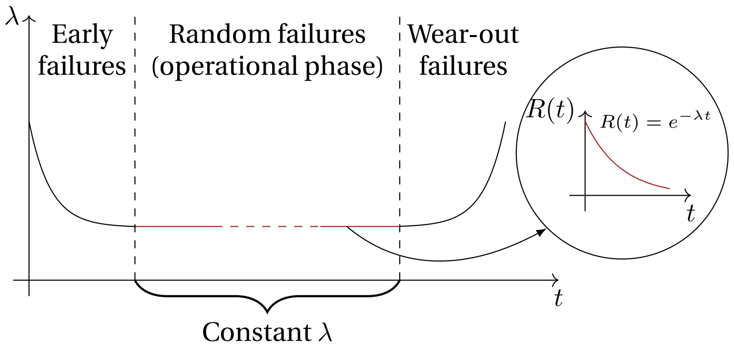


Figure 2.8: Bathtube model (refer to [37])

Estimating failure rates involves observing a substantial number of comparable systems. A common pattern observed is that the failure rate of systems undergoes significant changes at the beginning and ending of their operational lifetime. Early failures emerge immediately after production, arising from manufacturing variations. These failures can be pinpointed through system tests and induced by straightforward system stress. Throughout the operational phase, random failures predominate, and the failure rate $\lambda(t)$ can be regarded as relatively constant. Subsequently, as systems progress into the wear-out phase post-operational phase, the failure rate experiences a substantial surge due to aging. The described bathtub model, showing these three distinct phases, is illustrated in Figure 2.8.

The MTTF represents the average time until a system failure occurs. It is the arithmetic mean value of the reliability function $R(t)$, aligning with the expected value of the failure density function $f(t)$. In the system's operational phase, the failure rate remains constant, leading to the following expression:

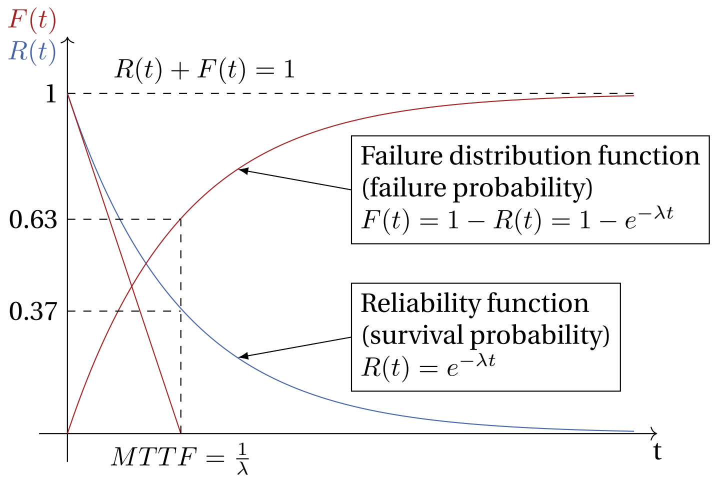


Figure 2.9: Visualization of the relationship between reliability function, failure distribution function, and MTTF (refer to [37])

$$MTTF = \int_0^\infty t f(t)\, dt = \int_0^\infty R(t)\, dt \xrightarrow{R(t)=e^{-\lambda t}} \int_0^\infty e^{-\lambda t}\, dt = \frac{1}{\lambda}. \tag{2.4}$$

The average time until a system failure, denoted as $t = MTTF$, is calculated as $1/\lambda$. For systems exhibiting a constant failure rate, the MTTF is simply the reciprocal of that rate.

In Figure 2.9, the relationship between survival probability, failure probability, MTTF, and failure rate is elucidated, focusing on a constant failure rate λ.

The MTTF serves as a metric for measuring a system's reliability, disregarding potential repairs. In contrast, availability, a essential characteristic for repairable systems, encompasses repair considerations. It encompasses not only the MTTF but also accounts for the time required to repair a system, known as **Mean Time to Repair (MTTR)**. MTTR signifies the average time necessary to repair a system failure, capturing the duration between the occurrence of a system failure and the restoration of its service. When combined, MTTR and MTTF resulting into the **Mean Time between Failure (MTBF)**. In particular, $MTBF = MTTF + MTTR$, representing the average time between two system failures.

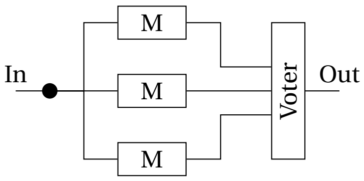
System availability is defined as the duration during which a system consistently delivers the intended service. This is calculated by dividing the MTTF by the MTBF.

$$A = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF} \tag{2.5}$$

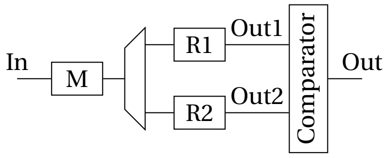
2.2.4 Fault Tolerance

Fault tolerance techniques are designed to safeguard systems against faults, enhancing their reliability and dependability. One of the most prevalent strategies involves introducing redundancy into the system. Typically, redundancy is categorized into three common types.

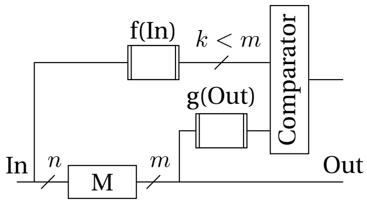
- **Spatial redundancy**: This redundancy concept is illustrated in Figure 4.8c, involving the redundant execution of the same function on distinct hardware components. In N-Modular Redundancy (NMR), the functionality is replicated N times, employing comparator and voting techniques to enhance system dependability. Examples include **Dual Modular Redundancy (DMR)** and **Triple Modular Redundancy (TMR)**, where the original module is duplicated two and three times, respectively. All redundant modules receive identical inputs, and errors are detected through result mismatches.



(a) Spatial Redundancy



(b) Temporal Redundancy



(c) Informational Redundancy

Figure 2.10: Comparison of Redundancy (refer to [39])

- **Informational redundancy**: Illustrated in Figure 2.10c, informational redundancy relies on coding techniques. Extra information is added to the original data to detect or correct errors, reducing duplication costs. This approach is widespread in telecommunication channel encoding, featuring error-detecting codes like AN codes or Error-Correcting Code (ECC), such as Hamming codes or Reed-Solomon-Codes [38].
- **Temporal redundancy**: Illustrated in Figure 4.8d, temporal redundancy involves executing a function multiple times on the same hardware module. The results from repeated executions are compared, and temporal redundancy can handle transient or intermittent faults. However, it is unable to detect permanent faults, as a malfunctioning system consistently produces the same incorrect result.

Backward Error Recovery (BER) is a universal concept applicable to most systems, utilizing one of the fault detection techniques previously introduced. In this method, the system cannot directly mask or correct the fault. Instead, the last valid system state is utilized to re-execute the flawed functionality. Examples include a DMR configuration or the use of AN codes to detect errors. System checkpoints are also considered as a form of Backward Error Recovery (BER). When a fault is detected, the system reverts to the last valid checkpoint and re-executes the functionality. Therefore, before creating a new valid checkpoint, the system must ensure the correctness of the checkpoint.

Unlike Backward Error Recovery (BER), **Forward Error Recovery (FER)** or **Forward Error Correction (FEC)** not only detects faults but places a greater focus on directly correcting the error. For instance, TMR with a majority voter is a typical spatial redundancy method, while ECC is a well-known informational redundancy technique capable of directly correcting present faults. ECCs are particularly utilized for data storage in memory and data transmission, being a common method in telecommunications for transmitting data over noisy communication channels.

The **Sphere of Replication (SoR)** provides a general description of redundant execution mechanisms with a specific focus on spatial redundancy. It remains independent of the employed error recovery method and comprises three essential design parameters [41].

- **Components**: Determining which components should undergo redundant execution and which should not is a critical decision. Components without redundant execution are either not safeguarded or necessitate alternative concepts, such as informational redundancy (e.g., ECC).
- **Outputs**: Deciding which output signals to compare is essential. To minimize comparison overhead, only relevant signals should

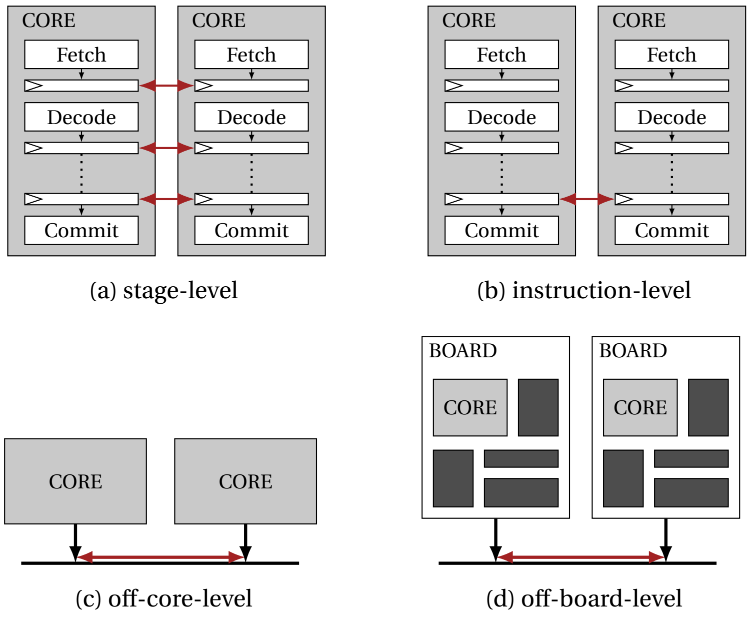


Figure 2.11: Sphere of Replication: Four different spheres are illustrated. The red arrows symbolized the comparison and fault detection. (refer to [40])

be considered. However, abstaining from comparing critical signals threaten fault coverage. Designing the right balance is essential for an effective fault detection strategy.

- **Inputs**: Identifying the inputs that require replication is crucial to guarantee the correct behavior of redundant components. Replicating the appropriate inputs ensures consistency and coherence in the performance of redundant elements.

In processor systems, four different SoRs are illustrated in Figure 2.11. One of the primary distinguishing features of the SoR is the location where results (output) are compared, encompassing the components generating the results and the correct inputs for those results.

Lockstep processors commonly employ either a stage-level or instruction-level SoR [42; 43; 44; 45]. The **stage-level** SoR, illustrated in Figure 2.11a, represents the most fine-grained redundancy mechanism. It involves comparing the result of each pipeline stage, utilizing the register values resulting from the previous stage to detect faults. This method ensures a fast and timely fault detection, but the hardware overhead is substantial. Each pipeline stage requires at least one comparator to verify the result. This technique is equivalent to a DMR on Register-Transfer Level (RTL).

In contrast, the **instruction-level** SoR, illustrated in Figure 2.11b, avoids comparing the result of every pipeline stage. Instead, it performs a single comparison at the last pipeline stage before committing the instruction result to the system state, which includes the processor register or main memory.

Moving towards a more coarse-grained approach, Figure 2.11c illustrates the **off-core-level** SoR. Here, only data leaving the processor core is checked for faults before being committed to the system state. While this increases fault detection latency, it reduces the number of required comparisons.

Finally, the even more coarse-grained SoR is demonstrated in Figure 2.11d. This **off-board-level** approach compares the I/O behavior of two boards, where a deviation in the boards' activities indicates a system fault. While the previous SoRs can be implemented within a multi-core architecture, the off-board-level SoR extends the comparison scope to the external I/O behavior of the boards.

2.2.5 Checkpoints

Checkpoints play a essential role in BER techniques, preserving the system state to enable resumption or restoration to a later point in time. The taxonomy illustrated in Figure 2.12 is commonly employed to classify these checkpoints. The axis labeled "sphere of checkpoint" defines the memories encompassed by the BER, drawing an analogy to the SoR. However, the BER's sphere encompasses the entire memory hierarchy, extending up to the highest memory element of the checkpoints. Faults within this sphere can be recovered by rolling back to the last valid checkpoint, restoring the previous system state. Information and data propagating beyond this sphere must be correct, as they are not reversible by subsequent checkpoint restoration.

The second aspect of the taxonomy pertains to the location where the checkpoint is stored. Leveled checkpoints are stored on a lower memory hierarchy compared to the covered checkpoint. In a dual scheme, the checkpoint is stored on the same hierarchy, and additional hardware structures are employed for storage.

The third axis of the taxonomy differentiates the separation of the checkpoint from the active data. In a full separation scheme, the checkpoint is entirely isolated from the active data. Conversely, in a partial scheme, the active data and the checkpoint data share a common memory. Modifications to the active data are tracked for potential rollback. Changes can be monitored through buffering, renaming, or logging methods. Buffering accumulates all changes since the last checkpoint, renaming redirects modifications to prevent overwriting current data, and logging saves both the old value and the modifications applied to the active data.

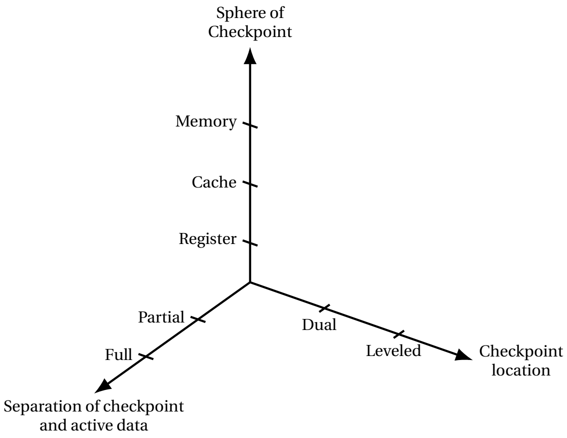


Figure 2.12: Taxonomy of checkpoint properties (refer to [46])

2.2.6 Adaptive Fault Tolerance

Static Fault Tolerance (SFT) aims to enhance the dependability of systems by incorporating redundancy, duplicating data and operations during design to achieve fault tolerance. The level of redundancy is predetermined at the design stage, ensuring the system is appropriately structured to meet safety requirements. However, this approach may lead to underutilization of resources if the system does not operate at its maximum specifications. Systems designed with SFT are typically tailored to a well-defined environment and predefined system requirements.

While SFT satisfies in static workloads, it may prove inadequate for dynamic scenarios, where less critical tasks requires guaranteed redundancy, potentially leading to the rejection of critical tasks [47]. Static Fault Tolerances are optimized for specific workloads and are not well-suited to adapt to environmental changes.

To overcome these limitations, the concept of AFT has been developed [48; 49; 50]. AFT is designed to successfully operate in dynamically changing environments, providing systems with the ability to adapt to varying conditions.

> "Adaptive fault tolerance (AFT) is an approach to meeting the dynamically and widely changing fault tolerance requirement by efficiently and adaptively utilizing a limited and dynamically chancing amount of available redundant processing resources." [48]

Environmental changes can result in alterations to both functional and non-functional requirements, including fault tolerance measures. Consequently, Adaptive Fault Tolerances are defined as an approach to dynamically fulfill varying fault tolerance requirements by adaptively utilizing limited processing resources.

In the face of environmental changes or node failures, adjustments to functionality and operating strategies become necessary. An adaptive fault tolerance management system is employed to orchestrate the processing systems and fine-tune the operating strategies. The adaptivity of the strategy encompasses three main types:

- Parametric changes of fault tolerant mechanism
- Anticipate faults through monitoring and control them
- Isolating faults of faults caused by common latent faults.

Complex systems often provide multiple services, consisting of numerous individual components. To enhance system availability, the quantity or quality of services may be reduced. Operating with limited system functionality prevents a complete system failure, a concept known as **graceful degradation**, which falls under the concepts of AFT methods [47]. In graceful degradation, parametric changes are made to the system to maintain reduced functionality. The system drops non-essential workload, ensuring a fail-operational state with a basic reduced functionality.

In contrast to AFT, there is **dynamic redundancy**. This involves replacing a faulty component with a backup component, which can be either in hot or cold standby mode. In **hot standby**, the backup component runs in parallel in the background and automatically takes over when the primary component fails. On the other hand, **cold standby** requires the component to start before taking over.

2.2.7 Standards for Safety Critical Systems

When developing safety-critical systems, designers must adhere to international standards and norms, which provide a framework and guidelines on ensuring system safety. However, these standards offer an abstract description of how functional safety is to be achieved, leaving the specific measures open to interpretation.

A detailed description of functional safety is provided by IEC 61508, titled "Functional safety of electrical/electronic/programmable electronic safety-related systems." This standard defines two key aspects. Firstly, it defines that overall safety depends on the correct operation of a system or equipment in response to its inputs. Secondly, it highlights the importance of detecting potentially dangerous conditions, leading to the activation of a protective or corrective devices or mechanisms to prevent hazardous events or mitigate their consequences.

The standard introduces a hazard and risk analysis, requiring an assessment of two categories. The first category quantifies the likelihood of a risk occurring, with six defined categories ranging from *frequent* (many times in a lifetime - failures per year > 10^{-3}) to *incredible* (cannot believe that it could occur - failures per year > 10^{-7}). The second category assesses the consequence of the asset's failure, with four

Table 2.1: Acceptable failure rates according to the IEC 61508 (refer to [51])

Safety Integrity Level	Failure Probability for a Function on-demand	Continuous Failure Rate
SIL 1	$10^{-2} \leq F < 10^{-1}$	$10^{-6} \leq \lambda < 10^{-5}$
SIL 2	$10^{-3} \leq F < 10^{-2}$	$10^{-7} \leq \lambda < 10^{-6}$
SIL 3	$10^{-4} \leq F < 10^{-3}$	$10^{-8} \leq \lambda < 10^{-7}$
SIL 4	$10^{-5} \leq F < 10^{-4}$	$10^{-9} \leq \lambda < 10^{-8}$

defined categories ranging from *catastrophic* (multiple loss of life) to *negligible* (minor injuries at worst). Based on a combination of these factors, a Safety Integrity Level (SIL) is determined. Therefore, the IEC 61508 outlines four different SILs, as shown in Table 2.1.

As seen in Table 2.1, SIL requirements vary based on functions, distinguishing between those on demand and those with continuous execution. For functions rarely used or on demand, the standard employs the failure probability F. In contrast, for frequently used or continuously executed functions, the failure rate λ is utilized. Conventionally expressed in Failure in Time s (FITs) units, where one fit unit corresponds to one failure after one billion (10^9) operating hours, a continuous operating functionality with SIL 4 should exhibit only one failure per billion hours of operation or, equivalently, a FIT rate of one.

Several standards inherit from IEC 61508 and incorporate these principles. Notable examples include DO-178C for avionics, IEC 62279 for railways, and IEC 62061 for machinery. The automotive industry relies on ISO 26262, "Road vehicles - Functional safety," for electrical and electronic systems in passenger vehicles, adapting the IE 61508 to suit automotive needs.

The standard distinguishes between systematic faults and random faults [52]. Systematic faults manifest during the design and implementation phase, originating from human error. To mitigate system-

Table 2.2: Suggested requirements associated to the ASIL class (refer to [54])

ASIL Level	Random Hardware Failure Target Values	Single-Point Fault Metric	Latent-Fault Metric
B	$< 10^{-7}\, h^{-1}$ $(100\, FIT)$	$\geq 90\%$	$\geq 60\%$
C	$< 10^{-7}\, h^{-1}$ $(100\, FIT)$	$\geq 97\%$	$\geq 80\%$
D	$< 10^{-8}\, h^{-1}$ $(10\, FIT)$	$\geq 99\%$	$\geq 90\%$

atic faults, processes must be implemented to prevent their occurrence. Random faults, on the other hand, are unforeseen and unpredictable. Hence, **safety mechanisms** are applied to mitigate the influence of a fault, maintaining independent functionality by detecting and mitigating faults. Safety mechanisms may provide technological solutions to tolerate faults or avoid and control failures.

For non-critical functionality, ISO 26262 introduces the class Quality Management (QM), indicating that no safety measures in regard to the standard are required. When assigning an ASIL class to a hazards analysis, three criteria are assessed [53]. Two classifications mirror those used in IEC 61508: *Severity*, similar to the consequence of a hazard, and *Exposure*, indicating the risk of occurrence. Additionally, ISO 26262 incorporates *Controllability*, classifying the relative likelihood that the driver can prevent injury through intervention. Based on these three classifications, a level between A and D or QM is determined.

Reliability requirements are defined based on the ASIL, utilizing three major metrics in ISO 26262:

- Random hardware failure (target values)
- Single-Point fault metric
- Latent-Fault metric

The **hardware failure rate** denotes the maximum allowable FIT rate for a hardware component. While, the **single-point fault metric** outlines the minimum number of individual faults, out of all possible faults, that must be detected by at least one safety mechanism. According to the standard, a **latent fault** is classified as a multiple-point fault, representing a combination of independent faults. In this scenario, the safety mechanism or the driver fails to detect the presence of at least one fault. The **latent-fault metric** quantifies how many of the potential latent faults must be addressed by a safety mechanism. The ISO 26262 provides suggestions for each of the metrics depending on its ASIL classification. Table 2.2 provides an overview of the suggested values.

Furthermore, the standard comprehensively defines all relevant time intervals, as illustrated in Figure 2.13. Among these, the most relevant is the Fault Tolerant Time Interval (FTTI), which establishes the maximum time between the appearance of a fault and the manifestation of malfunctioning behavior leading to a hazardous event. This interval represents the duration during which an unhandled fault can induce a hazardous event. Within the FTTI, fault detection and subsequent fault handling must be executed to prevent the occurrence of a hazardous event. Two scenarios are considered:

- **Transitioning into a Safe State:** After detecting a fault within the Fault Detection Time Interval (FDTI), the system requires time to transition into a **safe state**. The safe state is an operating mode with an acceptable level of risk. The duration of transitioning into the safe state is defined as the Fault Reaction Time Interval (FRTI). Both FDTI and FRTI collectively sum up to the Fault Handling Time Interval (FHTI), which must be shorter than the FTTI.

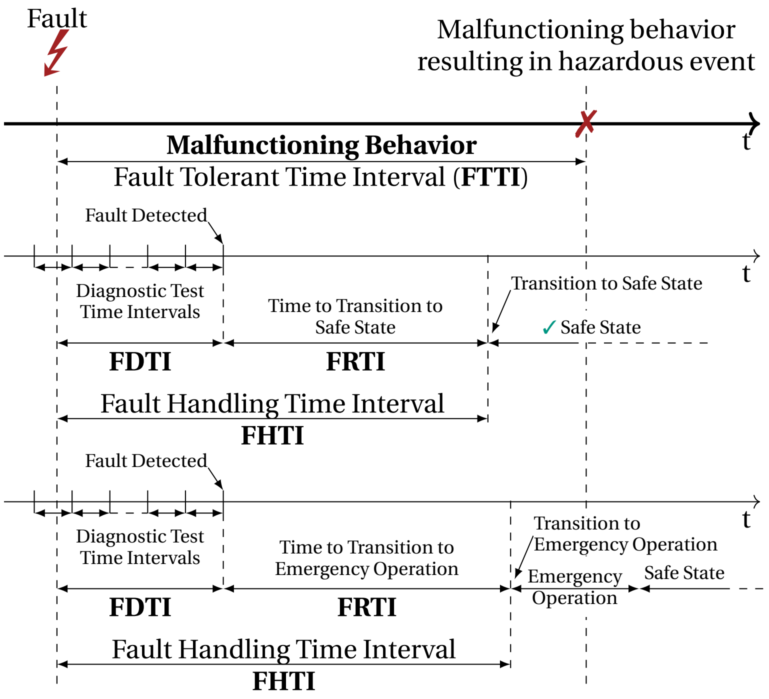

Fault
Malfunctioning behavior resulting in hazardous event
Malfunctioning Behavior
Fault Tolerant Time Interval (FTTI)
t
Fault Detected
Diagnostic Test Time Intervals
Time to Transition to Safe State
Transition to Safe State
✓ Safe State
FDTI
FRTI
Fault Handling Time Interval
FHTI
t
Fault Detected
Diagnostic Test Time Intervals
Time to Transition to Emergency Operation
Transition to Emergency Operation
Emergency Operation
Safe State
FDTI
FRTI
Fault Handling Time Interval
FHTI
t

Figure 2.13: Safety relevant time intervals defined by the ISO 26262 standard (refer to [52]).

- **Transitioning into Emergency Operation:** In this scenario, the system transitions into emergency operation before reaching the safe state. Emergency operation is employed to prevent the occurrence of a hazardous event and extends the time available to reach the safe state. Graceful degradation can be an option to facilitate the emergency operation.

Additionally, Table 2.3 provides an overview and comparison of defined safety levels from different standards.

Table 2.3: Comparison of safety levels (refer to [55])

Critical System Domain	**Safety levels (high to low)**				
General IEC 61508 SIL	4	3	2	1	-
Automotive ISO 26262 ASIL	-	D	C/B	A	QM
Avionic DO-178C DAL	A	B	C	D	E
Railway EN 50128 SIL	4	3	2	1	-

2.3 Mixed Criticality Systems

Mixed Criticality System (MCS) encompasses systems that execute tasks with varying levels of criticality. A essential question in MCS revolves around unifying conflicting demands for safety assurance while efficiently managing resource utilization. This challenge arises research questions such as exploration into theoretical aspects such as modeling, hardware platform design, software runtime implementation, and system verification [8].

Typically, criticality models in MCSs consist of two levels: tasks are categorized as either critically important with **high criticality** (*'Hi'*) or non-critical with **low criticality** (*'Lo'*).

The execution time of a task emerges as a crucial system parameter for the modeling and design of MCSs. Critical tasks demand termination within specified time frames, requiring scheduling algorithms that guarantee execution times for such tasks. Conversely, non-critical tasks do not require this guaranteed execution times.

Within the context of scheduling critical tasks in mixed critical systems, a essential system parameter is the **WCET**. The WCET represents the theoretically longest execution time a task might require, essential for planning in worst-case scenarios, especially in the realm of hard real-time systems [56].

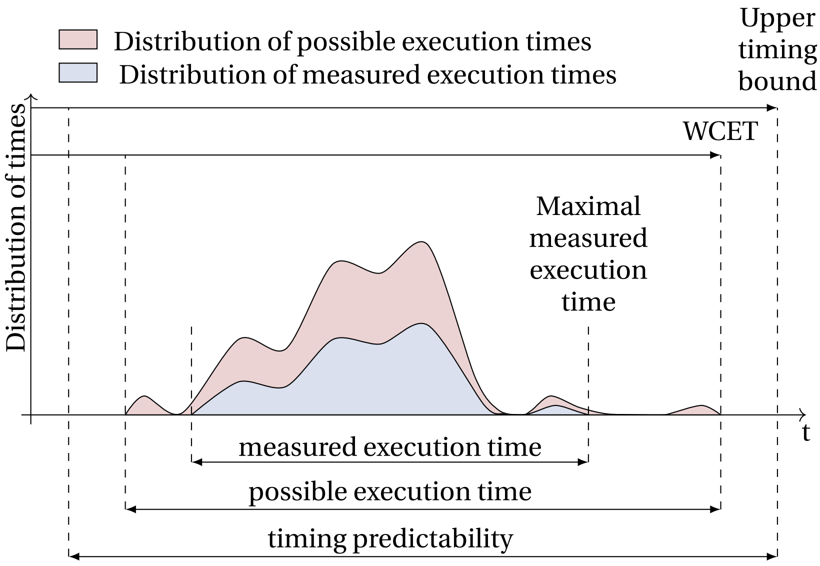

Distribution of possible execution times
Distribution of measured execution times
Upper timing bound
WCET
Distribution of times
Maximal measured execution time
t
measured execution time
possible execution time
timing predictability

Figure 2.14: Example WCET (refer to [56])

The distribution of potential and measured execution times is visualized in Figure 2.14. The WCET represents the theoretical worst-case execution time. It is important to note that the measured execution times may not always reach this theoretical maximum due to insufficient inputs. Nevertheless, specific input combinations have the potential to manifest the theoretical WCET.

Chapter 3

Related Work

This chapter conducts a comprehensive review of related work and the current state-of-the-art in software, hardware, and hybrid approaches aimed at enhancing system reliability. Special focus is placed on techniques suitable for mixed-critical systems. The explored software techniques and hardware architectures include strategies for fault detection and fault recovery. Transient faults are identified as the most common type of fault in embedded systems [15; 16; 17]. Consequently, this review focuses on approaches that address at least transient faults.

Fault detection techniques are classified, as outlined in Table 3.1, considering detection latency and the targeted fault modes. The review comprises all these detection techniques except for Built-in Self-Test (BIST). While Memory BIST and Logic BIST are well-known for detecting permanent faults, this work specifically focuses on detecting both transient and permanent faults. Therefore, BIST techniques are considered out of scope.

Table 3.1: Comparison of online fault detection techniques (refer to [57; 58; 59; 60])

Detection Technique	Detection Latency	Targeted Fault
Lockstep Execution	Cycle-by-Cycle Lockstep	Transient and Permanent Faults
Checkpoiting	Checkpoint Interval (Low)	Transient and Permanent Faults
Software-based Redundant Execution	Low	Transient Faults (Single Core) and Permanent Faults (Multi-Core)
Dynamic verification	Low	Transient and Permanent Faults
Built-in Self-Test (BIST)	Very Low (Test Period)	Permanent Faults

3.1 Software Fault Tolerance

SWIFT stands out as one of the most prominent software-based safety mechanisms [61]. Operating as a single-thread software approach, it focuses on fault detection [62]. This method leverages the ILP of the processor to minimize runtime overhead, making use of idle resources to enhance execution time. SWIFT employs compiler-based transformations to embed safety mechanisms directly into program instructions. Instructions are duplicated to detect SDCs and incorporate an advanced Control Flow Checker (CFC) for detecting Control Flow Error (CFE). The duplicated instructions differ on purpose to avoid interference, utilizing distinct registers as operands and destinations. Values from both sets of calculations are compared whenever a store instruction attempts to write outside of the Sphere of Replication (SoR). SWIFT assumes that the main memory and caches are protected by ECC. To optimize performance, the SoR is limited to the processor pipeline.

Chang et al. [63] introduce enhancements to the original SWIFT approach. The first improvement, **SWIFT-R**, adopts a TMR approach instead of DMR. SWIFT-R not only detects errors but also enables a fault recovery mechanism based on majority voting. However, triplicating execution introduces significant computational overhead. To address this, the authors propose **TRUMP**, which refines the DMR concept of SWIFT by introducing additional informational redundancy. Arithmetic codes, where operands are multiplied by a constant value, create this redundancy to protect calculations and detect errors. A more lightweight solution is **MASK**, which utilizes statistical properties to guard against bit flips in register values.

SWIFT and its improved techniques rely on temporal redundancy, detecting transient faults but providing insufficient protection against permanent faults. Moreover, these techniques utilize an indirect CFC mechanism, safeguarding only certain operations to ensure valid values are written to main memory. However, these compiler-based approaches fall short of protecting against errors affecting the operational code of an instruction. For instance, if an error results in decoding a non-store instruction as a store instruction, such execution flaws remain undetected.

While these compiler-based solutions require no application-specific knowledge, their error protection is limited to temporal and informational redundancy. Nevertheless, researchers have explored software solutions to enhance application reliability for multi-core systems, leveraging different cores for redundant program execution. The utilization of spatial redundancy enables the detection of permanent faults [61].

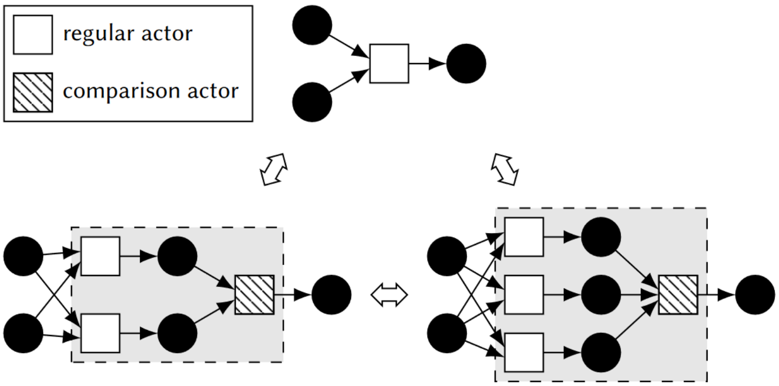


Figure 3.1: Different redundancy configurations within a dataflow graph. Data objects are marked as circles and data operations as square. On top a non-redundant execution graph, on the bottom a DMR (left) and TMR (right) configuration. (refer to [64])

For multi-core systems, various solutions are explored, with this thesis specifically concentrating on safety mechanisms at the hardware level, presenting bare-metal low-level software-based solutions. However, it is noteworthy to mention that a prevalent higher-level solution involves virtualization with hypervisors such as XtratuM, Linux-KVM, PikeOS, and Jailhous [65; 66; 67; 68; 69; 70; 71]. These hypervisors are configured with different partitions for safety-critical and non-critical applications.

Dataflow programming models or programming languages designed for parallel and stream-oriented execution are gaining attention in embedded systems [72; 73]. Researchers have explored various fault-tolerance approaches for these programming and execution models to enhance system reliability. In dataflow programming or similar concepts, problems are represented as directed graphs illustrating the relationships between data and operations on the data. Operations can range from simple arithmetic to complex data manipulations. State-of-the-art approaches leverage the dataflow concept by duplicating data operations and comparing results before committing to the final data [64; 74; 75]. For example, Figure 3.1 illustrates a redundant data flow graph, where regular actors (circles) are operations on input data, achieving spatial redundancy by executing these actors on different processing units.

Apart from the discussed techniques, various software fault tolerance approaches for mixed-criticality systems are found in the literature [76; 77]. Unlike fault tolerance mechanisms based on dataflow models, these techniques focus on task scheduling. The key difference between dataflow and task execution lies in granularity, with dataflow graphs being fine-grained and software tasks being coarse-grained, consisting of data with different lifetimes and not necessarily following a stream of data.

Recently, researchers emphasize scheduling redundant tasks to enhance reliability against transient faults. Typically utilizing N-Modular Redundancy (NMR), where each of the N copies is scheduled on different computing units, faults are detected by a voter comparing the results of each copied task [17; 78; 79; 80; 81]. Re-executing the faulty task is a common approach [76]. Current research in Mixed-Criticality Systems (MCS) not only considers scheduling redundant tasks but also explores power-aware scheduling and the re-execution of software tasks.

All these approaches share the use of a software voter to detect differences in results or computational behavior. However, the software voter itself may be susceptible to faults, resulting in misbehavior.

3.2 Hardware Fault Tolerance

Fault tolerance mechanisms and safety measures at the hardware level for multi-core devices can be categorized at different abstraction levels [82]. The lowest level pertains to the nanoscale, encompassing fault tolerance mechanisms in the physical and structural domains. In the physical domain, strategies such as hardening transistors and registers, such as radiation hardening, resilient transistors against faults, and design margin-based mitigation techniques (e.g., operating at higher supply voltages, gate sizing, body biasing, circuit guard banding) are employed [60; 83; 84; 85]. In the structural domain, hardening occurs at the circuit level or Register-Transfer Level (RTL), often utilizing redundancy (spatial, informational). Common methods involve triplication and voting, shadow registers (spatial redundancy), and ECC (informational redundancy) [58; 86].

For AMD FPGAs, tools such as TMRTool and SEM are available for controlling and mitigating faults [87; 88]. TMRTool utilizes a synthesis netlist to harden designs by triplicating design instances.

The **STRV** processor employs hardening techniques at the RTL level [89]. A fine-grained TMR technique, triplicating the circuit and flip-flops, is used to enhance system reliability, with a majority voter correcting potential faults. The **DuckCore** protects each pipeline register with ECC, introducing informational redundancy [90]. The **SHAKTI-F** processor combines the STRV and DuckCore cores [91]. ECC is used for register protection, and additionally, the data path of the ALU is safeguarded by a DMR scheme, where the circuit is duplicated, and a comparison is employed for fault detection.

All these hardware-based approaches and techniques are static and are designed during system design time. The fault tolerance mechanisms are implemented by hardware designers and remain unchanged during runtime.

On the component level, architecture features are applied to enhance system dependability and reliability. Similar to the nanoscale level, two major aspects classify the component level. The first involves hardening components against faults, typically utilizing informational redundancy and Forward Error Correction (FEC). ECC is a common hardening technique. The **LEON3FT** and **LEON4FT** use ECC to harden memory blocks [18; 92]. The protection of the LEON3FT focuses on safeguarding on-chip memory, the pipeline, register files, and cache memory against SEU. Therefore, component-level hardening is achieved through nanoscale hardening, implementing all flip-flops with TMR and safeguarded them against a single SEU [92].

The second aspect involves introducing redundancy to detect and apply fault mitigation. Various architectures are explored in literature and for COTS architectures. The following presents different approaches, from dynamic verification to full redundant Dual-Core Lockstep (DCLS) with DMR and Triple-Core Lockstep (TCLS) with TMR architectures.

Dynamic verification for processors aims to minimize complete replication of the architecture and reduce added redundancy. Instead of replicating the entire pipeline, the approach concentrates on verifying the results of a calculation. One of the pioneering approaches in this domain is **DIVA** [93]. DIVA extends the processor pipeline with a check unit before committing the result to the write-back stage. Figure 3.2 illustrates the architecture of the DIVA approach. The checker unit compares the result of the previous pipeline stages with a recalculated value. Thus, the checker unit repeats the calculation, and in the event of a mismatch, an exception is triggered. Furthermore, the communication from the processor pipeline to the register file and mem-

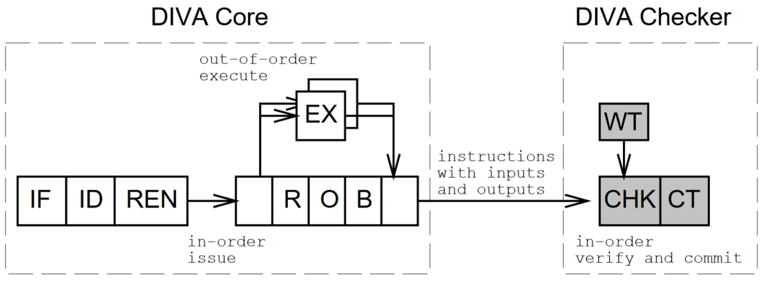


Figure 3.2: Architecture of the DIVA architecture (from [93])

ory is verified as well. DIVA does not directly check for CFEs, focusing on the detection and mitigation of SDCs. However, critical CFEs resulting in a processor hang are detected by a watchdog within the checker unit.

The **DIVA** architecture's concept is expanded and demonstrated on the LEON3 processor [94]. Instead of assuming a fully reliable checker unit, the authors flush the pipeline and re-execute the flawed instruction to enhance reliability. Another approach, Argus, utilizes the technique of dynamic verification and incorporates an additional check for CFEs [95]. However, the control flow checker requires additional pre-computed information and is not a pure runtime solution. Therefore, this approach is classified as a hybrid solution and is discussed later.

The ρ**-VEX** is an academic reconfigurable Very Long Instruction Word (VLIW) processor architecture originally developed at Delft University of Technology by Wong et al. [96]. It is configurable and extendable based on the VEX ISA with a five-stage pipeline design. The number of issue slots can be chosen (e.g., 2, 4, or 8). Each issue slot contains different functional units and is also called a pipeline. Each pipeline consists of an ALU and configurable components (e.g., multiplier, mem-

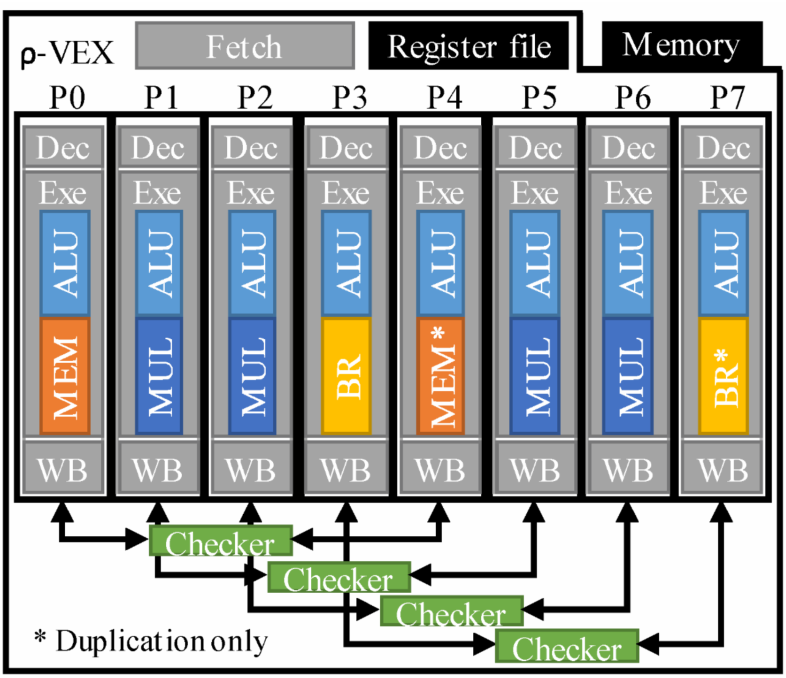


Figure 3.3: ρ-VEX Processor with adaptive ILP control to increase fault tolerance (from [97])

ory access unit, or branch control unit). The Hewlett-Packard VEX software toolchain is used to compile C code based on machine models of the ρ-VEX.

The VLIW processor design of the ρ-VEX is leveraged to explore various fault tolerance mechanisms. One such mechanism applied to the ρ-VEX architecture focuses on register protection [98]. Similar to circuit-level approaches on the nanoscale, processor flip-flops (FFs) are triplicated, and a TMR voting is employed to eliminate SEU. Importantly, this fault tolerance mechanism is configurable at runtime, making it an architectural feature that adapts the platform to the application's requirements. Non-critical applications without the need for FF protection can save energy consumption by disabling the triplicated FFs.

A fully architectural safety feature is presented by Sartor et al. [97; 99; 100]. The authors extend the ρ-VEX architecture with adaptive fault tolerance support. Pipelines are duplicated, and a pair of pipelines is always used to compare results, as shown in Figure 3.3. This architecture modification abstracts the safety mechanism, requiring no adjustment of the binary code.

The issue slots of the pipeline are capable of executing in both non-redundant and redundant modes. In the redundant mode, the pipeline and its redundant counterpart are checked to detect faults. Two strategies are presented for scheduling redundant computation. The first strategy duplicates computation when possible. Empty pipelines are utilized to execute redundant computation whenever feasible. Each time the redundant counterpart has a *NOP* operation, the *duplicate when possible* strategy duplicates the computation, and the results are compared by the *checker unit*.

VLIW compilers optimize the Instruction-level Parallelism (ILP) by reducing the number of *NOP* operations. However, the *duplicate when possible* strategy relies on the availability of free pipelines and issue slots to increase fault tolerance. The second strategy aims to increase the availability of free pipelines.

Regardless of the strategy used, the processor architecture implements a rollback mechanism to mitigate SEU.

A similar approach for VLIW architectures is investigated by Psiakis et al. [101; 102]. Similar to Sartor et al., the approach uses hardware-based rescheduling of operations within the VLIW bundle of pipeline instructions. However, the proposed hardware architecture supports more flexibility in comparison. A commit interconnect is used to freely compare results from different pipelines of the VLIW processor. Furthermore, the results of a bundle of instructions are stored for later comparison, leading to timely independent calculations of the results for voting.

The hardware architectural modification allows a more sophisticated scheduling approach of *NOP* Exploitation with Dependency Awareness (**NEDA**). NEDA exploits *NOP* instructions not only within an instruction bundle but also reschedules redundant execution over multiple points in time. It uses the flexibility to schedule redundant calculations freely within an instruction bundle and takes advantage of using a later instruction bundle. Only when not enough *NOP* instructions are available or the dependency between two instruction bundles does not allow delayed voting, NEDA inserts a complete redundant instruction bundle. Furthermore, the presented architecture can compare three results and use majority voting to correct errors.

In addition to detecting and mitigating transient faults, Psiakis et al. [102; 103] investigate detection and countermeasures against permanent faults for the VLIW architecture. The approach uses slots with *NOP* instructions to detect faults. Whenever a permanent fault of an execution unit is detected, the instruction is rescheduled to either a different pipeline or a different time slot. During execution, the proposed scheduler rebinds both the faulty and redundant instructions.

The presented approaches for VLIW architectures are designed to protect the calculation and operation of individual pipelines. However, the fault tolerance mechanisms do not target CFEs resulting from falsely fetched instructions.

DCLS and TCLS involve full architectural modifications, where the entire processor core is replicated, and checkers (DCLS) or voters (TCLS) are placed to detect faults. These voters operate on a far coarser abstraction level compared to the nanoscale level.

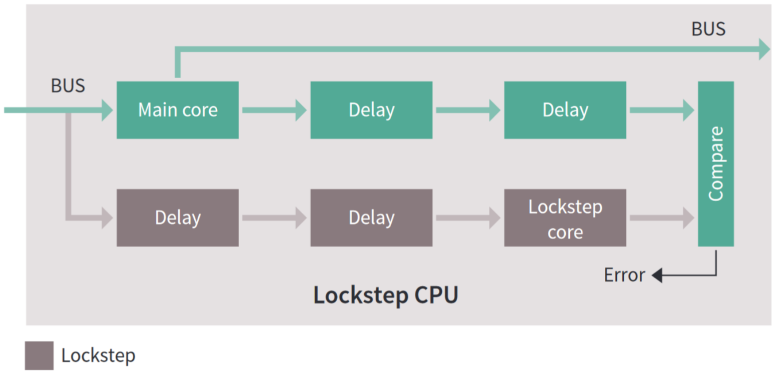


Figure 3.4: TriCore lockstep architecture (from [43])

Lockstep processors are widely used to increase system dependability and reliability. These processors execute instructions independently on two or more cores and compare the results before changing the system's state. One of the most common DCLS processor cores for automotive applications is the Infineon **TriCore** [43; 104]. The TriCore, built on a custom ISA for embedded processors, is integrated into the Infineon highly reliable AURIX processor systems (e.g., AURIX TC2xx, AURIX TC3xx, AURIX TC4xx). The lockstep architecture of the TriCore is illustrated in Figure 3.4.

Arm introduced the **Cortex-R5**, a real-time lockstep processor that incorporates split-lock technology, offering configurable redundancy for processor clusters. The split-lock technology allows users to set the redundancy of a processor cluster through a control register during boot time, configuring two physical cores to operate independently or in a DCLS configuration [105; 106]. Building on the DCLS design of the Cortex-R5, Iturbe et al. [107] presented a TCLS implementation with three physical cores grouped into one logical core. This configuration executes the same program, employing a majority voter to correct faults before they propagate to memory or I/O ports. Correctable errors detected by error detection units trigger the resynchronization logic to correct the processor state.

With the **DynamIQ Shared Unit AE** for automotive applications, Arm extended the split-lock technology to its application processors, such as the **Cortex-A65AE**, **Cortex-A76AE**, and **Cortex-A78AE** [108; 109; 110]. The DynamIQ Shared Unit AE offers three modes for a compute cluster of two, four, six, or eight cores: Split-mode, Lock-mode, and Hybrid-mode [111]. In Split-mode, all cores operate independently without redundancy. Lock-mode utilizes half of the cores for redundant execution, comparing the results of the primary and redundant cores to detect faults. In Hybrid-mode, applicable only to clusters of two or four cores, all processor cores run independently in high-performance mode, with safety mechanisms targeting the DynamIQ Shared Unit and the common control logic of the compute cluster. The configuration of any of the three modes affects all cores of the cluster and must be done after reset.

Both Infineon's TriCore and Arm's DCLS solutions rely on software-based fault recovery. CEVERO, building on the PULP platform [112], extends it with a fault tolerance module [113], implementing an DCLS architecture. The fault tolerance module halts both processor cores and rolls back to the last safe state upon detecting a fault, managing the complete recovery process in hardware and abstracting it from the program.

Shukla and Ray present a reconfigurable DCLS similar to Arm's split-lock approach for a quad-core processor platform [114]. The processor cores are custom-designed based on the RISC-V ISA [115] and implement both fault detection and fault recovery.

Ulbricht and Junchao et al. introduce a processor architecture based on **ResiliCell** [116; 117]. The platform is based on the PULP architecture using the RISC-V RI5CY core [112]. The RI5CY core is enhanced by ResiliCells to enable runtime AFT and switch between different NMR modes. ResiliCell takes inspiration from shadow registers, operating as an architectural feature. Instead of directly improving reliability, ResiliCell acts as a feature, replacing one FF with two. When the platform switches to a reliable mode, the slave FF is programmed with the value of the original FF. Each FF in the processor design is replaced with a ResiliCell. The switch between NMR modes is configured by a *HighRel Framework Controller*, which reconfigures the core based on an *SEU Monitor*, *Aging Monitor*, or an *Event Unit*. The reconfiguration process involves programming the slave FF, rerouting input signals, and enabling checker or voting logic.

Rogenmoser et al. [118; 119] introduce an on-demand redundancy approach for a multi-core cluster based on the PULP platform [112]. The architecture is extended by a *Hybrid Modular Redundancy* (HMR) module. The module controls the processing cores, detects faults, and mitigates them. The work investigates an on-demand solution for both DCLS and TCLS configurations. Similar to Arm, the redundant execution is called *lock mode*, and the non-redundant and unsafe execution is called *split mode*. Rogenmoser et al. do not change the core itself and use existing interfaces such as the debugging interface. The architecture is illustrated in Figure 3.5.

The HMR module efficiently manages comparison and fault detection. The authors explore two distinct approaches for handling detected faults. In the first method, a software solution triggers an interrupt upon fault detection. In the second approach, the HMR takes control of processor cores through the debug interface, rectifying processor register values such as Control and Status Register, Program Counter, and register file content.

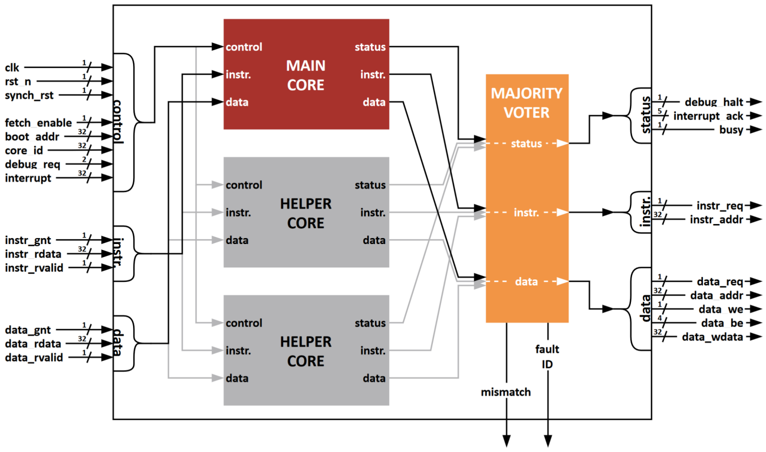


Figure 3.5: Triple-Core Lockstep configuration of the on-demand and hybrid modular redundancy (from [118])

Ulbricht and Junchao, among others, employ ResiliCells to transfer the processor state and restore the original state after completing a redundant execution. Rogenmoser et al., in contrast, unload the processor state of each core. The Control and Status Registers (CSRs) and Program Counter (PC) register, as well as the register file content, are then pushed to the stack associated with each core. The current stack pointer of each core facilitates writing the processor state to the main memory. Once all cores have offloaded the processor state to the main memory, they collectively read the state of the main core. By reading and loading the state of the master to the redundant cores, the state transfer is successfully executed. The stack pointer of each core is saved in the HMR unit and is utilized for both entering and leaving a locked configuration. Whenever a core exits the locked configuration, it restores its original state from the stack in the main memory.

While the architecture might be deemed a hybrid approach based on software routines, the realization of both state transfer and fault recovery within the HMR module classifies this work as a pure hardware solution.

The ongoing investigation has focused on fault tolerance mechanisms for hardware at a fine-grained level. However, architectural approaches allow for more flexible and coarse-grained strategies. One such approach is checkpointing, where various works exploit this technique at different levels. Koch et al. [120] introduced a checkpointing concept at the RTL level, modifying selected registers to create or rollback to a previously created checkpoint. Bourge et al. [121] adapted this concept, exploring methodologies for automatically creating checkpoints from high-level hardware descriptions, integrated into an open-source High-Level Synthesis (HLS) tool. Another hardware checkpointing approach, investigated by Kreß et al. [Kre23], utilizes more reliable Non-Volatile Memory (NVM) cells to store the checkpoint and fast CMOS FF for logic interaction. Both types of FFs are fused into a hybrid magnetic FF, representing low-level checkpointing approaches considered as general hardening methods for digital logic.

Checkpointing, as an architectural feature for processors, has been explored by various authors. **RECORD** presents a checkpointing mechanism at a register level [122]. It employs a logging approach for the register file and the PC. When a fault is detected, the approach undoes the changes and rolls back to the last logged checkpoint. The recovered checkpoint restores the register file and utilizes the saved PC.

CARER [123] and **SWICH** [46] are two similar approaches to realize checkpointing at the cache level. Both approaches employ a logging-based approach with a dual scheme. The checkpoint is stored on the same memory level as the active data, using dedicated cache lines for checkpoint data. SWICH extends CARER's checkpoint strategy to support larger checkpoint windows, simultaneously holding two checkpoints within the cache. SafetyNet is another logging approach covering cache data [124], further extended to the main memory, using additional checkpoint storage for each core in a multi-core system.

Authors have explored the use of Transactional Memory (TM) to enhance system reliability. TM logs changes and commits them as a transaction. However, transactions can be aborted, and the logged changes are undone—similar to a checkpoint and recovery mechanism. **LBRA** [125] is based on the **LogTM-SE** TM implementation [126]. The Hardware Transactional Memory (HTM) logs the changes of the leading master core and the trailing slave core. Both cores automatically create a signature over the logged data. If an error is detected, the log is traversed backward and used to restore the unmodified memory state. **FaulTM** [127] is another HTM approach, employing HTM error detection and recovery proposals based on a HTM architecture. Before committing results, the logged write-sets of two redundant cores are compared. If a fault is detected, the transaction is aborted, and the changes are undone.

A multiversioning approach based on HTM is investigated by Amslinger et al. [128; 129; 130]. This approach is designed for a system that introduces slack between two cores executing the same functionality. To fully utilize the slack, the comparison is detached from synchronization points. Multiple versions of transactions are logged and stored. The leading core continuously speculates and compares the transaction of the trailing core with its corresponding transaction version. In the event of a detected fault, both cores promptly abort the transaction.

However, the presented TM approaches predominantly explore automatic hardware solutions for fault detection and transaction aborting. Less explored is the transfer of the program state between two or multiple cores. Consequently, these approaches utilize multithreading to efficiently distribute redundant functionality.

3.3 Hybrid Hardware and Software Fault Tolerance

One prominent example of a hybrid solution is **Argus** [95], a system that integrates safety mechanisms and fault detection techniques across diverse targets. Argus addresses the four fundamental aspects of processor operations, namely control flow, dataflow, computation, and memory, employing a hybrid strategy by selecting hardware or software solutions based on the specific operation.

Argus incorporates a computation checker to identify errors in functional units. For the multiplier and divider, arithmetic codes are employed to detect faults during execution, while the ALU employs a specific checker at the logical level. The memory subsystem utilizes ECC for error detection, all of which are hardware-based solutions. However, the control flow dataflow checker employs a hybrid approach. In this instance, the hardware undergoes modification to calculate State History Signatures (SHSs) for various components. These signatures are then compared with pre-computed Dataflow and Control Signatures (DCSs) that are embedded into the program instructions. This approach in Argus shares similarities with other dynamic dataflow verification methodologies [131]. A comprehensive illustration of the Argus implementation for a simple processor core is provided in Figure 3.6.

Control flow checking is a well-established hybrid fault tolerance approach, with various techniques explored in the literature. Ragel et al. [132] and Azambuja et al. [133] introduce a hybrid error-detection

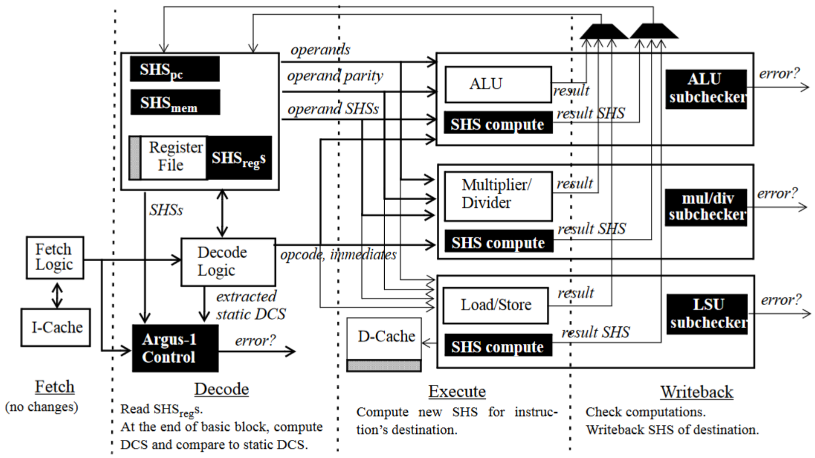


Figure 3.6: Argus-1 implementation: Dark boxes represent hardware added for Argus-1 (from [95]).

technique called **HETA**, combining software protection with an extension of the processor. These methods introduce additional instructions that control additional RTL hardware units to detect control flow errors, necessitating modifications to both hardware and software.

Hoppe et al. [134; 135; 136] propose trace-based techniques for control flow checking. Instead of introducing additional architectural features, they leverage existing processor debug and trace infrastructure such as Arm CoreSight [137] or Frontgrade Gaisler's Debug Support Unit (DSU) [18]. Trace information is then used to externally compare the control flow with a statically analyzed possible control flow graph.

Checkpoint and recovery approaches enhance the processor to create checkpoints and restore an old checkpoint when necessary. **Reli** employs a logging-based approach, implementing custom instructions to realize checkpoint and recovery functionality [138]. The software utilizes these functions to create checkpoints and recover from faults as needed.

Haas et al. [139; 140] employ transactional memories for fault tolerance, utilizing Intel's HTM system (TSX) [141] to restore the original processor state upon fault detection. The software is instrumented to create a set of transactions, comparing the set with a redundant execution using two different cores — one leading and the other trailing in time.

Watchdogs, as hardware timers initiated by software, play a crucial role in fault mitigation within a specified time interval. If the watchdog timer is not reset within the given timeframe, the hardware architecture intervenes to address the fault. In typical system behavior, a reset occurs, as watchdogs are designed to detect processor hangs when software fails to reset the timer. For multi-core processors, multilevel watchdogs can enhance system reliability [65]. **Nostradamus** demonstrated the utilization of watchdogs in out-of-order processors to detect control flow errors [142].

Chapter 4

Adaptive Redundancy

In this chapter, the concept and principles of AR, which draw inspiration from the concept of Adaptive Fault Tolerance (AFT) is introduced. AR is tailored to accommodate applications with varying requirements concerning fault detection and mitigation such as applications for Mixed Criticality System (MCS). The concept of AR is published and presented in [Kem23b]. Furthermore, the chapter discusses the conditions for applying AR to multi-core systems, specifically how the processor state can be transferred between processor cores.

4.1 The Concept of Adaptive Redundancy

4.1.1 Introducing Adaptive Redundancy

Implementing adaptive redundancy in multi-core architectures offers a broad strategy for achieving adaptive fault tolerance. Adaptive redundancy, in this context, refers to a system's ability to dynamically adjust its reliability by altering the level of redundancy it employs. This adjustment can be accomplished through software implementation or hardware means, such as a processor executing specialized software. From a reliability standpoint, hardware solutions tend to yield superior results. Nevertheless, conventional hardware solutions often lack flexibility and feature static designs, making them less suitable for mixed-criticality applications that require variable levels of reliability.

This work introduce the concept of **Adaptive Redundancy (AR)** as a dynamic system capable of adjusting its reliability by modifying the level of provided redundancy. The primary goal of AR is to achieve the required system dependability while managing the trade-off between redundancy and performance. The concept targets to implement and realize Adaptive Fault Tolerance (AFT) (refer to 2.2.6).

The adjustment of redundancy can be implemented across various domains, and one such domain involves changing the (Sphere of Replication (SoR)) (refer to 2.2.4). For instance, a system initially operating with fine-grained redundancy, such as instruction-level redundancy, can be reconfigured to adopt coarse-grained redundancy, such as core-level redundancy. The reconfiguration process is illustrated in Figure 4.1.

The illustrated example aims to demonstrate the differences between instruction-level SoR and core-level SoR, as well as the advantages of each approach. In the original instruction-level configuration, each individual instruction's result is compared before taking it into effect. However, a core-level redundancy approach, which is more coarse-grained, performs comparisons on data leaving a processor core, allowing multiple instructions to be executed between two comparisons. As a result, the comparison frequency in core-level SoR is lower compared to instruction-level SoR, providing greater flexibility in locating redundant executions across various units.

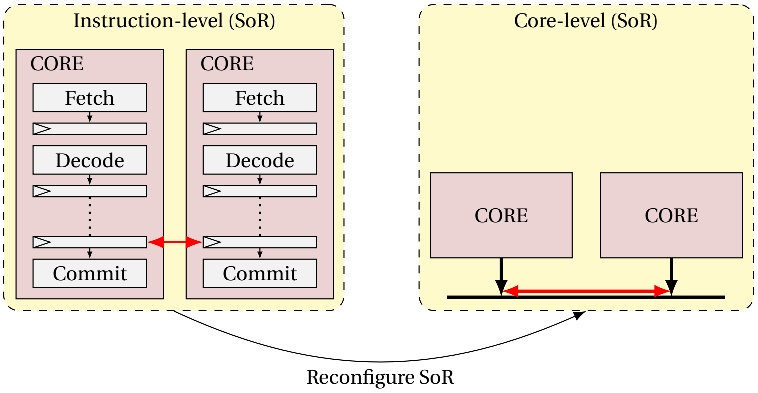


Figure 4.1: The Sphere of Replication (SoR) is reconfigured from a fine-grained instruction-level redundancy to a core-level configuration.

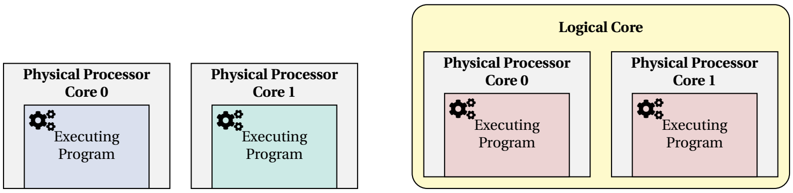
On the other hand, instruction-level compression necessitates a high frequency of compression, and any additional latency introduced significantly impacts the system's performance. In this scenario, the processor must stall until instruction comparisons are complete, resulting into a close coupling between cores and the voter to avoid performance penalties. Moreover, data required for compression must be provided immediately.

Using instruction-level SoR offers an advantage in terms of fault detection latency. The maximum detection latency is determined by the distance an instruction has to travel from the fetch stage to the voting stage (commit stage), as well as any processor stall time (e.g., cache miss). In contrast, a coarse-grained core-level SoR, due to reduced compressions, exhibits a larger fault detection latency. Faults are not detected at the instruction level; instead, they are only detected when data is leaving a processor core. The time interval during which data leaves a core is orders of magnitude larger than the time needed to process an instruction. Therefore, before data is written to a lower memory hierarchy (e.g., main memory), a fault detection mechanism is applied to ensure only flawless and valid data is written upon leaving the SoR. If faulty data is detected, a fault handling mechanism must be employed to prevent data corruption and maintain a consistent system state.

Adaptive redundancy covers another essential aspect. This covers the level of redundancy provided by the system, especially in systems with varying dependability requirements. The ability to modify the level of provided redundancy allows the system to adapt to current situations and provide necessary redundancy when required. Surplus resources from cases with lower dependability needs can be efficiently utilized for other tasks and functionalities. The system can smoothly transition between different redundancy configurations, such as from a non-redundant to a Dual Modular Redundancy (DMR) configuration, or from DMR to Triple Modular Redundancy (TMR) configuration. Such reconfigurations enable precise control of the system's reliability and dependability demands. While a system without redundancy lacks fault detection capability, employing DMR or TMR enables the identification and, in the case of TMR, even immediately correction of incorrect behavior.

4.1.2 Adaptive Redundancy for Multi-Core Processors

Figure 4.2 provides a illustration of a reconfiguration example for a processing system with two physical cores. Here, a **physical core** refers to a hardware processor core capable of independently executing program instructions (as illustrated in Figure 4.2a). When these physical processor cores are reconfigured into DMR or TMR configurations, they form a **logical core**, as shown in Figure Figure 4.2b. A logical core can comprise one or more physical cores executing the same function-


Physical Processor Core 0
Executing Program
Physical Processor Core 1
Executing Program
Logical Core
Physical Processor Core 0
Executing Program
Physical Processor Core 1
Executing Program

(a) Two physical cores independently executing two different programs (green and blue).

(b) One logical core consisting of two physical cores. Both cores execute the same program (red).

Figure 4.2: Reconfiguration from two physical cores (4.2a) into one logical core (logical)

ality or program. In the simplest case, each of the two physical cores operates independently, forming two separate logical cores (as seen in Figure 4.2a). However, in a DMR or TMR configuration, two or respectively three cores redundantly perform the same instructions. For processor cores, this means executing the same functionality, applications, or program, making them part of the same logical core since they perform identical logical tasks and are functionally equivalent. The logical core includes all physical cores associated with the DMR or TMR configuration.

The concept of AR offers a versatile approach to effectively implementing and managing system dependability. It enables the system to choose an optimal balance between reliability and performance by dynamically managing the applied adaptive and changeable redundancy at runtime. This adaptability makes AR particularly well-suited for mixed-criticality applications.

In a mixed-critical application, the system deploys various functions or tasks with different levels of criticality on a common physical platform. The system must meet the necessary application dependability requirements, including timely detection and response to system faults for critical services. The ability of AR to adjust redundancy levels at runtime ensures that resources are allocated efficiently based on the current criticality demands. During periods of lower criticality, the system can optimize performance by reducing redundancy, while during critical tasks or high-demand periods, it can increase redundancy to enhance fault tolerance and overall reliability.

To illustrate this concept, consider the mixed-criticality application illustrated in Figure 4.3. This application follows a widely adopted approach of categorizing functions into two criticality levels: low and high. Functions with low criticality (*Lo*) are executed independently on a single physical core, while functions with high criticality (*Hi*) are assigned critical sections (highlighted in red) and require redundant execution.

Non-critical functions *funcA*, *funcB*, and *funD* have a *Lo* criticality level and run in parallel on *core 0* and *core 1*. Each function is executed independently on a dedicated physical processing core. However, when a critical function (*funcC* or *funcD*) is scheduled for execution, the processor cores are reconfigured into a single logical core, combining the processing power of *core 0* and *core 1*. This logical core provides a safety mechanism for fault detection.

The given example demonstrates the usage of spatial redundancy as a method to detect permanent system faults. Spatial redundancy involves parallel execution of the same functionality on multiple independent units, whereas temporal redundancy relies on re-executing functions on the same unit at a later time.

During the execution of a critical section, the system reconfigures both independent physical cores into one logical core. In this configuration, the same code is executed on both cores, and their results are compared using a chosen SoR, such as instruction level or core level. This redundancy mechanism allows for the detection and, if neces-

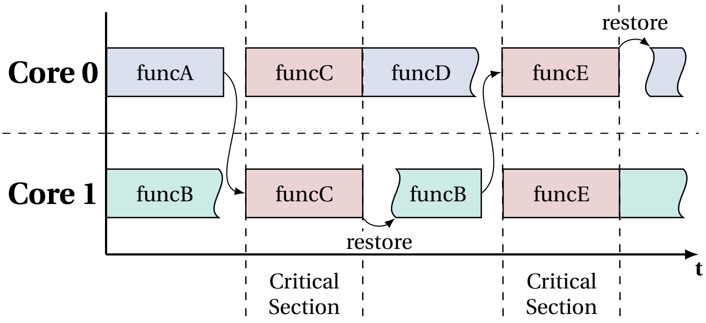


Figure 4.3: Schedule with different criticality. The critical functions, *funcC* and *funcE*, are assigned to a critical section (redundant execution), where the current program of the *slave* is suspended and the state of the *master* is taken. When leaving the critical section, the suspended program is resumed by restoring the previous state.

sary, masking of potential faults that may occur during program execution. It is important to note that a critical section may encompass a specific subsequence of program code within a function, not necessarily limited to the entire function level.

To achieve the reconfiguration of two independent physical cores into one logical core, the state of the *master* core must be replicated onto the *slave* core, ensuring that both cores behave identically during the execution of the program.

The term *master* core refers to the physical core within the logical core that provides the functionality and state of the program. On the other hand, the term *slave* core refers to the receiving core, which is used for the redundant execution of the *master* core's program. Essentially, the *slave* core takes over the program and state from the *master* core, ensuring that both cores execute the same code. By replicating the *master* core's state onto the *slave* core, the logical core achieves redundancy, effectively detecting and handling potential faults.

The concept of adaptive redundancy extends beyond MCSs and provides a broad solution for realizing Adaptive Fault Tolerance (AFT) in a wide range of systems. AFT involves a system's ability to adapt and respond to internal or external events, ensuring the system fulfills its dependability requirements. The concept of Adaptive Redundancy (AR) takes into account the need for system adaptation and encompasses a reconfiguration process that can be triggered either internally or externally. These triggers can arise from events such as program schedules, input signals, or interrupts, prompting the system to implement carry out execution of program code.

The replication process of AR for multi-core processors involves copying and moving the complete processor state of the *master* to the *slave* core. Once the state is copied, both cores execute the same code redundantly. While a *Hi* function is being executed, the original code of the *slave* core is suspended. Consequently, the processor state of the *slave* core is preserved during the state replication and restored upon leaving the logical core. Meanwhile, the *master* core continues executing the code without interruption, as its program code remains unaffected. However, the *slave* core needs to restore its original state to resume the suspended program.

As previously discussed, adaptive redundancy in multi-core architectures offers a flexible approach to achieve AFT and support MCS applications. The presented redundancy adaptations cover two different domains: one involves the change of the used Sphere of Replication (SoR), and the other addresses the system-provided level of redundancy.

The implementation and realization of the AR concept can be approached in various ways, allowing for flexibility and adaptability. The most versatile solution involves a software implementation, enabling temporal or spatial redundant execution of the application code. In this approach, the software must incorporate a software comparator or voter to detect faults, and the fault handling and potential correction are managed by the software [64; 143].

Hybrid solutions represent a mixture of software and hardware-based approaches. Here, the software leverages specific hardware safety mechanisms to detect faults. For instance, hardware-supported DCSs can be employed to identify data and control flow faults, making them a typical hybrid solution [131]. Such solutions may require the use of special instructions for fault detection during code execution, which are specific to the processor [95]. However, this implementation can introduce static overhead and performance degradation in non-critical compute sections.

From a reliability standpoint, hardware solutions generally yield more robust results. However, traditional hardware approaches tend to be less adaptable and possess a static design, which might be unsuitable for mixed-criticality applications where reliability requirements vary.

This work aims for a minimally invasive solution in which the hardware abstracts and hides most of the AR implementation. The goal is to minimize the software's responsibilities concerning AR tasks, reducing its complexity and potential points of failure. Hybrid solutions, while effective in certain scenarios, may introduce overhead and performance penalties in non-critical sections of code.

To achieve these objectives, a programming interface for applications with AR is introduced. The example interface is based on a C++ implementation and allows precise control of the used redundancy, ensuring efficient and reliable utilization of adaptive redundancy in multicore architectures. The interface provides an abstraction layer that simplifies the programming process, making it easier for developers to incorporate AR features while maintaining high levels of reliability and adaptability.

4.1.3 Programming Interface

The programming interface is constructed based on a non-intrusive programming concept, placing a strong emphasis on achieving a high level of abstraction. The underlying hardware is carefully designed to hide and abstract as much complexity as possible. Consequently, developers can direct their focus towards the application's logic and design while minimizing concerns about AFT and AR. Simple code assignments in the program result in reliable code execution with the associated reliability level. Moreover, the goal is to leverage well-established and proven toolchains without any modifications for compiling the program code.

Before delving into the specifics of the programming interfaces, let's provide a brief overview of the program generation and hardware deployment process. As illustrated in Figure 4.4, the intended procedure for developing applications with AR involves abstracting most of the logic required for AR implementation in the hardware. The concept supports minimal hardware interaction, which is abstracted and made accessible through the Hardware Abstraction Layer (HAL). Through the *AR HAL Library*, application and software developers can effectively control AR features. This library provides all the necessary functions to configure the hardware into the intended safety mode, allowing precise control over individual hardware-provided safety mechanisms, enabling or disabling them as needed.

Notably, the *AR HAL Library* requires no special instructions and is fully compatible with standard C++ compiler toolchains. By combining the *AR HAL Library* with application-specific source files, standard

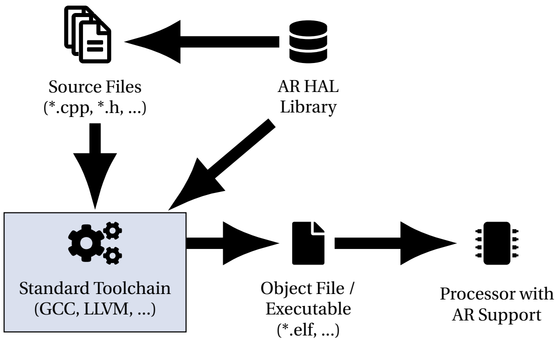


Figure 4.4: Development process for AR applications

toolchains like GCC or LLVM [144; 145] can be used to produce executable files. A widely adopted format for these files is Executable and Linkable Format (ELF) [146], which encapsulates all the required data, including executable instructions and static program data. Once the toolchain generates the ELF files, they can be readily deployed to the processing system. The processing system then executes the program code according to the software-configured redundancy. It is important to note that the software's role is solely to configure the processor, while the redundancy provision and reconfiguration process are handled by the processing system itself.

The code snippet showcased in Listing 4.1 exemplifies the non-intrusive programming paradigm for attributing criticalities to different segments of a program. With minimal modifications to the code, criticality levels can be effortlessly assigned, as demonstrated in the temporal behavior outlined in Figure 4.3. In this example, the functions *funcA*, *funcB*, and *funcC* are designated a *Lo* criticality level, while the functions *funcC* and *funcE* are ascribed a *Hi* criticality level and executed with hardware redundancy. This redundancy is achieved through runtime adaptive redundancy, wherein executing processors dynamically reconfigure their behavior. The allocated criticality levels align with the shown schedule, illustrating the seamless adaptability of the non-intrusive programming approach.

Listing 4.1: Example code which generates an schedule as shown in Figure 4.3. The funcC and funcE are assigned critical sections and are executed with hardware redudnancy.

```
template <typename T, typename P1, typename P2>
T section(T (*func)(P1, P2),
    P1 arg0,  P2 arg1, ARConf arConf){
    setARConf(arConf);
    T returnValue = func(arg0, arg1);
    setARConf(SCNon);
    return returnValue;
}

//Code Core 0
while(1){
    section(&NaviGPS, NonCritical);
    section(&ABS, Critical);
    section(&NaviRoute, NonCritical);
}

//Code Core 1
while(1){
    section(&Radio, NonCritical);
    section(&ADAS, Critical);
}
```

The redundancy management model offers a high degree of software flexibility, empowering users to individually assign criticality levels to different segments of the program. Programmers have the freedom to choose which physical cores to reconfigure into a logical core, and the core initiating the reconfiguration need not be part of the resulting logical core. To achieve this, the model defines sections without redundancy (*SCNon*) and sections with redundancy (*SC[M][S]*), where *M* designates the master core and *S* designates the slave core of the resulting logical core. In the code snippet from Listing 4.1, *SC01* configures *core 0* as the master core and *core 1* as the slave core, while the reverse configuration can be achieved with *SC10*.

The *section* function in the code snippet illustrates how to assign a criticality level to a specific function. To initiate the reconfiguration of processor cores, the function calls the *setARConf(arConf)* function from the *AR HAL Library*, thereby setting the redundancy configuration of the logical core. Upon successful execution of the section, the redundancy cluster is released with *setARConf(SCNon)*, and the logical core is dissolved, release both physical cores visible to the software. The *master* core seamlessly continues its execution, while the former *slave* core restores its original state and resumes the suspended program. This streamlined reconfiguration facilitates efficient utilization of hardware resources while upholding system reliability.

4.1.4 Deadlock prevention

The reconfiguration process, exemplified in Listing 4.1, is typically triggered by an internal event. However, AR supports reconfigurations initiated by either internal or external events, such as input signals or interrupts, ultimately leading to redundant execution of program code. Nevertheless, managing external events can raise challenges, and upon analysis, certain situations may be identified where reconfiguration could potentially result in deadlocks within the logical core, composed of the master and slave cores. These deadlocks are primar-

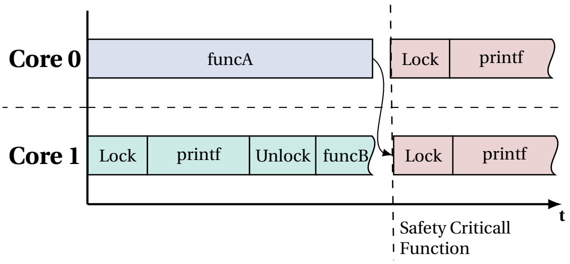


Figure 4.5: Deadlock free execution of mutex locks and unlocks of two concurrent programs.

ily associated with software-based synchronization between independent physical cores and concurrency control mechanisms.

In multi-core architectures, both synchronization and concurrency control play crucial roles in preventing race conditions and ensuring mutually exclusive access to shared resources [147; 148; 149], such as shared memory or I/O functions like *printf()*. Concurrency control is often realized through mutexes [150], a widely adopted technique in software development. When a concurrent program attempts to access a shared resource, a mutex is locked. The mutex can be successfully locked only if it is not already locked by another core or task; otherwise, the execution is suspended until the mutex is unlocked. Once the mutex is locked, the program gains exclusive access to the shared resource for its execution duration. This mechanism guarantees that only one core or program at a time can access the shared resource, ensuring data integrity and preventing conflicts among concurrent processes.

Executing code concurrently on two logically independent cores requires meticulous consideration of concurrent program execution and may necessitate the implementation of concurrency control mechanisms. Listing 4.2 presents a code snippet illustrating the use of mutexes to access shared resources in a concurrent program. In this example, *core 0* executes non-critical code, while *core 1* runs the critical function *funcC*. Both cores access a shared resource, *printf()*, and follow the mutex locking and unlocking procedure before and after accessing the shared resource. The timing behavior of executed functions can vary depending on the program's execution flow, as demonstrated in Figure 4.5, which illustrates the expected behavior with no deadlock occurrences. In this case, *core 1* unlocks the mutex before *funcA* completes, allowing the logical core to lock the mutex and continue code execution after the physical cores are reconfigured into a redundant cluster forming one logical core.

Listing 4.2: Two functions accessing a shared resource (printf). The concurrency control is manged by a software mutex.

```
int funcC() { // Safety Critical Function
    ...
    lock(mutex);
    printf("Func C"); // Shared Resource
    free(mutex);
    ...
}
... // Execution on Core 0
funcA();
section<int>(funcC); // Safety Critical Section
...

... //Execution on Core 1
lock(mutex);
printf("Function B"); // Shared Resource
free(mutex);
...
```

However, it is crucial to recognize that the timing of program execution may be influenced by various factors, including input data and the actual program process. As illustrated in Figure 4.6, slight variations in execution timing can lead to a different outcome. For instance, if *funcA* completes before *core 1* accesses the shared resource, *core 0* proceeds with reconfiguration without being aware of *core 1*'s progress. Consequently, the mutex remains locked after the reconfiguration, as *core 1* could not unlock it before the transition. This results in a deadlock, as *funcC* is unable to lock the mutex due to the suspended program of physical *core 1* not releasing it. This scenario exemplifies how a mutex and processor reconfiguration can lead to a deadlock situation.

To mitigate the risk of deadlocks as described earlier, a possible solution is to implement a strategy that prevents the reconfiguration process until the mutex is unlocked. For instance, one can disable the reconfiguration of the slave core when it locks the mutex and enable it again once the mutex is unlocked. This approach is demonstrated in Figure 4.7. By employing this strategy, deadlocks are avoided, and the system ensures that concurrent program execution and reconfiguration proceed seamlessly without interfering with each other.

Implementing a deadlock prevention strategy through scheduling turn out impractical, especially when reconfiguration is triggered by external events. In such cases, a more practical and effective solution involves inhibiting the reconfiguration process until the mutex is unlocked. This approach ensures that the physical cores do not transition into a logical core while the mutex remains locked. Consequently, the potential scenario where the slave core's program becomes suspended with the mutex locked, posing a risk of entering a deadlock state, is effectively averted.

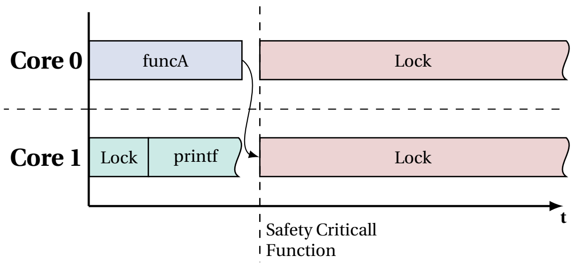


Figure 4.6: Execution with deadlock. The mutex of the non-critical function of *core 0* has not been unlocked before the reconfiguration. The *funcC* is stalling until the mutex is unlocked by the non-critical function.

By delaying reconfiguration until the mutex is unlocked, the system maintains the coherence of concurrent program execution and the reconfiguration process. This strategy guarantees that the program's synchronization and concurrency control mechanisms align correctly with the hardware-level changes in core configurations. As a result, the risk of encountering deadlocks during reconfiguration is significantly reduced.

With this deadlock prevention strategy, the processing system can confidently handle external events that trigger reconfiguration without compromising the integrity of concurrent program execution. This enables smooth coordination between mutex-based synchronization and processor reconfiguration.

To avert deadlocks in adaptive redundancy scenarios, the *AR HAL Library* provides an adaptive mutex that encapsulates standard mutex lock and unlock functions while incorporating a deadlock prevention mechanism. Listing 4.3 illustrates an exemplary implementation of this adaptive mutex, featuring the functions *lockAR* and *unlockAR*.

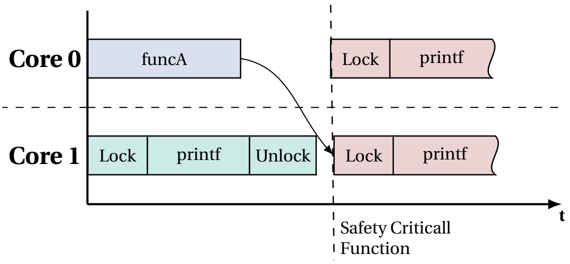


Figure 4.7: Execution without deadlock. The reconfiguration is suspended until the program of *core 0* unlocks the mutex. *core 1* stalls until the reconfiguration has been finished.

Listing 4.3: Mutex lock and unlock with deadlock prevention for runtime adaptive adaptive redundancy. The processor reconfigraution is dissabled during the lifetime of the mutex lock.

```
void lockAR(mutex m){
    dissableReconfigAR();
    lock(m);
}
void unlockAR(mutex m){
    unlock(m);
    enableReconfigAR();
}
```

When an adaptive core endeavors to acquire a lock using *lockAR*, the function initially disables the reconfiguration of the core, ensuring that the core cannot undergo reconfiguration until the lock is released. This measure guarantees that a physical *slave* core does not reach a state where the mutex is locked while reconfiguration is pending, a scenario prone to deadlocks. After acquiring the lock, the core can securely access the shared resource. Upon completion of resource access, the core releases the lock using *unlockAR*, re-enabling the reconfiguration of the physical core.

Employing these adaptive mutex functions ensures the prevention of deadlocks during concurrent program execution and the reconfiguration of adaptive redundancy cores.

4.2 Program Replication

Whenever two physical cores reconfigure into one logical core, it is crucial that both cores operate on the same program state. Consequently, the transfer of the state from the *master* to the *slave core* becomes necessary. This section consider the various program states that are relevant and require to transfer. Firstly, it addresses the distinctions between migration and replication. Subsequently, it explores the program structure and the section concludes by discussing the transfer of the program state from one core to another in a shared memory architecture.

4.2.1 Program Migration and Replication

Program migration and replication can be time-consuming tasks. In the case of software migration, a program running on one processor core is transferred to another core. On the other hand, in software replication, the program is duplicated and copied to another core. Both migration and replication processes require a consistent state of the current program to be transferred, which involves recording and copying the current state to the target processor core. This copy serves as a snapshot of the original program state at a specific point in time.

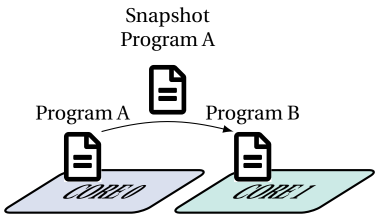
The **program state** encompasses all the information needed to reproduce the behavior accurately. After loading or restoring a snapshot, the software execution must behave identically to the original, without any differences in behavior. Although creating a snapshot may introduce some delay in code execution, it must not affect the functional behavior of the program.

The main distinction between migration and replication lies in the behavior of the original master core. In migration, the master core stops executing the original software and hands over its execution to the target core. As a result, the master core becomes available to perform other software tasks unrelated to the original program. Conversely, in program replication, the master core continues executing the original software while simultaneously running the same software on the target core. This leads to the redundant execution of the program.

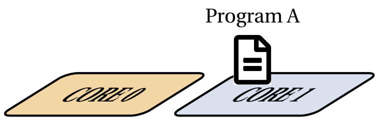
For a visual understanding of the difference between program migration and replication, refer to Figure 4.8. Both processes involve working with a snapshot of the program, which is transferred to the target core. This snapshot must contain all the necessary information for the program to run correctly on the target core. Therefore, it is crucial to comprehend how a program runs on a processor and how it is organized to ensure successful migration and replication processes.

4.2.2 Program Organization, Structure, and States

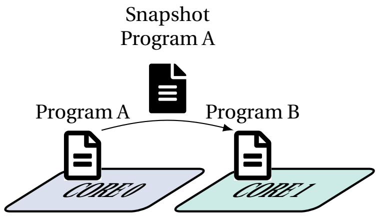
In general, software comprises a collection of computer programs designed to run on hardware components. These computer programs utilize instructions to control the hardware, performing tasks such as loading or saving variables, manipulating variable values, and controlling the flow of instructions. The set of available instructions is defined by the ISA, which serves as an abstraction layer between the



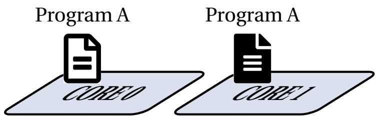
(a) Task Migration Process



(b) Result of the Task Migration Process



(c) Task Replication Process



(d) Result of the Task Replication Process

Figure 4.8: Comparison of task migration and replication.

software and the underlying hardware. It acts as the interface through which executable program interacts with the hardware. The ISA also defines the available registers, supported data types, and the used memory model. A compiler, relying on the ISA, translates the source code into individual instructions, facilitating the process of compiling software.

During the compilation process, the compiler generates more than just program instructions. It also creates additional program-related information, such as constant data and reserved memory space necessary for correct and successful program execution. Consequently, the resulting executable object file is structured into different sections, each with a specific purpose and storing various types of information.

Typically, object files define the following sections:

- Code segment or text segment
- Data segment
- Read-only data segment
- BSS segment

The code segment, also known as the text segment, contains the instructions of the executable code. Typically, this segment is intended as read-only and has a fixed size that remains constant throughout code execution.

The data segment is further divided into a modifiable area and a intended read-only area. The modifiable area holds initialized variables with a singular instance, including initialized static and global variables. On the other hand, constant values, which are immutable, are placed in the read-only section of the data segment. Uninitialized static and global data are mapped to the Block Start Symbol (BSS) section of the object file.

While the object file represents the static data of a program, its size remains fixed and does not change during execution. However, programs are composed not only of static and global data but also of dynamic data, whose size varies during runtime. Therefore, for task migration or replication, in addition to the static data, the target core must also access the dynamic data relevant to the executed program.

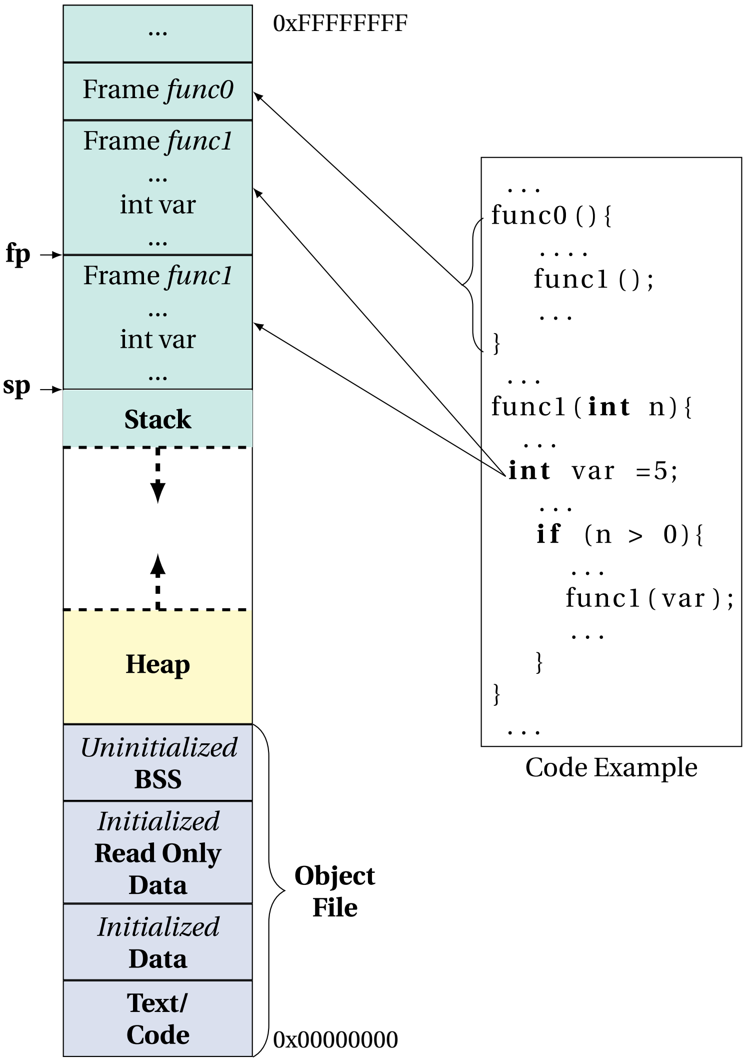
Dynamic program data comprises both stack and heap data. The stack and heap are generated and evolve during program execution. The stack dynamically grows and shrinks as the program progresses, holding function parameters and local variables. Each function call pushes the required data onto the stack and pops it upon exiting the function. The stack operates on a Last in First out (LIFO) structure, automatically managing memory allocation without explicit intervention from the programmer.

The stack size grows and shrinks over time, and objects on the stack have limited lifetimes. Larger data amounts are stored in the heap, a more complex data structure compared to the stack. Data management within the heap requires explicit control by the user software. Developers must manually allocate and free data within the heap, giving precise control over the lifetime and accessibility of individual data within the software. Unlike local variables on the stack, data residing on the heap persists and remains accessible by any part of the software.

In traditional computer systems with linear address spaces, the heap is located at the end of the BSS segment, growing towards the end of the address space. Conversely, the stack grows from the highest memory address to the heap and address zero.

As previously mentioned, the stack follows a LIFO structure. The Stack Pointer (SP) keeps track of the top of the stack and is adjusted with each data push operation. The set of values associated with a function call forms the stack frame, with the Fram Pointer (FP) marking the end of the stack frame. The stack frame can consist of the return address of the function call (caller address) or temporary values, depending on the ISA. An example layout of program memory is shown in Figure 4.9, illustrating the static object file, heap, and stack.

The program state encompasses not only the values stored in main memory but also the contents of dedicated working registers within the processing units. These registers play a crucial role in performing operations and altering the program state. Processors with ISAs featuring a register-memory architecture offer flexible operand usage, allowing operands to be either registers or memory locations. Although these processors can theoretically work directly on memory, practical



```
 ...
func0(){
    ....
    func1();
    ...
}
 ...
func1(int n){
   ...
 int var =5;
    ...
    if (n > 0){
       ...
       func1(var);
       ...
    }
}
 ...
```

Figure 4.9: Memory layout of a program. It illustrates the static data of the object file and dynamic data. The stack includes function calls of *func0* and *func1*. The local variable *var* of *func1* is placed twice on the exemplary stack.

considerations, such as differing clock speeds between processors and SRAM or DRAM-based memory [4], lead to processors primarily operating on values in registers rather than directly manipulating main memory.

Processors with a RISC ISA usually adopt a load-store architecture, which restricts memory access to specific operations. Load and store operations, specified in the ISA, facilitate data transfer between main memory and internal processor registers. Instructions involving the ALU, like *ADD* or *AND* instructions, alter state solely within the local register values. State transitions involve loading values from main memory into registers, manipulating these values, and subsequently writing the results back to main memory. Additionally, values can be loaded from or saved to I/O interfaces.

Task migration and replication must consider the state of these internal processor registers as they represent the current progress of the program. Neglecting the state of these registers could result in erroneous states during the migration or replication process.

Processor registers, including general-purpose registers, serve not only as temporary storage for intermediate calculation results but also for tracking the frame of the stack using the FP and SP pointers. Especially the SP pointer, used frequently, marks the location and end of the stack, which stores local variables as described earlier. The SP is integral to maintaining the state of the current function call.

The program flow is governed by the PC register, dictating which instruction to fetch next. By modifying the PC register, the program flow can be altered, enabling software loops, conditional statements, and conditional branches. Function calls also change the program counter, shaping the execution order of commands based on the order of PC register values. Therefore, the PC register plays a essential role in defining the current program status.

In a pipelined processor architecture, multiple state transitions can occur simultaneously due to instructions being divided into several subtasks across different pipeline stages. For example, the register access stage reads values from the register file, the memory stage accesses main memory, and the write-back stage writes to the register file. This results in multiple instructions being present in the pipeline at the same time. Consequently, both the write-back stage and the storage stage can simultaneously modify the system state. The memory stage modifies the state in main memory, while the write-back stage can simultaneously alter a value in a general-purpose register.

Hence, the program state encompasses the state stored in main memory, the current progress of the software, and the values of general-purpose registers. Accounting for these aspects is essential in achieving successful task migration and replication, ensuring accurate and seamless execution within multi-core architectures.

4.2.3 Program Migration and Replication in Shared Memory Architectures

This work focuses on program migration or replication within shared memory architectures. Specifically, it deals with scenarios where physical cores are configured into one logical core and share the same address space, as illustrated in Figure 4.10.

The shared memory architecture allows multiple cores to access the same memory. In the figure, simplified representation of two cores and a common shared memory is illustrated, but the architecture can easily scale up with more processing cores. Each of the core have access to shared memory over a interconnect like a Network-on-Chip (NoC) or bus.

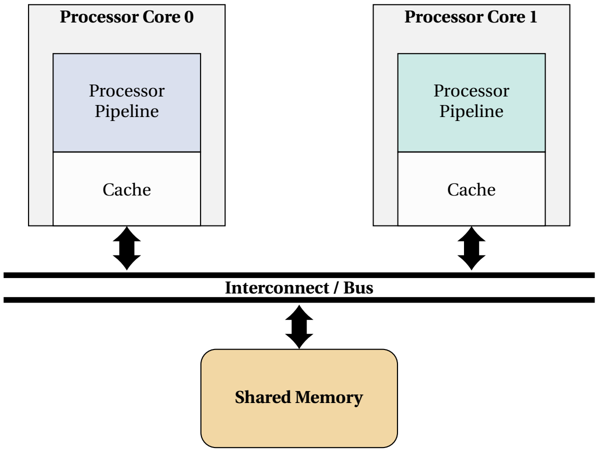


Figure 4.10: Shared memory architecture with two cores

The primary advantage of shared memory architectures lies in their ability to simplify the migration or replication process. By utilizing a common memory accessible by all cores, there is no need to copy or transfer static and dynamic data (stack and heap) from the main memory. This eliminates the overhead associated with data movement.

In contrast, a distributed memory system requires data of the main memory associated to the physical master core has to be copied or transferred to the physical slave core's memory. This data transfer can become a significant challenge, especially when dealing with a huge amount of data. However, it's important to note that not all the data in the main memory is necessary for the proper execution of replicated or migrated functions. Usually, only a subset of the dynamic data in the main memory is required for these operations, particularly for redundantly executed functions or program subsequences. The selection of the appropriate data subset is a complex task, involving filtering of the needed stack and heap data before transferring the reduced data to the slave core's memory.

The required data can be randomly fragmented through the complete memory. Furthermore, the actual data location may change over time. As heap data allocation and stack variables are time dependent.

In the shared memory architecture, the advantage lies in avoiding the data movement from main memory, as the data is already accessible to the slave core. However, the internal state of the master core must be taken over by the slave core, which includes all relevant state-holding registers. For processors, the minimum subset of these registers is defined by the ISA.

For instance, in the case of the SPARC v8 RISC ISA, the processor state registers consist of control and status registers, each defined as 32-bit registers. Below is a list of these registers:

- Program Counter (PC)
- Processor State Register (PSR)
- Window Invalid Mask (WIM)
- Trap Base Register (TBR)
- Multiply/Divide Register (Y).

In addition to the control and status registers, the SPARC v8 RISC ISA specifies a substantial number of 32-bit general-purpose registers, ranging from 40 to 520 in total. To manage such a large number of registers efficiently, they are traditionally implemented using an SRAM-based register file.

By understanding and managing these state registers appropriately during the migration or replication process, shared memory architectures can efficiently handle these tasks without the need for extensive data transfers.

4.3 Summary

This chapter introduces the concept of Adaptive Redundancy (AR) for multi-core processors, providing a means to achieve the dynamic behavior of Adaptive Fault Tolerance (AFT). AFT is a broad concept, and AR can be implemented as both software and hardware. This thesis mainly focuses on the diverse hardware realizations of the AR concept.

The AR concept introduces two distinct domains. One domain involves a change in the provided redundancy, where a system may alter the Sphere of Replication (SoR) and transition from a fine-grained to a coarse-grained configuration or vice versa. The other domain targets a change in the provided redundancy, where two physical processor cores execute non-critical software independently and critical software with a safety mechanism. To facilitate the configuration of AR, a programming interface is introduced, abstracting the reconfiguration process from the application programmer.

Whenever the AR concept is utilized to transition the processor configuration from independent to redundant execution on two or more cores, the program state needs to be replicated. Consequently, the chapter investigates the data utilized by a program and how programs are organized. Given the substantial dynamic data present in the stack and heap, this work concentrates on shared memory systems. This approach avoided the need to replicate the required data from the main memory, limiting replication to the processor state.

Chapter 5

Adaptive Lockstep Processor

This chapter introduces the concept of Adaptive Lockstep Processor (ALP), a novel architecture concept carefully designed to address and meet the challenging requirements of AR. Diverging from traditional lockstep processor architectures that relay on a static design paradigm, the presented architecture seamlessly incorporates the dynamic nature of AR as discussed in Chapter 4.

The chapter delineates the ALP concept, accompanied by a comprehensive Hardware Description Language (HDL) implementation. The subsequent section of the chapter evaluates the HDL implementation. The concept and implementations of the ALP have been previously published in [Kem21], and this chapter extends upon that publication by providing additional and more intricate details.

5.1 The Concept

This section provides the concept of the Adaptive Lockstep Processor (ALP). It begins with an introduction to the core principles underpinning the ALP, outlining its design concept tailored to fulfill the previously presented AR concept. Following the introduction of the ALP principles, a comprehensive failure mode analysis is conducted on an exemplary processor architecture, specifically the LEON3 – an open-source processor architecture widely utilized in both academia and industry. The section concludes with a conceptual description of a pipeline architecture derived from the LEON3 design, strategically investigated to realize the AR concept.

5.1.1 The Adaptive Lockstep Principle

Lockstep processors, in general, provide fault tolerance through the implementation of a SoR mechanism at either the register-level or instruction-level (as discussed in Section 2.2.4). At the register-level, each register within a pipelined processor architecture necessitates its own fast fault detection mechanism. This approach results in a significant increase in the number of required comparators and interconnections between processor cores, a complexity that escalates with the number of pipeline stages. Since each pipeline register demands at least one dedicated comparator and the data from these registers must be forwarded to the voting logic, it leads to heightened interconnection complexity and increased utilization of hardware resources.

Alternatively, replication at the instruction-level incurs slightly higher fault detection latency but offers resource advantages over the register-level SoR. In this scenario, faults are identified after a few clock cycles as the faulty data propagates through the pipeline until a comparator detects the discrepancy. Nevertheless, the detection latency remains within acceptable limits.

It is crucial to assess the impact of the fault detection mechanism on the maximum achievable clock frequency of a synchronous digital system, which depends on the critical path – the longest path between two registers consisting of a sequence of logic gates. Within the processor pipeline, a dedicated comparator stage is typically integrated to detect faults and increase system reliability.

In the architecture of the adaptive lockstep processor, a register-level SoR has been chosen, prioritizing the advantages of lower hardware expenses over the slightly higher fault detection latency. An abstraction of the employed architecture is illustrated in Figure 5.1. As previously discussed in Section 4.2.3, this choice of state replication and migration limits the AR concept to a shared memory architecture. To achieve this, each physical core associated with a *logical core* must share the same memory space. Consequently, all *logical cores* have access to the same address space, ensuring that large data transfers do not incur significant time overhead for the reconfiguration process. This is important for maintaining performance and data consistency across the cores. To facilitate this shared memory architecture, each core is connected to an interconnect leading to a shared memory architecture. Within this shared memory, the executable program, along with the static data sections (text/code, data, etc.), and dynamic data (stack and heap) are stored.

The illustrated processors consist of a pipeline unit and a cache unit. The SoR at the instruction-level covers the entire processor pipeline. This means that before any change to the system state, each executed instruction must be compared before committing to the system state.

The adaptive lockstep processor architecture aims to find an optimal balance between fault detection latency and hardware implementation expenses. Through the dynamic switching between the *split* and *lock* modes based on the application's requirements, it becomes possible to achieve the best trade-off between performance and fault tolerance. This adaptability enables the customization of the processor

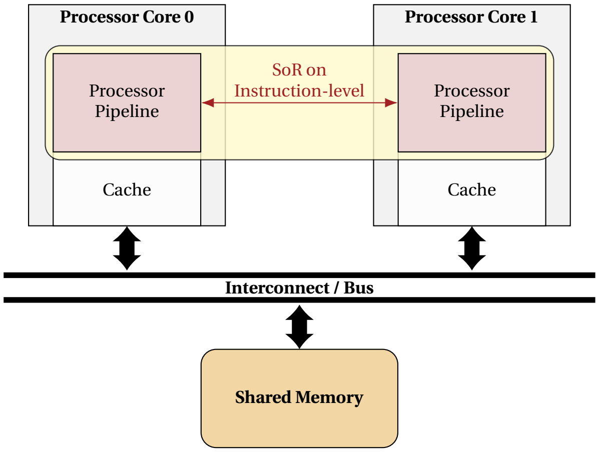


Figure 5.1: SoR of the Adaptive Lockstep Processor (ALP) on a shared memory architecture.

to meet the specific demands of AR applications, making it a promising solution for future AR systems.

5.1.2 Failure Mode Analysis

The instruction-level SoR technique aims to ensure fault tolerance by incorporating a single comparator within the pipeline architecture. This thesis focuses on applying AR techniques and mechanisms to the LEON3 processor as an example target platform. In general all presented consideration can be applied to other processor architectures. The LEON3 processor features a seven-stage pipeline architecture, comprising the following stages:

1. Fetch (FE)
2. Decode (DE)
3. Register Access (RA)

4. Execute (EX)
5. Memory (ME)
6. Exception (XC)
7. Write-back (WR)

The placement of the comparison logic requires careful consideration. In the analysis, the focus is on studying pipeline behaviors concerning faults, specifically discerning between data corruption and control flow errors. SDC poses a particular challenge as it can be difficult, and often impossible, to detect immediately. These faults may temporarily go unnoticed, lacking an immediate impact on code execution, without causing a processor hang or trap. Instead, they manifest as latent faults and may influence future system behavior. Depending on the error location, SDC faults can either negatively affect correct system behavior or be relatively uncritical. The subsequent analysis investigates how faults in the LEON3 processor's pipeline can lead to SDC.

The analysis primarily focuses on the data path since data corruption mostly results from faults within this path. While faults within the control logic could potentially lead to data path errors or execution halts, identifying a processor aborting execution and entering a hang or trap state is less challenging compared to detecting SDC. Consequently, the analysis excludes the examination of the control logic in this context.

Concerning the data path, it's essential to note that a fault propagating from the control logic to the data path or directly appearing at a data path unit is indistinguishable in terms of its impact. The LEON3 architecture's data path is illustrated in Figure 5.2, providing a representation of the relevant registers and units of each pipeline stage, along with their dependencies.

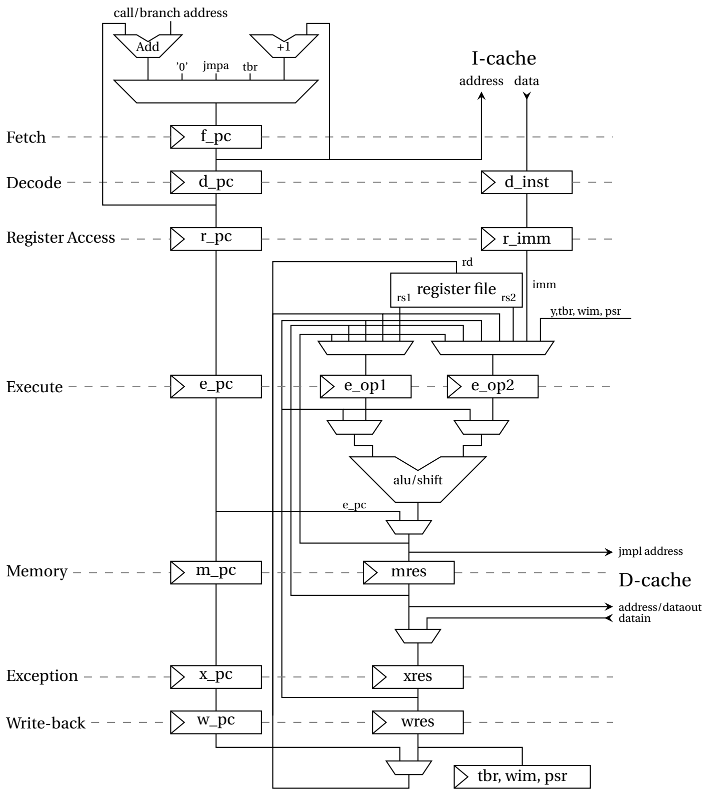


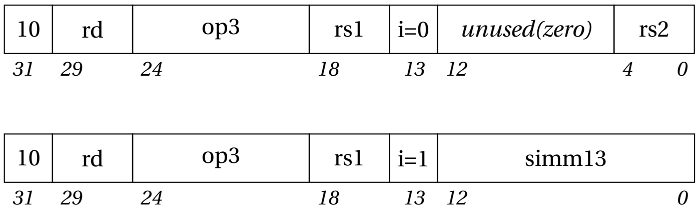
Figure 5.2: LEON3 Datapath (refer to [18])

By focusing on the data path and exploring potential fault scenarios, this analysis aims to investigate how the SoR implementation effectively mitigates and manages SDC faults in the LEON3 processor.

A fault occurring at the *fetch stage* of the pipeline can lead to either a CFE or behavior resulting in SDC, depending on the nature of the fault. If a fault occurs at this stage, an incorrect instruction may enter the processor pipeline. The consequence can manifest as either a CFE or data corruption, contingent upon the type of instruction being fetched. Control-Transfer Instruction (CTI) directly influence the control flow, and errors related to them are usually detectable at a later point and unlikely to cause SDC. However, fetching a *sub* instruction instead of an *add* instruction can lead to SDC as the operation's result would differ from the expected value.

Similarly, a fault in the *decode stage* exhibits behavior analogous to fetching a wrong instruction. This stage is responsible for generating the control signals used in subsequent pipeline stages. For instance, the Arithmetic Logic Unit (ALU) configuration for an *add* instruction is generated in this stage. Figure 5.3 illustrates how a single-bit change at the instruction level can lead to SDCs. In this example, the fault does not arise from a wrongly fetched instruction; instead, it occurs after the instruction fetch and results in a single-bit flip. This flip causes a misconfiguration of the ALU, making it perform a *sub* instruction instead of an *add*. This demonstrates how a fault in the *decode stage* can lead to undetected data corruption.

Various fault patterns within the *decode stage* can lead to SDCs. One pattern involves faults in generating control signals, directly influencing the ALU to perform addition or subtraction incorrectly. Another pattern comprise the incorrect selection of register values during decoding. Faulty target register decoding results in data corruption as the instruction's result is redirected to an unintended register, over-



opcode	op3	operation
ADD	000000	Addition
SUB	000100	Substraction

Figure 5.3: Bit structure of the *ADD* and *SUB* instruction (refer to [19]).

writing its contents. This leads to two instances of SDC: one where the original intended register does not contain the expected value, and the other where the actual unintended register does not hold its original value but is corrupted due to the overwrite.

Similarly, decoding errors in read registers directly propagate to the subsequent *register access stage*. Any error related to the *register access stage* compromises the correct selection of register values, potentially leading to selecting the wrong register or fetching a flawed value. Consequently, erroneous data is provided to the execute stage, and at least one operand used by the *execute stage* has a corrupted value, influencing potentially faulty outcomes.

Faults can occur not only during register file access but also during the result forwarding mechanism. This mechanism minimizes pipeline stalls by directly forwarding results to the *execute stage*. However, the *register access stage*, responsible for finding and forwarding operands, must execute flawlessly. Any fault in this process, including identifying the correct pipeline stage with the accurate result and ensuring its correctness during forwarding, can lead to SDC.

The critical section of the processor pipeline is the *execute stage*, where decoded instructions and provided operands are processed. The ALU performs all arithmetic and logical operations based on these inputs. Faults during these calculations result in unrecoverable errors, and the ALU produces a result either stored in a register or main memory.

The subsequent *memory stage* facilitates access to the main memory, using the memory address computed in the *execute stage*. A fault at this stage can lead to two outcomes: data corruption within the main memory or flaws in the register data. Data corruption within the main memory occurs when the processor writes data to an unintended memory address or the written data differs from the intended dataset, resulting in undetected and enduring SDC. In contrast, if a fault occurs during memory read, it impairs the internal processor state's integrity. This may involve reading incorrect data from an erroneous memory location due to a fault in memory address calculation or errors in the read data itself, leading to flawed data processing downstream in the pipeline.

Moving on to the *exception stage*, which does not handle data processing but only forwards results from the *memory stage*. Faults directly capable of introducing SDC in this stage are limited to bit flips within forwarded results. This stage primarily maintains control flow and detects traps and unusual pipeline behavior.

In contrast, the *write-back stage* manages write access to internal registers and the register file. A fault at this stage has the potential to result in SDC, impacting the internal processor state. This could manifest as corruption in the selected destination register or the register value itself.

While the analysis has primarily focused on SDC, another significant category of errors pertains to (Control Flow Errors (CFEs)). Our exploration of CFEs caused by faults begins, much like the SDC analysis, with the *fetch stage*. A fault during the *fetch stage* may impact the Program Counter (PC) register or the calculation of the Next Program Counter (NPC), directly creating a CFE. A modification in the value of the PC register has lasting consequences on subsequent PC values. Following a fault, all subsequently fetched instructions deviate from the intended control flow. Depending on the specific flawed address and the executed object file, the processor could show behaviors such as hanging, trapping, or running continuously. The behavior of the processor in such cases becomes inherently unpredictable.

A fault pattern is closely linked to the type of the fetched instruction. This becomes particularly evident when the fetched instruction falls under the category of an unintended Control-Transfer Instruction (CTI), consequently resulting in an change of the PC. This change has a lasting effect on the PC, and an analogous fault pattern emerges in cases where a potential fault materializes during the instruction encoding phase within the *decode stage*.

As illustrated in Figure 5.2, the data path of the LEON3 processor encompasses range of possibilities to manipulate the PC during the fetch stage. Crucially, the computation of the NPC depends on the specific attributes of the active CTI. This relationship is displayed in Table 5.1, outlining distinct CTI classes as defined within the SPARC v8 ISA. Notably, any CTI not depending on register values is directly executed at the decode stage. In this scenario, the immediate value of the instruction is directly forwarded for NPC calculation, a process exemplified by the Jump and Link (JMPL) and Call and Link (CALL) instructions. It is imported to note that a fault arising during the decoding of the immediate value instantaneously triggers a CFE.

Control-Transfer Instruction	**Target Address Calculation (NPC)**	**Link Register**
Branch	PC + Immediate Value	No
CALL	PC + Immediate Value	Yes
JMPL	Register Value + Register or Immediate Value	Yes
RETT	Register Value + Register or Immediate Value	No
Trap	Trap Base Address + Register Value + Register or Immediate Value	Yes

Table 5.1: Comparison of all Control-Transfer Instruction (CTI) instructions defined by the SPARC v8 ISA (refer to [19])

CALL instructions are executed unconditionally, devoid of dependencies. In contrast, a branch instruction can be either unconditional or conditional depending on the type. When dealing with conditional branches, the SPARC v8 ISA prescribes the use of ICCs within the PSR. The *decode stage* is tasked with assessing these PSRs.

The outcome of a conditional branch hinges on the interplay between the branch's conditional configuration and the value of the ICC. If the branch is taken, the *decode stage* modifies the NPC of the *fetch stage*. Thus, an undetected SDC affecting the PSR and consequently the ICC value can result in a CFE.

As demonstrated in Table 5.1, both Branch and CALL instructions are always relative to the current PC. However, JMPL and Return from Trap (RETT) instructions without exception require at least one register value to compute the target address. Consequently, as illustrated in Figure 5.2, the jump address for the NPC only becomes available post the *execute stage*.

Faults arising at the *register access stage* can trigger a CFE in instructions depend on one or two register values to determine the target PC address. Hence, a fault emerging from either the *decode stage* or the *register access stage*, resulting in erroneous register selection and thereby incorrect register values, leads to a CFE. Such faults, when they persist into the *execute stage*, ultimately yield an inaccurate target address. This erroneous NPC mistakenly directs the execution towards unintended instructions.

Given that the *execute stage* determines the target address (NPC) using the ALU, any faults arising within the ALU can skew the jump address intended for the *fetch stage*, consequently precipitating a CFE.

Up to this point, all discussed instances of CFE relate to flaws within the PC or to inaccuracies within fetched instructions. In contrast, a fault emerging during the *memory stage* does not directly influence the control flow. Nevertheless, the *memory stage* could potentially trigger a hardware trap, which is subsequently managed by the *exception stage*.

In contrast to the *fetch stage*, where the PC is automatically aligned to word size, the memory stage introduces the possibility of encountering non-word-aligned memory accesses. Each of these memory accesses has the potential to trigger a *memory address not aligned* trap. For load or store instructions, the memory address must align with the data size. For instance, a word storage operation requires a word-aligned address, which must be a multiple of the word size. In the case of LEON3 and the SPARC v8 ISA, a word consists of four bytes, making a word-aligned address a multiple of four. It is important to note that the *memory address not aligned* trap is also triggered when the calculated target address for JMPL or RETT instructions is not word-aligned.

Address misalignment can come from latent faults that result in an SDC. If this SDC leads to address misalignment, it can potentially cause further SDCs. The divergence between the used and intended addresses due to this fault propagates outward from the register level of the processor pipeline. However, if the SDC results in both misalignment and a processor trap, the latent fault becomes apparent.

The *exception stage* serves as the final pipeline stage involved in control flow. As previously described, this stage does not directly control the intended program flow, except for trap instructions. Regular instructions, when executed correctly, do not directly influence this stage. Instead, the *exception stage* manages processor-generated traps and external interrupts. Trap instructions initiate software traps, while hardware events like *window overflow*, *window underflow*, or *memory address not aligned* lead to hardware traps. When a trap, whether intentional or caused by a fault, occurs, the NPC is based on the Trap Base Register (TBR). Consequently, any fault leading to an unintentional trap results in a CFE.

Corruption of the Trap Base Register (TBR) or a fault influencing the NPC value can also lead to CFEs. A SDC affecting the TBR results in a CFE. The precise value of NPC depends on the type of trap. Hardware traps have their offset encoded in the trap source, while software-generated traps using a trap instruction calculate the offset using two register values or a register value and an immediate.

Unintentional traps have an direct impact on control flow. For instance, a *memory address not aligned* trap caused by a fault can lead to the processor being trapped, effectively making it non-operational unless the program incorporates appropriate error handling..

Certain CTIs define a link register that stores the return address (refer to Table 5.1). For example, the CALL instruction is utilized to invoke functions. This instruction redirects the PC and stores the address where the CALL instruction was executed. The current PC at the time of the CALL is stored in the link register. When the function concludes, it returns to the call position using the stored PC to jump back. In case of a SDC affecting the link register, an CFE occurs upon returning to the location of the CTI.

This analysis demonstrates that SDCs can propagate throughout the system, causing operations using corrupted data to trigger additional SDCs. The use of register values for CTI makes them vulnerable to SDC. A faulty register can result in an incorrect PC and subsequent CFE. Additionally, SDC can causing misaligned addresses that lead to hardware traps present another source of CFEs coming from SDC.

SDCs can manifest as CFEs long after their occurrence. A corrupted link register value, for instance, might be stored in the main memory's program stack. As a result, the SDC exits the processor core and persists as long as the stack frame remains valid. At a later point, the processor retrieves the corrupted link register value. This flawed value is then used to jump back to the caller's address to resume the correct program flow. However, the corrupted value leads to an incorrect target for the PC. Consequently, the ongoing program flow from this point on is faulty, and the CFE emerges from an SDC that took place much earlier.

5.1.3 Pipeline Architecture

The following section gets deeper into the evaluation and analysis of the optimal placement for the comparator unit. Building upon the insights gained from the previous section's analysis, the discourse revolves around the conceptualization of a processor pipeline architecture. This pipeline must seamlessly align with the functional requirements of the Adaptive Redundancy. Emphasizing adaptability and scalability, the design of the pipeline is of highest importance.

It is necessary for the comparator to encompass a comprehensive portion of the pipeline's logic while strategically situating itself prior to an instruction's potential impact on and modification of the system state. The previous analysis has underscored the significant risk posed by Silent Data Corruptions. These faults lead not recoverable errors and have the potential to lead to Control Flow Errors. Consequently, protection against Silent Data Corruptions takes center stage. Proactively ensuring the correctness of outcomes before committing them into the system state is necessary. Detecting any irregularities must emerge prior to the modification of a state value. This encompassing approach spans the internal processor state, inclusive of registers, as well as the external state.

Critical stages within the pipeline architecture are the *write-back stage* and the *memory stage*. In the former, instructions results are written into the register file or the processor status registers. Managing these write accesses to the internal state is the task of the *write-back stage*, while control over the interface to the external state is located in the *memory stage*. As such, ensuring the correctness of every outcome entering both the *memory stage* and the *write-back stage* is essential.

Positioning the comparator unit necessitates its placement ahead of both pipeline stages. This positioning guarantees the non-interference of faults between these stages and their subsequent effects on behavior and results. The architectural structure of the pipeline mandates comparator mechanisms preceding the *memory stage*. To comprehensively cover the pipeline logic, it is necessary to situate the comparator as close to the pipeline's tail end as possible. The *execute stage* represents the terminal phase of the LEON3 data processing. Any faults originating from previous pipeline stages, as well as the *execute* *stage* itself, propagate through to the output of this stage. All pipeline stages past the *execute stage* operate on the basis of the generated data. These pipeline stages does not operate on nor manipulate the results of *execute stage*.

Introducing a voting unit between the *execute stage* and the *memory stage* encapsulates faults across all four stages. An analysis of prior failure modes underscores the increased vulnerability of these four pipeline stages to faults, rendering them the most probable sources of SDCs or CFEs.

These stages accommodate a significant part of logic. The *decode stage* generates control signals, the *register-access stage* supply instruction execution operands, and the *execute stage* processes these operands. In contrast, logic beyond the *execute stage* is fairly streamlined, primarily concerned with forwarding control signals. These signals either direct to the cache controller, situated outside the SoR, or to the processor's register – encompassing the register file or Flip-Flop-based register.

This remaining logic escapes the scope of the discussed comparator. Consequently, traditional RTL-level methods come into play to harden the logic. These approaches encompass DMR or TMR solutions for the logic. The previously discussed comparator serves as a key architectural component in the processor pipeline, efficiently tackling the dynamic aspects of the AR concept. At the instruction level, the SoR minimizes connectivity and comparison overhead, justifying its role.

Two possible locations exist for situating the comparative logic: one involves a centralized comparator positioned outside the processor pipeline, while the other contains a decentralized voting infrastructure, with each pipeline housing its own comparator. Utilizing the decentralized comparison approach comes with a slightly extented

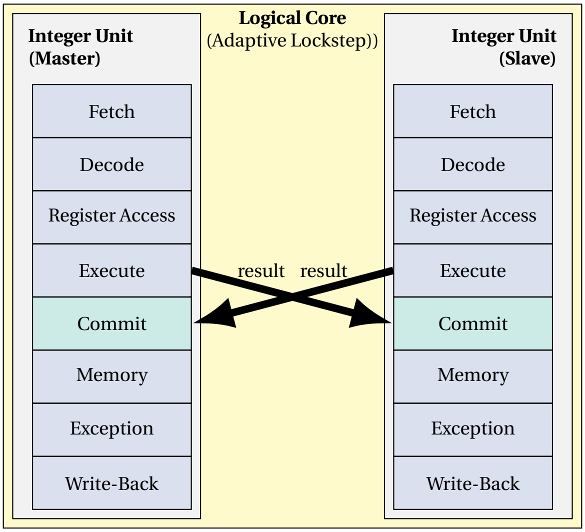


Figure 5.4: Principle pipeline architecture and concept of the ALP.

hardware cost. Nevertheless, the replication of comparator contributes to increases reliability. In the event of a fault occurring at a comparator during the comparison process, the decentralized approach ensures distinct control flows in the two pipelines, thereby making the fault easily detectable.

The results generated in the *execute stage* undergo a comparative assessment within the *commit stage*. This critical phase leverages results from both the local pipeline and its counterpart for this purpose. Consequently, each pipeline transmits its own result to its counterpart. The occurrence of a fault triggers a discrepancy between these results, signifying the need for fault handling. The *commit stage* takes charge of initiating and overseeing the requisite countermeasures to address the identified issues.

The conceptual architecture of the adaptive lockstep pipeline is illustrated in Figure 5.4. As previously discussed, the integration of comparison units is seamlessly integrated into the fabric of the pipeline stages. These comparison units find their dedicated abode within a specific pipeline stage. This strategic placement introduces an increase of required resources, while carefully mitigating the impact of comparator logic on the critical path.

Situated between the *execute stage* and the *memory stage*, the *commit stage* emerges as a crucial interface. Its foremost role encompasses the fulfillment of comparison procedures and the carefully detection of faults. Beyond this, the *commit stage* assumes the mantle of managing voting discrepancies. Furthermore, the *commit stage* takes on the responsibility of handling comparison discrepancies. This entails coordinating fault-handling mechanisms to guarantee the smooth passage of only valid instructions and outcomes to the *memory stage*. In cases where a mismatch is detected and a fault is identified, the *commit stage* reacts by annulling of further instruction execution, thereby nullifying its effects.

In Figure 5.5, the pipeline behavior is depicted in the scenarios of both normal and fault-free execution. During fault-free execution, instructions are seamlessly committed to the next pipeline stage, facilitating the alteration of the system's state. However, in the event of a detected fault, a rollback operation is triggered. Consequently, the faulty instruction is nullified, preventing its propagation to the subsequent pipeline stage. This mechanism effectively safeguards against erroneous alterations to the system state. The processor pipeline is compelled to execute a correct rollback to the last valid instruction, thereby successfully reinstating the original control flow.

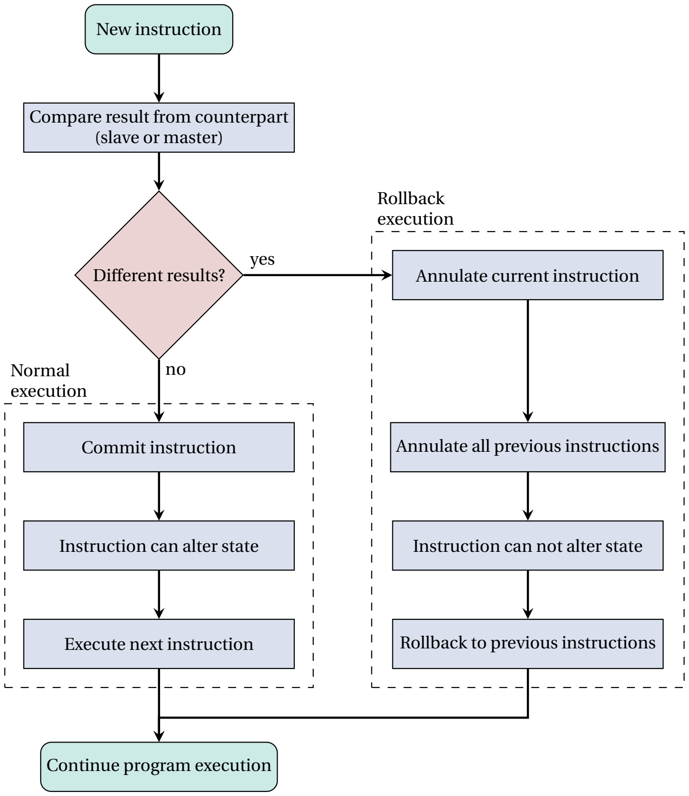

New instruction
Compare result from counterpart (slave or master)
Different results?
yes
no
Rollback execution
Annulate current instruction
Annulate all previous instructions
Instruction can not alter state
Rollback to previous instructions
Normal execution
Commit instruction
Instruction can alter state
Execute next instruction
Continue program execution

Figure 5.5: Flowchart illustrating the fault free and the rollback behavior of the pipeline.

5.2 The Processor Core Architecture

Within each processor core, a essential role is played by the Integer Unit (IU) responsible for orchestrating the execution of the program instructions. Complementing this, there exists a Level one (L1) data cache and an instruction cache. An important component is the register file further supplements the core's functionality. For a visual overview of this core structure, refer to Figure 5.6.

Both of these cores coexist within a shared memory architecture. This arrangement enables each core to access a common memory resource through an interconnect, operating with a linear address space. Importantly, the address space is identical for both cores. In the context of the LEON3 architecture, the cache leverages an Advanced Microcontroller Bus Architecture (AMBA) Advanced High-performance Bus (AHB) interface for its connection to the cache infrastructure. This interface serves as a bus accommodating multiple masters (processor cores) and slaves (memory, I/O devices, etc.).

Facilitating seamless interaction with the interconnect, both the data and instruction caches utilize a uniform interface. The design philosophy between the IU and both cache elements belong to a Harvard architecture. However, it's worth noting that a considerable portion of the LEON3 architecture employs a von Neumann architecture between the processor caches and the memory subsystem.

The cache subsystem is seamlessly integrated into the AHB system bus, establishing a connection that enables the loading and storing of data to and from lower levels of the memory hierarchy within the shared memory architecture. In this regard, the *memory stage* of the processor processes load and store instructions, leveraging the available data cache to execute these memory operations. On the other hand, the *fetch stage* accesses the instruction cache to retrieve instructions intended for execution. Unlike the *memory stage*, the *fetch stage* is not able for writing to the instruction cache. Notably, both the instruction and data caches rely on the same underlying system bus infrastructure to access the unified memory space. However, only the data cache is able to write data to the shared memory.

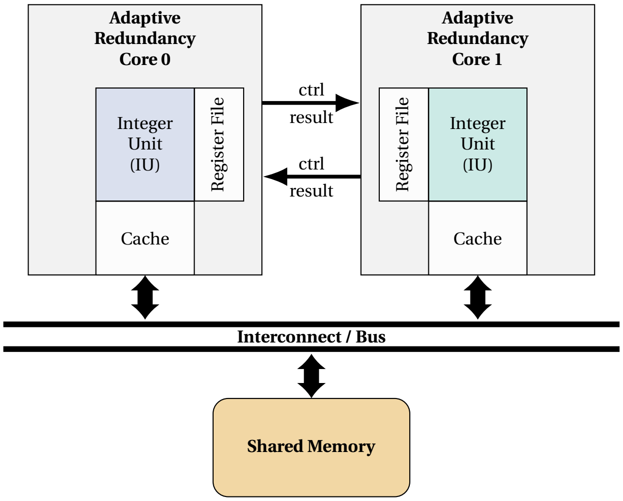


Figure 5.6: Conceptual interconnect design between two AL cores of a shared memory architecture.

The pipeline within the IU is structured to store data from general-purpose registers into the *register file*, accordingly to the specifications of the SPARC v8 ISA. The *register access stage* is tasked with reading values from the *register file*, especially when no intermediate results can be effectively forwarded. These register values are either sourced from a pipeline stage or retrieved from the *register file* to serve as operands for the *execute stage*.

The final step and pipeline stage of instruction execution, the *write-back stage* fulfills the role of writing the result into the designated destination register. The destination register, in this case, can either be an internal state register or a register within the *register file*.

In addition to the system bus, Figure 5.6 also illustrates a direct interconnect between both cores. The adaptive lockstep concept, as previously described and discussed, needs a continuous exchange of results between the two cores. Ideally, both cores should transmit results to their counterpart every clock cycle. However, due to the required low latency and the simultaneous need to provide intended services, the interconnect alone is not capable of handling these demands.

Furthermore, both *master* and *slave* cores must communicate control signals to each other to ensure correct lockstep behavior in the absence of faults. This entails the synchronization of both cores to effectively compare results of identical instructions. Moreover, these control signals play a key role in initiating reconfiguration to enter a *locked* configuration or to release it. Upon transitioning to a *locked* configuration, the *master* core's internal state needs to be shared with the *slave* core.

The adaptive nature of the AR concept avoids restricting the *master* core to a single physical core. As a result, both cores need to be adaptable to functioning either as *master* or *slave* within the *locked* configuration. This symmetry necessitates the design of identical architectures for both cores.

Inside the IU, critical processor registers like PC, WIM, TBR, and PSR are located. To ensure alignment in data access between *master* and *slave* cores, precise control over data sharing is essential. Specifically, the *master* state must be shared with the *slave* core, which encompasses registers mentioned above as well as general-purpose registers within the *register file*.

The LEON3 architecture implements these critical control registers as FF-based registers, constantly required at different pipeline stages. For instance, WIM is used to select the active registers within the register window, while PC is important for fetching instructions incrementally. These registers are accessible to additional logic in a simultaneous and uninterrupted manner.

In contrast, general-purpose registers are stored within an SRAM-based register file featuring three ports: two for read access (*rs1* and *rs2*) and one for dedicated write access (*rd*).

While FF-based registers can be directly shared from *master* to *slave* core, providing all register values from the *master* core simultaneously, transferring general-purpose registers stored in the register file is more time-consuming. This challenges the practicality of direct interconnect sharing due to latency concerns. Thus, the core architecture adopts a mechanism for sharing the register file only during *locked* execution.

During a *locked* configuration, the multiplexer shown in Figure 5.7 manages register file access for both cores. The *master* core can directly write to its associated *register file*, but any write access from the *slave* core to its *register file* is ignored and prevented.

Considering the synchronized pipelines, read access to the *register file* is propagated from *master* to *slave* core without any interruptions. Since both cores execute stages in lockstep, they work with synchronized register data and accesses. Assuming error-free execution, the results obtained by both cores will be consistent and equivalent.

In Figure 5.7, not only is the supervision of register file access evident, but also the dependency of cache access on the processor's configuration (*split* or *locked* mode) is highlighted. When in a lockstep configuration, employing two independent data caches can introduce the risk of a corrupted system state if double reads or double writes occur. This is particularly problematic for components like FIFO buffers, where each access can impact the current state. To mitigate this, in *locked* mode, only the *master's* data cache is employed. The mechanism governing access to the data cache mirrors that of the register file.

Typically, an instruction cache fetches data by reading from memory. Given that program code is static and remains unchanged over time,

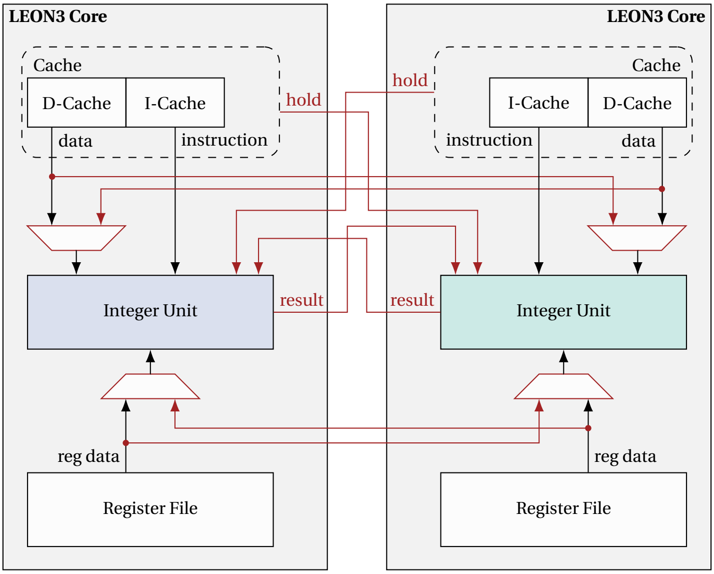


Figure 5.7: Processor configuration: Additional multiplexer and signals (red) are used for synchronization and data consistency (refer to [Kem21]).

double reads do not disturb the system. Moreover, the cache system prevents double writes in the instruction cache, as it does not support writing instruction data back.

In Figure 5.7, a processor configuration without an instruction cache selection is illustrated. *Master* and *slave* core uses their associated instruction cache both when operating independently or when operating in lockstep configuration. Input data from both the data cache (D-Cache) and the register file is determined by the processor's state.

Due to the presence of separate instruction caches (I-Cache), each IU pipeline must wait during an instruction cache miss, whether it pertains to the *master's* or the *slave's* instruction cache. This mutual waiting guarantees synchronization between both pipelines. The utilization of independent instruction caches extends the scope of the SoR beyond the pipeline itself, encompassing the level of instruction caching. In this setup, any fault within the cache can be detected by the commit stage.

Both configurations are considered. Sharing a cache system during lockstep execution has the notable advantage of reducing cache misses in the system, consequently leading to faster execution times. Moreover, no additional modifications to the cache system are necessary to counter the risks of double writes or reads of the same data. These risks, if unaddressed, can corrupt the system and disrupt data integrity. For instance, a double write to a component utilizing a FIFO can push the same data into the FIFO twice. Similarly, a double read from a FIFO could lead to either different data being stored in the data caches or, due to cache coherency mechanisms, result in data drops. In the latter scenario, data is removed from the cache before being utilized by the integer unit.

A hybrid approach, illustrated in Figure 5.7, maintains a shared data cache while exclusively allocating instruction caches to each IU. This configuration segregates the instruction cache and highlights the synchronization capabilities of the two pipelines.

During lockstep execution, both integer units operate identical software and consequently need to work with the same register data, implying utilization of the same *register file*. In this scenario, register data from the *master* unit is directly forwarded to the integer unit of the *slave*. Conversely, during separate execution, each integer unit interacts with its associated *register file* and cache resources. The selection of inputs and outputs for these components depends on the processor's configuration (*split* or *locked* mode) and the synchronization status.

5.3 The Pipeline of the Adaptive Lockstep

As previously discussed in the conceptual section (refer to Section 5.1.3), the modification of the processor design involves the incorporation of the *commit stage* as an additional pipeline stage. This additional stage, positioned between the *execution stage* and the *memory stage*, is integrated into the LEON3 original seven-stage pipeline design.

The modification of this additional pipeline stage requires modifications to the base design. In Figure 5.8, a simplified representation of the resultant data path architecture is illustrated. This illustration elucidates the *commit stage*, showcasing a modified data bypass architecture, the mechanics of context switching, and an adaptation of the *fetch stage*. Essentially, the newly introduced *commit stage* serves to accomplish the following tasks:

- Pipeline synchronization
- Processor state preservation
- Software replication
- Fault detection
- Fault handling
- Restoring the original processor state

Hereinafter, the pipeline structure is firstly discussed and afterwards all the individual tasks of the commit stage are considered.

5.3.1 The Pipeline Structure

Derived from the original pipeline design, the adaptive lockstep processor incorporates an additional *commit stage* situated between the *execute stage* and the *memory state*. Ensuring correct and flawless execution behavior requires the seamless integration of the newly introduced *commit stage* into the data path.

This integration involves adapting the operand selection for the execution stage to align with the updated pipeline structure. Consequently, the *commit stage* becomes an integral part of the data bypass architecture. To prevent RAW pipeline hazards, the results of previous instructions bypass the register file. If necessary, the bypass architecture smoothly directs the outcomes of register-manipulating instructions straight to the *execution stage*. Notably, the *commit stage* imposes no additional requirements on the data bypass mechanism.

The resulting data path architecture is illustrated in Figure 5.8, with the needed extensions and modifications for the ALP highlighted in red. Beyond the extended operand selection and data forwarding infrastructure, the *commit stage* is explicitly illustrated. This stage comprises comparators for fault detection and the necessary incoming signals, originating from the counterpart of the lockstep core. Consequently, each lockstep core forwards its current *commit stage results* to its counterpart. Both cores leverage the received results to detect faults using local comparators. Additionally, the showcased architecture encompasses two extensions.

The first extension refers to the NPC selection and instruction fetching of the *fetch stage*. The *fetch stage* is expanded to fetch instructions controlled by the *commit stage*. The *rbpc* (instruction address/PC) provided to the *fetch stage* influences the fetching of the next instruction. Depending on the fetch stage's selection, the *rbpc* value is used to retrieve the subsequent instruction, altering the control flow. The *rbpc* can be employed by the *commit stage* for either fault handling or program replication.

The task of program replication from the *master* to the *slave* involves not only precise control of the program an control flow but also the

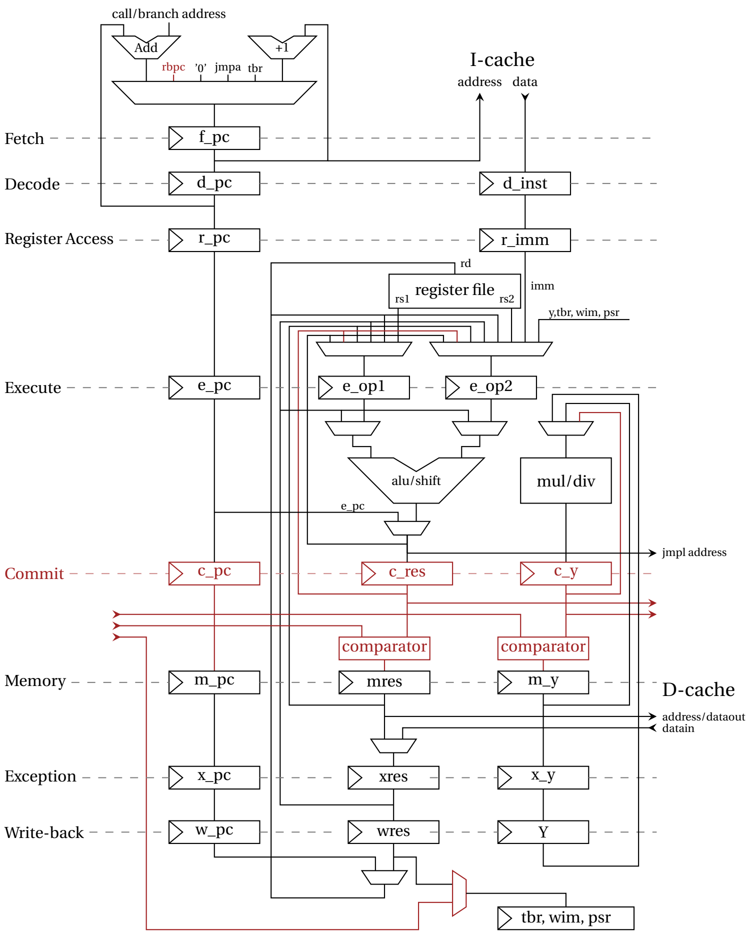


Figure 5.8: Datapath of the eight-stage-pipeline: The original seven-stage-pipeline LEON3 is extended by the commit stage (red). (refer to [Kem21])

takeover of the processor state. Thus, the *write-back stage* manages the write access of the TBR, WIM, and PSR registers. The result of an instruction is written to these registers, or the internal processor state of the *master* is assumed by the slave core.

The comparators within the *commit stage* only come into play when the pipeline operates in lockstep mode and the *locked* configuration is activated. In this mode, all data after the *commit stage* remains error-free, guaranteed by the comparison of *master* and *slave results* in the *commit stage*. Valid data proceeds from the *commit stage* to the *memory stage*, where it is either written to the data cache or directed to a register in the *write-back stage*. If an error is detected, fault handling is triggered, discarding both the result and the instruction. Importantly, the flawed instruction leaves system states, both processor and system states beyond the SoR, unchanged.

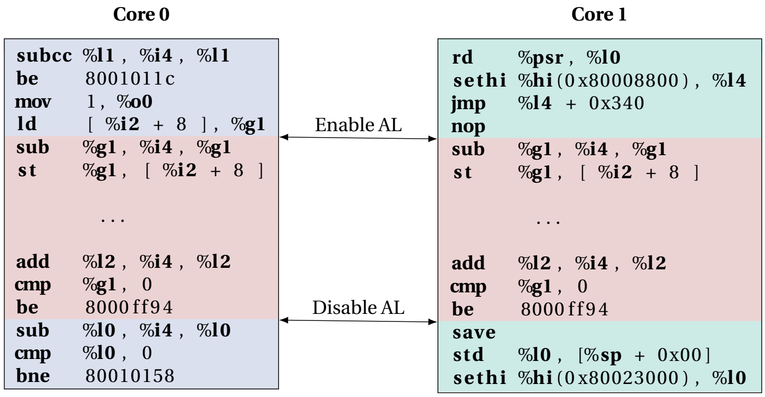
The *memory stage*, *exception stage*, and *write-back stage* comprise, as previously discussed, compact logic circuits, easily protected and hardened with a reasonable hardware overhead, such as TMR on RTL. Assumptions include fault-free memories in the system, and additional protection through ECC for the register file, caches, and system memory (e.g., Double Data Rate (DDR) DRAM). ECC protection is a common practice in safety-critical systems, offering detection of up to two errors and correction of one memory error.

Memory operations involving read and write activities affect the data cache. The data cache manages access to the AMBA AHB system bus and the shared memory at the lower level of the memory hierarchy. The introduction of the *commit stage* introduces a one-pipeline-stage delay to the original timing of data cache access. However, the load-store architecture, coupled with modifications to the data bypass, ensures that a delayed data load does not significantly impact the execution behavior of an instruction. Aside from these bypass adjustments and the delayed cache access, no further modifications are necessary. The cache interface and data access methodology remain unchanged through these modifications, making the data cache independent to the introduced changes.

5.3.2 Pipeline Synchronization

The principal and concept of the adaptive lockstep processor lies in the redundant execution of instructions, utilizing the principle of adaptive redundancy. Adaptive redundancy involves the ability to reconfigure two independent *physical cores* into one *logical core* following the occurrence of an external or internal event. In the case of the adaptive lockstep processor, this requires replicating the state of the *master core* onto the *slave core*. Both cores then execute each instruction in lockstep, necessitating the simultaneous execution of identical instructions by the *master* and *slave cores* within the *logical core*.

Figure 5.9 illustrates two *physical processor* cores and the corresponding executed instruction traces. The blue-highlighted instructions are associated with *Program A* of *Core 0*, while the green-highlighted instructions belong to *Program B* of *Core 1*. An event triggers reconfiguration, activating adaptive lockstepping. The AL enabling configures both *Core 0* and *Core 1* into a locked configuration, with *Core 0* serving as the *master* and *Core 1* replicating the state from the *master core*. During the locked mode, both cores execute the same instruction, highlighted in red, and this synchronized execution terminates when the *lock* configuration is released. A lockstep execution is terminated upon the disable AL event. Following this event, the *master core* (*Core 0*) continues its execution, and the *slave core* (*Core 1*) restores to its original execution state



```
subcc %l1 , %i4 , %l1
be    8001011c
mov   1 , %o0
ld    [ %i2 + 8 ] , %g1
sub   %g1 , %i4 , %g1
st    %g1 , [ %i2 + 8 ]

        ...

add   %l2 , %i4 , %l2
cmp   %g1 , 0
be    8000ff94
sub   %l0 , %i4 , %l0
cmp   %l0 , 0
bne   80010158
```

```
rd    %psr , %l0
sethi %hi(0x80008800) , %l4
jmp   %l4 + 0x340
nop
sub   %g1 , %i4 , %g1
st    %g1 , [ %i2 + 8 ]

        ...

add   %l2 , %i4 , %l2
cmp   %g1 , 0
be    8000ff94
save
std   %l0 , [%sp + 0x00]
sethi %hi(0x80023000) , %l0
```

Figure 5.9: Synchronization of instruction on the *master* and *slave core*.

To achieve the illustrated behavior in Figure 5.9, synchronization of both pipelines is needed. This synchronization involves fetching identical instructions simultaneously, resulting in identical execution on both cores. Each instruction from the *master* and *slave cores* progresses through its respective pipeline. The identical processor state in both pipelines ensures uniform behavior, allowing instructions to reach the *commit stage* concurrently.

Control over the pipeline synchronization of the two pipelines is allocated in the *commit stage*, with the replication process being a critical element of this synchronization.

The pipeline synchronization serves two primary goals. Firstly, it aims for simultaneous instruction execution, as exemplified in Figure 5.9. Secondly, it manages to provide a consistent processor state for state replication. Therefore, no intermediate results should be present in the pipeline; they must either proceed through the remaining pipeline stages or be flushed and nullified. Consistency is achieved when no valid instruction is present in the complete processor pipeline, indicating a flush with no further changes to the processor state. This task is essential during the pipeline synchronization phase, ensuring only a stable and consistent processor state is transferable from the master to the slave core.

The pipeline synchronization guarantees the lockstep behavior, enabling simultaneous fetching and execution of instructions while ensuring a transferable processor state. Executing the same instruction simultaneously on two integer units necessitates synchronized pipelines. It is essential to note that this work focuses on the adaptive lockstep architecture and does not investigates safeguarding against common mode faults in processor architectures. Techniques to prevent such faults, common in state-of-the-art lockstep architectures, include delayed execution, where a delay is introduced between the execution of the same instruction on different execution units. This serves as a preventive mechanism for common mode faults, and additional delay pipeline stages can be incorporated into the master and slave pipeline architecture, resulting in different execution times.

The synchronization process within the adaptive lockstep architecture involves simultaneous instruction fetching and state replication on the *slave core.* The state replication process on the *slave core* encompasses not only the takeover of the state but also the preservation of the current processor state for subsequent program continuation. Therefore, the *slave core* must achieve a consistent state to save it for later program resumption.

The *master* pipeline provides both the necessary program counters (PC and NPC) and the processor state (TBR, WIM, and PSR) to the *slave core.* Concurrently, the slave core preserves its own state for a later resumption. The slave core ensures that it takes over the *master's* state only after preserving its original state and copying the state of the *master* IU.

Preservation of the original state is accomplished by maintaining two sets of status registers in the *write-back stage*. The selection of the status register set depends on the synchronization state and is controlled by the *commit stage*. These sets include special registers like the Processor State Register (PSR). The PC and NPC of the current program state are saved to enable the continuation of software execution after releasing the lockstep cluster. Both PC and NPC are essential for ensuring a correct program resumption, especially considering that the NPC is needed to accurately restore the behaviors of Delayed Control-Transfer Instruction (DCTI). DCTI can alter the control flow after a CTI, and each CTI may have a CTI in its delay slot. To address this behavior, both the PC and NPC are necessary.

The AL core implements a Finite State Machine (FSM) to orchestrate the reconfiguration from *split* to *locked* or from *locked* to *split* configuration. The state machine is illustrated in Figure 5.10, defining four states: *split*, *sync*, *locked*, and *restore*.

In *split mode*, both processor cores operate independently in the *split* state, executing their respective programs and instructions.

The activation of the lockstep event (*lockstepEn = '1'*) triggers a state transition from the *split state* to the synchronization state (*sync*). The synchronization state is dedicated to performing the tasks of state replication and simultaneous instruction execution, as introduced earlier in this section. All synchronization tasks described in this section are carried out in the *sync* state.

Once both pipelines are synchronized, a transition from *sync* to the lockstep state (*locked*) takes place. In this state, both pipelines execute instructions simultaneously, ensuring and maintaining synchronization. This is achieved through measures detailed in Section 5.2, which describes the core architecture. Notably, the forwarding of the cache-generated *hold* to each other is important for maintaining pipeline co-

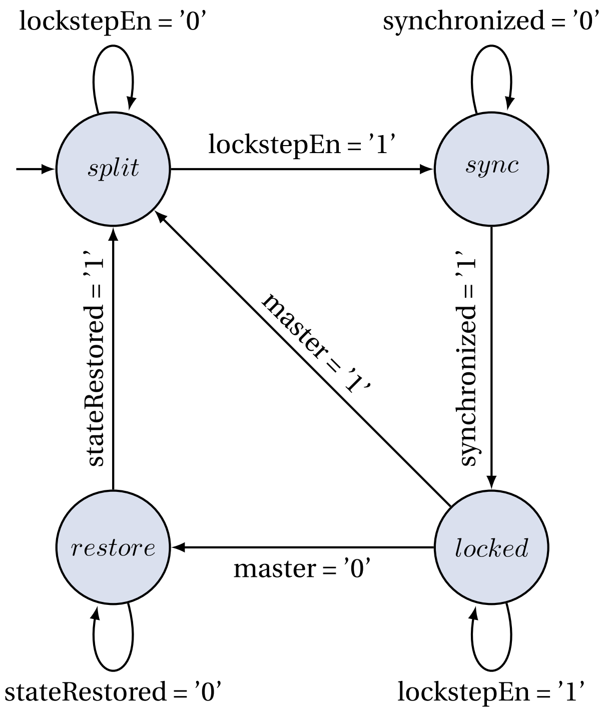


Figure 5.10: FSM of the synchronization process to enter and leave the adaptive lock configuration

herence and synchronization. Any variation in results at the *commit stage* triggers a similar fault handling mechanism in both pipelines.

The reconfiguration from *locked* into *split* depends on the core type (*master* or *slave core*). For the *master core*, the reconfiguration has no effect on the processor state or instruction fetching. The *master core* can seamlessly continue program execution, but without comparing results from the *slave core* at the *commit stage*. Therefore, the *master core*, indicated by *master = ’1’*, can directly transition from the *locked* state to the *split* state.

To leave the lockstep configuration, the *slave* must initially restore its original state. The mechanism for restoring the original state is similar to the process of replicating the state from the master to the *slave*. However, during this re-establishment, the *slave* utilizes its previously preserved state from the *sync* phase, instead of the *master's* state. Unlike the *sync* phase, where the original state is preserved, during the *restore* phase, the re-establishment of the original state can waive the current processor state. In the *restore* state, only the old processor state is recovered, and the current state is discarded. The control registers of the *slave cores* are restored to their original values in the *restore* state.

In the *locked* state, the *slave* receives data and operates on the *master's* register file and data cache. The *restore* phase is responsible to discontinue this access and terminate the redirection. When the *slave core* reaches the *split* state, it operates entirely independently of the *master core*. The two physical cores no longer function as a single *logical core*.

Both the *sync* and *restore* states need to control the execution and fetching of instructions. The following discussion primarily focuses on the *sync* phase, investigating how the *master* and *slave cores* synchronize their executed instructions and reach a consistent state. This consistent state of the *slave* is essential for resuming the program later, and the *master* requires a consistent state for replication. After examining the *sync* phase, the *restore* phase is discussed.

In Figure 5.11, the synchronization of a *master core* (*Core 0*) and a *slave core* (*Core 1*) is illustrated. The program executed on *Core 0* needs to be replicated to *Core 1*. Initially, both physical cores operate independently as two *logical cores* in a *split* configuration. Enabling the lockstep configuration triggers reconfiguration from the *split* into the

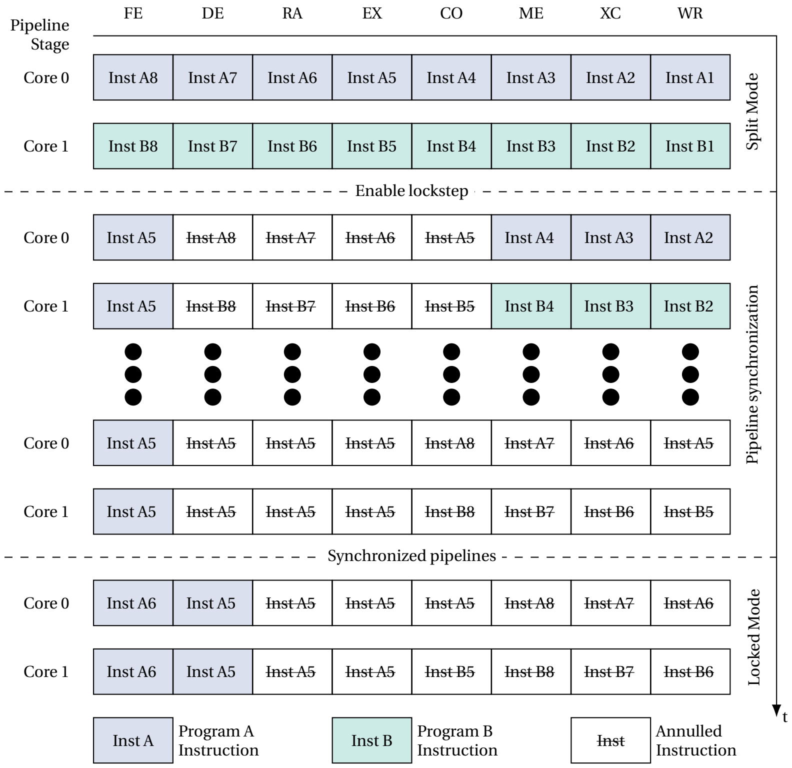


Figure 5.11: Pipeline synchronization

locked state and mode. Both cores undergo a transition from the *split* to the *sync* state, initiating the synchronization phase.

The pipeline must reach a consistent processor state that is either transferable or can be used for restoration and later program continuation. Achieving a consistent state involves flushing the entire pipeline to ensure no further modifications to the processor state. To prevent data corruption, only the instructions between the *fetch stage* and the *commit stage* are annulled. Only these stages, along with instructions in the *commit stage* and all instructions in previous pipeline stages, of both the *master* and *slave* pipelines are flushed. All instructions that pass the *commit stage* proceed through the rest of the pipeline. The instruction at the *memory stage* is complete and undergoes the remaining pipeline stages until its result is written back. Once the last remaining instruction is completed, the processor pipeline has written back all remaining instructions at the *write-back stage*. At this point, the complete pipeline is empty, as all newly fetched instructions are annulled. The entirely flushed pipeline has reached a consistent state, making both processor pipelines ready to replicate the state from the *master* to the *slave* and save the *slave state*.

In the *sync* state, the *slave* saves the current control flow when entering the replication process. It identifies the current PC and NPC. The master's *commit stage* starts searching for the PC from the *execution stage* to the *fetch stage*. Both PC and NPCs are essential for correct program flow. The saved PC and NPC are forwarded from the *master* to the *slave core*. Both cores simultaneously begin fetching the same instruction from the shared and common PC address. The *commit stage* of both cores uses the *rbpc* signal to provide the PCs and configures the associated *fetch stage* to fetch the next instruction from the *rbpc* address (PC followed by NPCs).

The following section delves into the latency involved in transitioning from a *split* to a *locked* configuration. This latency is directly influenced by the instructions yet to be executed on both the *master* and *slave cores*. The comprehensive pipeline flush latency is quantified through the model presented in Equation 5.1.

$$L_{IU,S->L} = L_{Commit}(Inst_{Type}) + L_{Pipeline,S->L} + L_{Cache} \tag{5.1}$$

$L_{IU,S->L}$ represents the latency, measured in clock cycles, required for the remaining instructions to reach completion. The LEON3 architecture executes load and store instructions as multi-cycle instructions with distinct address and data phases. This behavior is donated by the model $L_{Commit}(Inst_{Type})$, considering the latency of the instruction to complete in the *memory stage*. Additionally, $L_{Pipeline,S->L}$ denotes the time the last valid instruction takes to pass through the remaining pipeline stages. The memory access involves a private data cache interaction, and the associated latency introduced by the cache access is denoted by L_{Cache}.

In Table 5.2, the required clock cycles for different instructions are delineated. Memory instructions, depending on their type, necessitate between one and three cycles. For store instructions, the process is split into an address phase followed by the data phase. In the case of a double load or store, two values are read from or written to the main memory. For store instructions, two general-purpose registers are accessed sequentially, while a double load sequentially writes two values into the register file.

Instruction	**Cycles**
Arithmetic operation	1
Single load	1
Double load	2
Single store	2
Double store	3

Table 5.2: Number of cycles a multi cycle instructions requires. (Refer to [18])

$$L_{Commit}(Inst_{Type}) = lockUpMultiCycles(Inst_{Type}) \tag{5.2}$$

$L_{Commit}(Inst_{Type})$ calculates latency based on the number of cycles an instruction requires to complete memory access. The function $lockUpMultiCycles()$ utilizes the cycle information provided in Table 5.2.

$$L_{Pipeline,S->L} = \begin{cases} 3, & \text{Valid Instruction } \textit{memory} \\ 2, & \text{Valid Instruction } \textit{exception} \\ 1, & \text{Valid Instruction } \textit{write-back} \\ 0, & \text{Otherwise} \end{cases} \tag{5.3}$$

The pipeline latency, denoted as $L_{Pipeline,S->L}$, is dependent on the remaining instructions within the pipeline that are awaiting completion. $L_{Pipeline,S->L}$ quantifies the duration for which the last valid instruction remains unfinished. A valid instruction is one that has not been annulled within the pipeline. The pipeline is considered completely flushed when the last non-annulled instruction reaches completion.

$$L_{Cache} = \begin{cases} 0, & \text{Cache Hit} \\ 5 + L_{Memory} + L_{Access}, & \text{Cache Miss} \end{cases} \tag{5.4}$$

The latency of the cache, denoted as L_{Cache}, depends on the presence or absence of data within the cache. If a cache hit occurs, and the data is already present in the cache, the introduced latency is zero. Conversely, in the event of a cache miss, where the data needs to be loaded from the main memory, certain considerations come into play.

The configured cache in this context has a line size of four words, implying that each cache miss loads four words from the main memory. To retrieve any word from the main memory, the AHB protocol specifies an address phase of one clock cycle, followed by the response containing the data. The response time is contingent on two factors: the size of the transaction and the fact that each word requires one cycle. Therefore, for four words, this process encompasses four cycles, plus one address cycle, summing up to a fixed latency of five cycles.

Beyond this fixed latency, additional latency, denoted as L_{Memory}, can be introduced by the memory access. This represents the time the memory requires to respond correctly to the read request and deliver the data. Moreover, the cache latency is influenced by the latency associated with gaining bus access, expressed as L_{Access}. This particular latency is contingent upon simultaneous bus transfers and the number of bus masters, with the AHB bus ultimately determining access to the memory subsystem.

$$L_{ReConfig,S->L} = max(L_{IU,master,S->L}, L_{IU,slave,S->L}) \tag{5.5}$$

The reconfiguration processor of the master and the slave core is simultaneously and starts at the same time. Hence, the reconfiguration latency $L_{ReConfig,S->L}$ for configuring two independent physical cores into one logical lockstep core is determined by the longer of the two $L_{IU,S->L}$ latencies from both pipelines.

The following section covers the details of reconfiguring from a *split* to a *locked* configuration. As previously discussed, it is essential to differentiate between the *master* and *slave cores* of the *logical core.* The physical *master core* can seamlessly switch from the *locked* state to the *split* state without requiring further action. The pipeline can smoothly resume execution, as illustrated in Figure 5.12.

However, the behavior of the *slave core* is distinct. Unlike the *master core,* the *slave core* cannot directly transition from the *locked* to the *split* state. Continuing the original slave program involves restoring the initial status registers before executing the original program code.

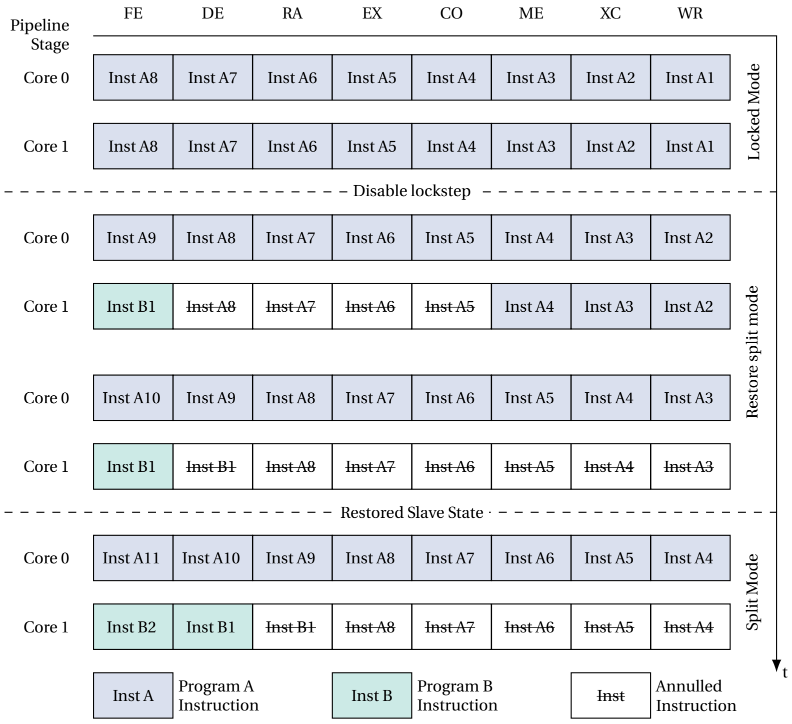


Figure 5.12: Pipeline restore

To restore the original state, the entire pipeline must be flushed. The *restore* state is employed to flush the pipeline and revert to the original state. Upon entering the *restore* state, all instructions, except the one at the memory stage, are immediately annulled and flushed. To ensure correct system behavior, the memory stage instruction is completed and subsequently discarded.

Once the pipeline is completely flushed, the *slave core* can return to its original state by utilizing the preserved control register at the write-back stage. To resume the original program, the commit stage leverages the *rbpc*. This stage controls the fetch stage to retrieve instructions from the saved PC and NPC. By fetching these instructions, the correct control flow is restored, and the original program resumes. After successfully restoring the original processor state, the former *slave core* transitions back into the *split* state.

The latency of reconfiguration from a lockstep configuration to a split configuration depends on the type of core. $L_{IU,L->S,master}$ represents the latency the *master core* requires for reconfiguration, and $L_{IU,L->S,slave}$ denotes the latency for the *slave core*.

$$L_{IU,L->S,master} = 0 \tag{5.6}$$

The master core undergoes a direct and seamless reconfiguration, smoothly transitioning into the *split* state. This process involves no alterations to the control or program flow, resulting in a reconfiguration latency $L_{IU,L->S,master}$ of zero.

$$L_{IU,L->S,slave} = L_{Commit}(Inst_{Type}) + L_{Cache} \tag{5.7}$$

The slave core completes the instruction involving memory access, a process influenced by both $L_{Commit}(Inst_{Type})$ (refer to Equation 5.2) and L_{Cache} (refer to Equation 5.4). Unlike Equation 5.1, $L_{IU,L->S,slave}$ does not account for the completion of results; although the memory access is finished, the results are discarded.

The reconfiguration from a *locked* to a *split* configuration occurs independently for the former master and slave cores. Consequently, the reconfiguration latency for the slave core is only dependent on its individual state.

impl	ver	icc	reserved	EC	EF	PIL	S	PS	ET	CWP
31:28	*27:24*	*23:20*	*19:14*	*13*	*12*	*11:8*	*7*	*6*	*5*	*4:0*

Figure 5.13: Structur of Processor State Register (PSR) (refer to [19])

5.3.3 Deadlock prevention

The AR concept enables the just-in-time reconfiguration of two processor cores. As discussed in detail in Section 4.1.4, adaptive reconfiguration has the potential to induce a deadlock, allowing code execution on both the *master* and *slave cores* simultaneously. To prevent this issue, a deadlock prevention mechanism is introduced, disabling reconfiguration as long as there is a risk of deadlock. The software controls the reconfiguration prevention of the processor core, and the pipeline design must account for this deadlock prevention mechanism.

The SPARC v8 ISA prescribes the following instruction for accessing the PSR:

- **wr rs1**, **rs2 or imm**, %**psr** *;(write psr register)*
- **rd** %**psr**, **rd** *;(read psr register)*.

The Processor State Register (PSR), as illustrated in Figure 5.13, includes a reserved segment. This reserved area in the PSR field is used for storing and managing the reconfiguration capability of the pipeline. In Figure 5.15, the extended PSR field is illustrated. The value stored in the Dissable Reconfigration (DR) field determines whether the pipeline's reconfigurability is enabled or disabled. When the DR field's bit is set, the core cannot reconfigure and switch into the *sync* state.

The originally presented state machine for reconfiguration is expanded to incorporate deadlock prevention, resulting in the FSM illustrated in Figure 5.14. When the DR field is set (*disableReconfig* = *'1'*), the

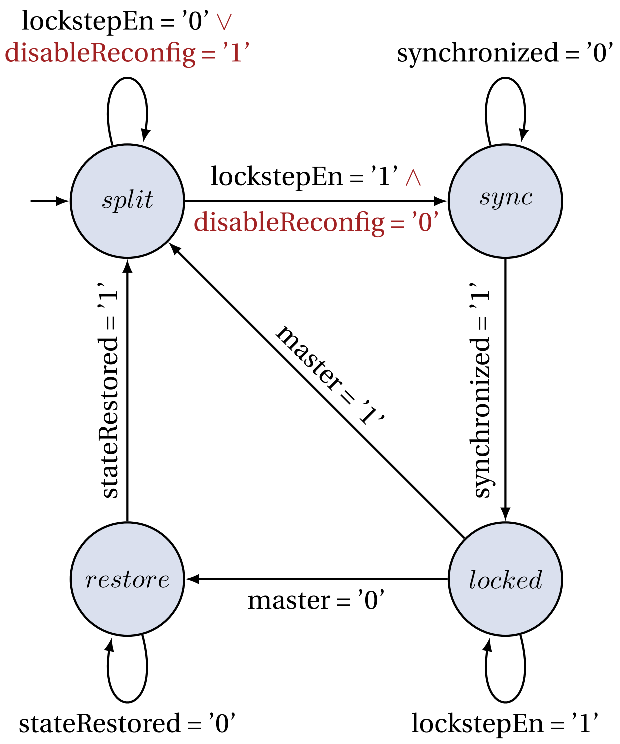


Figure 5.14: FSM of the synchronization process to enter and leave the adaptive lock configuration with deadlock prevention mechanism (red).

pipeline is prevented from transitioning to the *sync* state. This ensures the continuation of the original code execution as long as the DR field is set. Once the software changes the DR field to '0', the pipeline becomes eligible for reconfiguration.

The comprehensive reconfiguration process is illustrated in Figure 5.16. This flowchart shows both variants of reconfiguration. Enabling redundant instruction execution necessitates synchronized pipelines

impl	ver	icc	reserved	DR	EC	EF	PIL	S	PS	ET	CWP
31:28	*27:24*	*23:20*	*19:13*	*14*	*13*	*12*	*11:8*	*7*	*6*	*5*	*4:0*

Figure 5.15: Modified PSR to disable the reconfiguration of the pipeline. When the *DR* bit is set the pipeline is prevented to enter the *sync* state.([19])

of both the master and slave core. The synchronization of these cores encompasses the implementation of a deadlock prevention mechanism.

The disabling of a redundant rexecutions requeirs an reestablishing of independent and non-redundant cores. This necessitates only the resumption of the slave core. Therefore, only the slave core restores the orignal pipeline state by flushing out the master core instructions und fetching the orignal slave instructions. However, the master core continuous uninterruptedly the prorgam execution.

5.3.4 Fault Detection and Handling

The fault detection and handling are supervised by the *commit stage*, which employs comparators to identify faults through result mismatches. A mismatch occurs whenever the results from the *master* and *slave* pipelines are not identical. In this setup, the *master* pipeline compares its result with that generated by the *slave*, and vice versa. Upon detecting a fault through mismatch, the pipeline engages the fault handling mechanism.

The fault handling approach employed for the ALP goes beyond just fault detection and notification to the application. Instead, a fault handling strategy is applied, with the processor pipeline autonomously managing faults. Upon each fault detection by the commit stage, the flawed instruction is re-executed. The processor core rolls back by

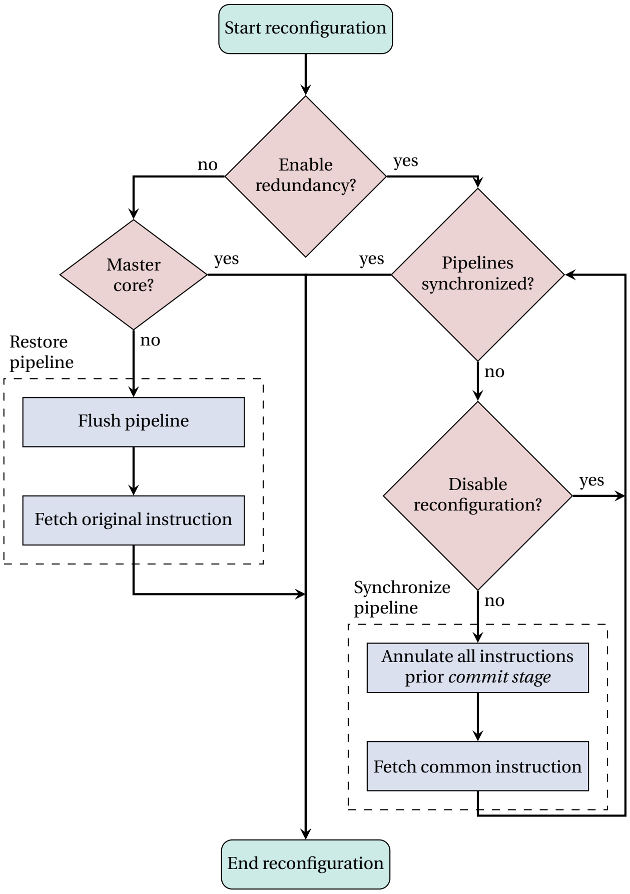


Figure 5.16: Flowchart illustrating the reconfiguration process of both master and slave core.

fetching the faulty instruction, utilizing the *rbpc* for this purpose. In case of a discrepancy detection in both commit stages, a simultaneous rollback to the same *rbpc* occurs. The choice between using the *rbpc* or the saved PC is supervised by the commit stage.

The detection of faults leads to the annulment of the flawed instruction, ensuring that it has no impact on the system state. Neither the internal processor state, comprising control registers in the write-back stage and general-purpose registers, nor the cache and main memory are affected by the detected fault.

The process of re-executing the instruction resembles the resumption of the suspended slave program or program replication. The commit stage employs a consistent mechanism to determine the required PC and NPC. Thus, the commit stage searches through the pipeline to identify the correct PC and NPC.

To emphasize the importance of selecting the correct NPC, it is important to understand the behavior of Control-Transfer Instructions (CTIs). CTIs influence the program flow before a result undergoes validation in the commit stage, examples being Branch and CALL instructions. These instructions generate the target address relative to their program counter in the decode stage and forward it to the fetch stage. All CTIs defined by the SPARC v8 ISA are Delayed Control-Transfer Instructions (DCTIs). The control transfer of DCTIs does not occur immediately; it is delayed by one instruction. This delayed instruction immediately follows the DCTIs and is always fetched but can be annulled by the preceding DCTIs.

Listing 5.1 provides an example code of a trap handler for a window underflow trap. In the event of a window underflow trap, the processor jumps to the address of the window underflow handler. The four reserved instructions are used to jump and execute the software routine of the window underflow handler. This routine is designed to save

Listing 5.1: Code example of window overflow trap handler

```
PC Address    Instruction

;Trap Table (Window Overflow)
80000050:   rd    %psr, %l0
80000054:   sethi %hi(0x80006000), %l4
80000058:   jmp   %l4 + 0x238
8000005c:   nop
     ...           ...

;Window Overflow Trap Handler
80006238:   save
8000623c:   std   %l0, [%sp]
     ...           ...
8000627c:   restore
80006280:   jmp    %l1
80006284:   rett   %l2

;Window Underflow Trap Handler
80006288:   sethi  %hi(0x80023000), %l6
8000628C:   or    %l6, 0x3dc, %l6
80006290:   ld    [ %l6 ], %l7
```

all register values of the register window on the stack and subsequently jump back to the source of the window underflow trap.

The code example features several DCTIs. The first instance is the **jmp** %**l4** + 0x238 in the third line. This jump instruction directs the fetching of instructions from the address *0x80006238* onwards. The DCTI changes the control flow one instruction after its fetch. Prior to fetching and executing the **save** instruction at the target address, the delayed **nop** instruction is fetched and executed. The No Operation (NOP) instruction is located in the delay slot of the jump instruc-

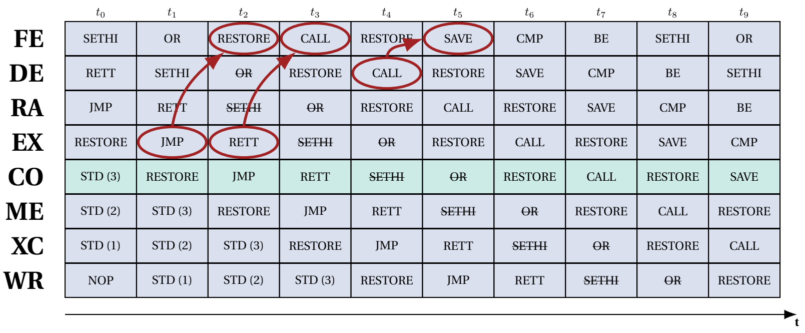
```
PC Address   Instruction

80012f1c:    ret
80012f20:    restore  <---------- Generates Trap
80000050:    rd   %psr, %l0  <--- WO Trap Handler
80000054:    sethi %hi(0x80006000), %l4
80000058:    jmp  %l4 + 0x238
8000005c:    nop
80006238:    save
8000623c:    std  %l0, [%sp]
   ...         ...
8000627c:    restore
80006280:    jmp   %l1  <------ Return to Trap Source
80006284:    rett  %l2  <------ Set Correct NPC
80012f20:    restore
80009f94:    call  80012ea0
80009f98:    restore
```

Figure 5.17: Instruction trace of an executed window overflow trap handler.

tion, and it is important to know that the instruction in the delay slot must not be a NOP and may also modify the program counter and control flow.

A delayed instruction that changes the control flow is shown on line ten at the address *0x80006284*. The structure on lines nine and ten is typical for exiting a trap handler, ensuring the correct setting of PC and NPC. When a trap occurs, the PC is stored in the *local 1 register* (*l1*) and the NPC is stored in the *local 2 register* (*l2*). Upon leaving the trap handler subroutine, the stored PC (*l1*) and NPC (*l2*) are utilized to restore the execution state before the trap occurred.

In Figure 5.17, an instruction trace illustrates a program execution. The traced snippet includes the source of a trap and the subsequent return from the trap to the source. The **restore** instruction in the sec-



	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
FE	SETHI	OR	RESTORE	CALL	RESTORE	SAVE	CMP	BE	SETHI	OR
DE	RETT	SETHI	~~OR~~	RESTORE	CALL	RESTORE	SAVE	CMP	BE	SETHI
RA	JMP	RETT	~~SETHI~~	~~OR~~	RESTORE	CALL	RESTORE	SAVE	CMP	BE
EX	RESTORE	JMP	RETT	~~SETHI~~	~~OR~~	RESTORE	CALL	RESTORE	SAVE	CMP
CO	STD (3)	RESTORE	JMP	RETT	~~SETHI~~	~~OR~~	RESTORE	CALL	RESTORE	SAVE
ME	STD (2)	STD (3)	RESTORE	JMP	RETT	~~SETHI~~	~~OR~~	RESTORE	CALL	RESTORE
XC	STD (1)	STD (2)	STD (3)	RESTORE	JMP	RETT	~~SETHI~~	~~OR~~	RESTORE	CALL
WR	NOP	STD (1)	STD (2)	STD (3)	RESTORE	JMP	RETT	~~SETHI~~	~~OR~~	RESTORE

Figure 5.18: Correct Instruction execution of the pipeline. The relationship between CTI and fetched instructions is highlighted.

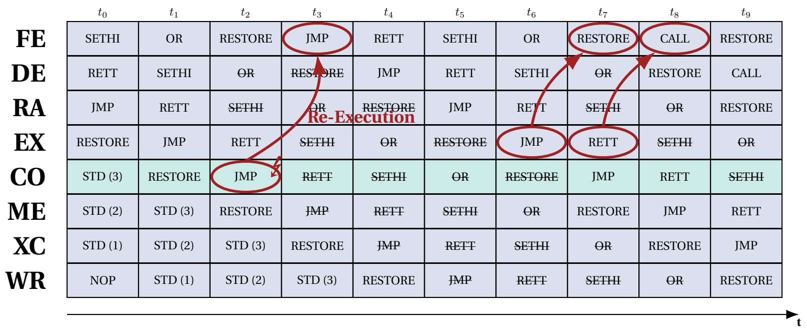
ond line triggers a window overflow trap. Instructions associated with the trap handler are highlighted in blue. As previously demonstrated in Listing 5.1, the trap handler concludes with the instruction combination of **jmp** %**l1** and **rett** %**l2** in lines eleven and twelve. After the trap handler is completed, these two instructions restore the original control flow.

The **jmp** %**l1** instruction restores the PC of the trap source. The register *l1* contains the address *80012f20* of the **restore** instruction. The **restore** instruction at address *80012f20* is located in the delay slot of the **ret** instruction. The **ret** instruction is used to exit the subroutine of a function call and jumps back to the *caller address* + *8*. In this example, the return address is *80009f94*. The correct re-establishment of the original control flow is facilitated by the **ret** instruction, which restores the NPC with the address *80009f94*. This ensures that after the trap, firstly, the trap source instruction at the address *80012f20* (saved PC) and, secondly, the instruction at the target of the jump at the address *80012f20* (saved NPC), are executed.

```
PC Address   Instruction

80012f1c:    ret
80012f20:    restore
~~80000050:   rd   %psr, %l0~~
                                 Re-Execute
80000050:    rd   %psr, %l0
80000054:    sethi %hi(0x80006000), %l4
80000058:    jmp  %l4 + 0x238
8000005c:    nop
   ...        ...
8000627c:    restore
~~80006280:   jmp    %l1~~
                             Re-Execute
80006280:    jmp    %l1
80006284:    rett   %l2
80012f20:    restore
80009f94:    call   80012ea0
80009f98:    restore
```

Figure 5.19: Instruction trace resulting from a simple re-execution of instructions.

The corresponding pipeline that generates the instruction trace is illustrated in Figure 5.18. The figure emphasizes the presence of CTIs, including those designed to exit the trap handler. The instruction combination of **jmpl** followed by the **rett** instruction ensures that the fetch stage restores the original program state from before the window overflow occurrence. This pair of instructions sets the correct PC and NPC, with the **jmpl** setting the PC and the subsequent **rett** instruction setting the NPC. The sequential execution is illustrated in the figure, where the execution of the **jmpl** restores the correct **restore** instruction, and the **rett** ensures the correct fetch of the subsequent **call** 80012ea0 instruction.



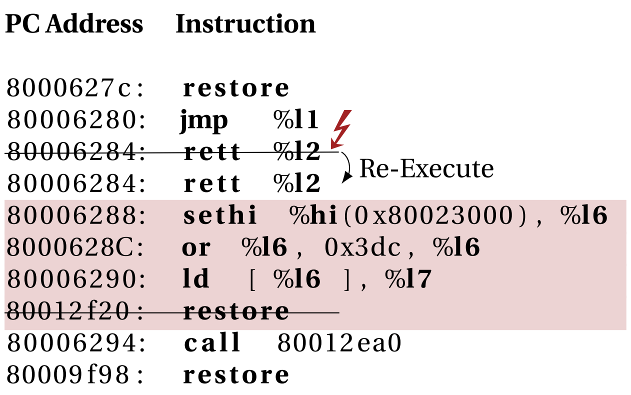
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
FE	SETHI	OR	RESTORE	JMP	RETT	SETHI	OR	RESTORE	CALL	RESTORE
DE	RETT	SETHI	~~OR~~	~~RESTORE~~	JMP	RETT	SETHI	~~OR~~	RESTORE	CALL
RA	JMP	RETT	~~SETHI~~	~~OR~~	~~RESTORE~~	JMP	RETT	~~SETHI~~	~~OR~~	RESTORE
EX	RESTORE	JMP	RETT	~~SETHI~~	~~OR~~	~~RESTORE~~	JMP	RETT	~~SETHI~~	~~OR~~
CO	STD (3)	RESTORE	JMP	~~RETT~~	~~SETHI~~	~~OR~~	~~RESTORE~~	JMP	RETT	~~SETHI~~
ME	STD (2)	STD (3)	RESTORE	~~JMP~~	~~RETT~~	~~SETHI~~	~~OR~~	RESTORE	JMP	RETT
XC	STD (1)	STD (2)	STD (3)	RESTORE	~~JMP~~	~~RETT~~	~~SETHI~~	~~OR~~	RESTORE	JMP
WR	NOP	STD (1)	STD (2)	STD (3)	RESTORE	~~JMP~~	~~RETT~~	~~SETHI~~	~~OR~~	RESTORE

Figure 5.20: Instruction re-execution of the pipeline. The relationship between flawed **jmp** instruction and the following instructions is shown. The re-execution results into a correct control flow.

The re-execution after a fault, based on the trace example, should be discussed. The example encompasses all relevant instructions and instruction combinations to cover various behavioral patterns. The first scenario in Figure 5.19 illustrates the simplest case, featuring two faults, one at the beginning of the trap handler. The commit stage detects a mismatch at the **rd %psr**, **%l0** instruction, where the results of the *slave* and *master cores* are not equivalent, triggering a re-execution of the flawed instruction.

The example trace has a second fault at a DCTI. The **jmp %l1** instruction contains a fault, prompting a re-execution of this instruction. The annulment and re-execution of the **jmp %l1** result in a correct program flow.

Both faults at the **rd %psr**, **%l0** or **jmp %l1** instructions in lines three and ten lead to a re-execution of these instructions. The flawed instructions are not completed and are annulled. Consequently, the resulting instruction trace is identical to the intended trace from Figure 5.17.



PC Address **Instruction**

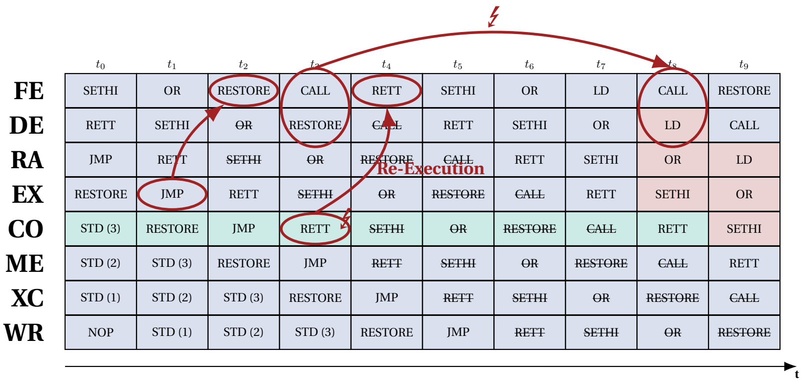
8000627c: **restore**
80006280: **jmp** %**l1**
~~80006284: **rett** %**l2**~~
80006284: **rett** %**l2**
80006288: **sethi** %**hi**(0x80023000), %**l6**
8000628C: **or** %**l6**, 0x3dc, %**l6**
80006290: **ld** [%**l6**], %**l7**
~~80012f20: **restore**~~
80006294: **call** 80012ea0
80009f98: **restore**

Figure 5.21: Instruction trace resulting from a simple re-execution of a flawed **rett** instruction. The re-execution leads to an incorrect program execution (highlighted in red)

The pipeline behavior of fault handling and the annulment of the flawed instruction is illustrated in Figure 5.20. The figure shows the re-execution of the flawed **jmp** %**l1** instruction. The instruction itself and all instructions prior to the *commit stage* are annulled. The fetch stage then re-fetches the **jmp** %**l1** instruction for re-execution. The subsequently fetched instruction is the correct **rett** instruction, ensuring the correct continuation of the program.

The next two examples demonstrate that only based on the PC for re-execution is insufficient. A straightforward re-execution of the PC while neglecting the NPC results in flawed program execution, manifesting as an incorrect or incomplete program flow with missing instructions, multiple incorrectly executed instructions, or an entirely incorrect program sequence.

The first example in Figure 5.21 illustrates the consequences of a simple re-execution solely at the PC address without further considerations. This leads to an incorrect program flow, with one missing in-

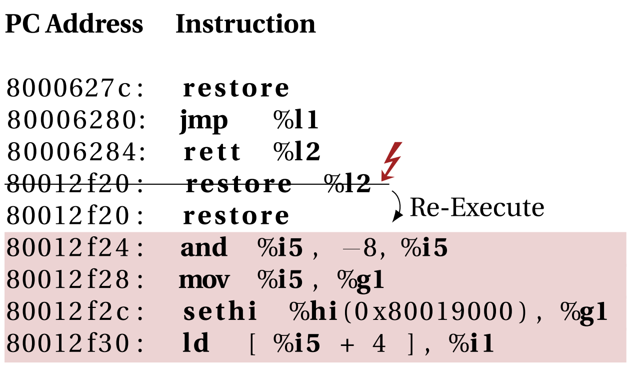


	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
FE	SETHI	OR	RESTORE	CALL	RETT	SETHI	OR	LD	CALL	RESTORE
DE	RETT	SETHI	~~OR~~	RESTORE	~~CALL~~	RETT	SETHI	OR	LD	CALL
RA	JMP	RETT	~~SETHI~~	~~OR~~	~~RESTORE~~	~~CALL~~	RETT	SETHI	OR	LD
EX	RESTORE	JMP	RETT	~~SETHI~~	~~OR~~	~~RESTORE~~	~~CALL~~	RETT	SETHI	OR
CO	STD (3)	RESTORE	JMP	RETT	~~SETHI~~	~~OR~~	~~RESTORE~~	~~CALL~~	RETT	SETHI
ME	STD (2)	STD (3)	RESTORE	JMP	~~RETT~~	~~SETHI~~	~~OR~~	~~RESTORE~~	~~CALL~~	RETT
XC	STD (1)	STD (2)	STD (3)	RESTORE	JMP	~~RETT~~	~~SETHI~~	~~OR~~	~~RESTORE~~	~~CALL~~
WR	NOP	STD (1)	STD (2)	STD (3)	RESTORE	JMP	~~RETT~~	~~SETHI~~	~~OR~~	~~RESTORE~~

Figure 5.22: A pipeline illustrating re-executing a flawed **rett** instruction in the delay slot of a CTI. The simple re-execution leads to a incorrect program continuation, which is highlighted in red

struction compared to the ground truth trace, highlighted in red and crossed out. Additionally, some falsely executed instructions are highlighted in red. Following the flawed program flow, the correct **call** instruction is fetched. However, the system state may already be corrupted due to the missing and incorrectly executed instructions. This undesired behavior corresponds to the naive re-execution of the instruction in the delay slot of the **jmp %l1** instruction.

The re-execution process involves initially flushing all instructions prior to the commit stage, including the already fetched **restore** instruction, which is annulled like any other instruction. After flushing all pipeline stages prior to the commit stage, the flawed instruction is re-fetched and re-executed. Without considering any NPC, the next fetched instruction depends on either the PC increment or the result of a CTI.



```
PC Address    Instruction

8000627c:     restore
80006280:     jmp     %l1
80006284:     rett    %l2
~~80012f20:     restore   %l2~~
80012f20:     restore
80012f24:     and     %i5 ,  −8,  %i5
80012f28:     mov     %i5 ,  %g1
80012f2c:     sethi   %hi(0x80019000) ,  %g1
80012f30:     ld      [ %i5 + 4 ] ,  %i1
```

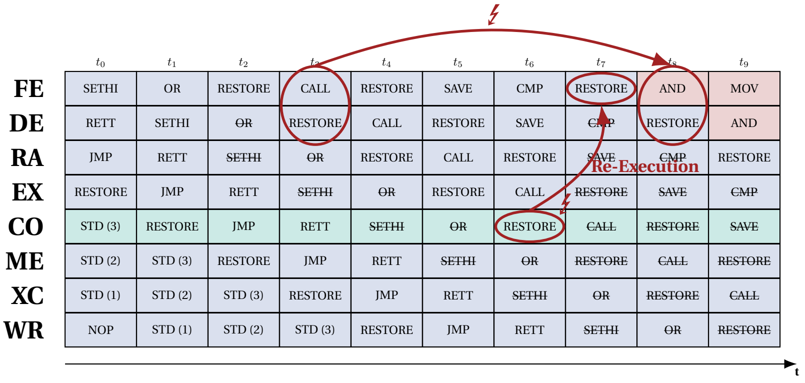
Figure 5.23: Instruction trace resulting from a simple re-execution of a flawed target **restore** instruction. The re-execution leads to an incorrect program execution (highlighted in red)

The **jmp** %**l1** has taken effect before the fault at the delayed instruction is detected. Figure 5.22 illustrates this. The **jmp** %**l1** instruction has passed the *commit stage* and is not affected by a simple re-execution of **rett**. Consequently, the original control transfer does not take place again, resulting in an incorrect control flow. The falsely executed instructions are highlighted in red.

A **jmpl** or **jmp** instruction must be executed immediately before a **rett**. If a **rett** does not follow a **jmp** instruction, one or more instructions following the **rett** are incorrect [19].

The simple re-execution of the **rett** instruction after fault detection is equivalent to executing just a single **rett** instruction. This behavior is the same as the case where the **rett** does not immediately follow a **jmp** instruction.

The second example demonstrates a fault occurring in the target instruction of a **jmp** instruction. A simple re-execution of the flawed instruction without considering the NPC is discussed below. Similar



	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
FE	SETHI	OR	RESTORE	CALL	RESTORE	SAVE	CMP	RESTORE	AND	MOV
DE	RETT	SETHI	~~OR~~	RESTORE	CALL	RESTORE	SAVE	~~CMP~~	RESTORE	AND
RA	JMP	RETT	~~SETHI~~	~~OR~~	RESTORE	CALL	RESTORE	~~SAVE~~	~~CMP~~	RESTORE
EX	RESTORE	JMP	RETT	~~SETHI~~	~~OR~~	RESTORE	CALL	~~RESTORE~~	~~SAVE~~	~~CMP~~
CO	STD (3)	RESTORE	JMP	RETT	~~SETHI~~	~~OR~~	RESTORE	~~CALL~~	~~RESTORE~~	~~SAVE~~
ME	STD (2)	STD (3)	RESTORE	JMP	RETT	~~SETHI~~	~~OR~~	~~RESTORE~~	~~CALL~~	~~RESTORE~~
XC	STD (1)	STD (2)	STD (3)	RESTORE	JMP	RETT	~~SETHI~~	~~OR~~	~~RESTORE~~	~~CALL~~
WR	NOP	STD (1)	STD (2)	STD (3)	RESTORE	JMP	RETT	~~SETHI~~	~~OR~~	~~RESTORE~~

Figure 5.24: A pipeline illustrating re-executing a flawed **restore** instruction. The simple re-execution leads to a incorrect program continuation, which is highlighted in red

to previous examples, the resulting instruction trace is shown in Figure 5.23. The simple re-execution of the flawed **restore** instruction leads to a program continuation from the instruction address. The incorrectly fetched and executed instructions are highlighted in red.

The corresponding illustration of the pipeline is provided in Figure 5.24, highlighting the incorrect control flow as well. Specifically, the incorrectly fetched and later executed instructions are highlighted in red. The re-execution does not consider the target address of the **rett** instruction. The **rett** instruction has passed the *commit stage* and is not considered for a simple re-execution based only on the flawed instruction PC. Consequently, the program continues executing instructions from the re-executed **restore**. The *fetch stage* linearly fetches the next instructions until a CTI changes the NPC. This leads to a flawed control flow and falsely fetched and executed instructions.

The previous examples show that re-execution cannot only be based on the PC of the flawed instruction. For DCTIs, the correct PC and NPC must be determined. The *commit stage* therefore identifies both the correct PC of the current instruction and NPC. Both values are stored for possible re-execution. The control flow after re-execution depends on the *Rollback Program Counter (RBCP)*. The *commit stage* configures the *fetch stage* to use the *RBCP*, which is set to PC and NPC successively. This ensures that the *fetch stage* restores the correct program flow after a flawed instruction is detected.

The state replication from the *master* to the *slave core* and the program suspension are similar to a re-execution in the case of a fault. Therefore, the same mechanism of the *commit stage* is employed to detect the correct and required PC and NPC. Both values are used to replicate the program. The *master* and *slave core* re-execute the PC and NPC to replicate the correct program flow. After the critical section is finished, the *slave core* restores its original state. Thus, the original PC and NPC are re-established, and the original program flow resumes.

In the following the fault detection latency should be discussed. The latency to detect a fault depends on the pipeline stage where the fault occurs. The maximum latency is defined by the maximum distance between the pipeline stage where the fault occurs and the *commit stage*. The worst-case latency is for a fault occurring in the *fetch stage*. Therefore, the maximum fault detection latency is defined as the time the fetched instruction needs to reach the *commit stage*, as shown in Equation 5.8.

$$L_{FaultDetection,Max} = L_{Fetch,Commit}(Inst). \tag{5.8}$$

5.4 Improvements of the Adaptive Lockstep Architecture

The demonstrated Adaptive Lockstep Processor (ALP) concept has been illustrated with two AL cores thus far. In the following, a generalization of the concept is discussed. Additionally, some enhancements concerning pipeline design are considered to address a consistent runtime overhead resulting from the additional *commit stage*.

5.4.1 Direct Fault Masking

The ALP and the Adaptive Redundancy (AR) concept cover redundant execution on two processor cores. However, for some applications, the DMR configuration is not sufficient. To enhance the system's dependability, a TMR processor configuration is required instead. This section delves into the realization of the TMR configuration within the ALP.

The design of the TMR configuration follows the same principles as the previously presented DMR configuration. Similar to the DMR setup, the processor cores share common subsystems during the *locked mode*. The architecture described in Section 5.2 serves as the foundational design. In the case of a TMR configuration, the *register file* of the *master* core is shared not only with a single slave core, as in the DMR configuration, but with two *slave* cores. Furthermore, the cache signals are distributed to both slaves, with both data and hold signals required on both slaves to ensure identical and functional execution across all three cores.

In the synchronization process, as described in Section 5.3.2, both *slave* pipelines must be considered. The *synchronization* state is not exited until both *slave* pipelines are completely flushed; it is insuffi-

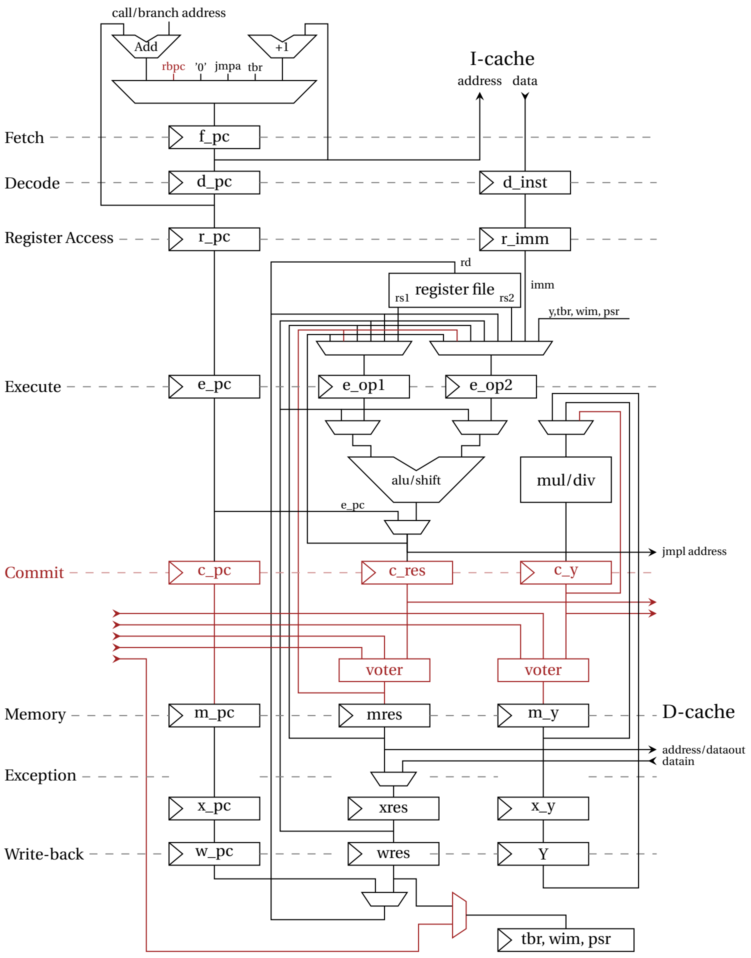


Figure 5.25: Datapath of the eight-stage-pipeline incorporating voter instead of comparators.

cient for just one of the *slave* pipelines to be flushed. After both *slave* pipelines are flushed, both *slave cores* can accept the provided *master* state.

The data path of the LEON3 processor employs voters instead of comparators to detect and directly correct faults. For the TMR configuration, 2-out-of-3 majority voters are utilized.

The resulting data path architecture is shown in Figure 5.25. For correct data forwarding, the result after the voter needs to be forwarded to the *register access stage*. In the original design based on comparators, the location of data forwarding is negligible; it makes no difference whether the result is forwarded before or after the comparator.

The use of comparators in the DMR configuration to detect a fault and re-execute, if necessary, leads to a Backward Error Recovery (BER). Fault handling requires the time-consuming re-execution. However, the use of voters allows for an Forward Error Correction (FEC).

5.4.2 Reconfiguration of the Pipeline Architecture

The presented pipeline architecture employs the *commit stage* to detect and mitigate faults, statically placing the pipeline stage between the *execute* and *memory stage*. The integration of the additional *commit stage* introduces additional latency in certain scenarios, notably delaying the execution of load operations. Additionally, the detection of trap occurrences (window overflow or underflow) is delayed compared to the seven-stage pipeline design.

To address the negative runtime impact, an adaptive pipeline structure is explored. The adaptive pipeline concept utilizes a reconfiguration approach. During the *split mode*, the pipeline adopts a *shadow mode*. Figure 5.26 illustrates the adaptive pipeline concept. Depend-

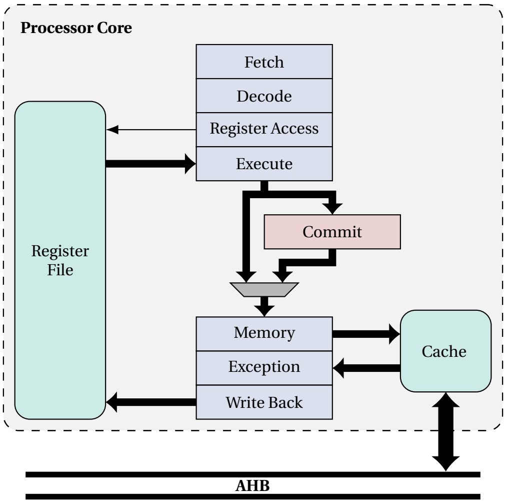


Figure 5.26: Reconfigurable pipeline architecture. The *commit stage* operates as shadow pipeline stage during *split* mode and is embedded into the data path during *locked mode*.

ing on the processor mode, the commit stage is included or excluded from the pipeline data path.

The pipeline employs a bypass architecture to exclude the *commit stage* from the data path, configuring it as a *shadow pipeline stage*. During the *split mode*, the *commit stage* only tracks the correct PC and NPC for processor state replication. When the processor transitions from *split* to *locked mode*, the *commit stage* takes over the task of orchestrating the replication process. During the synchronization phase, when all instructions are flushed, the *commit stage* is inserted into the data path of the pipeline architecture.

Figure 5.27 illustrates the instruction execution of the pipeline. Each executed instruction is highlighted in blue, while instructions discarded by the *commit stage* due to the *shadow mode* configuration are highlighted in gray.

Activation of the *commit stage* and its inclusion into the data path occurs immediately after pipeline synchronization. During pipeline synchronization, both the *master* and *slave* cores' pipelines are flushed. Consequently, the inclusion of the *commit stage* into the instruction stream of the pipeline takes place seamlessly, with no impact on the instruction flow.

Recovering the shadow pipeline stage depends on the core type. The *slave core* flushes the complete pipeline to restore the original state and resume the suspended program. Flushing the pipeline leads to annulling the instructions at the *execute, commit,* and *memory stages*. By annulling the instruction at the *commit stage*, the commit stage can be excluded from the data path without risking flawed instruction execution.

Excluding the *master's commit stage* requires additional effort. Direct deactivation and bypassing of the *commit stage* are not feasible as long as a valid instruction is available at the *commit stage*. When both the *execute* and *commit stages* have valid instructions, removing the *commit stage* from the data path leads to flawed execution. Only one instruction can be passed to the *memory stage*, either the valid instruction from the *execute* or *commit stage*. Figure 5.27 not only illustrates instruction execution before and after switching from *split* to *locked mode* but also includes the restoration of a seven-stage pipeline architecture with a shadow stage when switching from locked back to split mode. The figure illustrates the more interesting case of the master pipeline, which can seamlessly continue code execution.

The *commit stage* can be excluded from the data path as soon as the *commit stage* has an annulled instruction. As soon as the annulled instruction is detected at the *commit stage*, the result of the *execute stage*

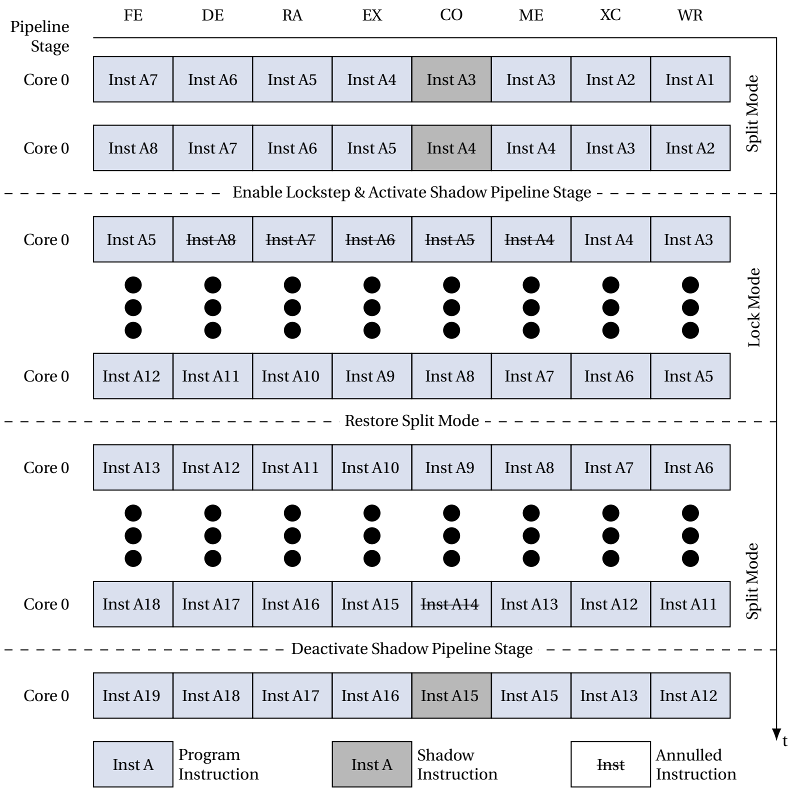


Figure 5.27: Pipeline synchronization and restoration of the adaptive pipeline structure.

can be safely forwarded to the *memory stage*. By forwarding the result of the *execute* instead of the *commit stage*, the seven-stage design with the shadow pipeline stage is recovered. Each instruction entering the commit stage is discarded, ensuring that the *commit* and the *memory stage* process the same instruction.

5.5 Experimental Results

The lockstep architecture is evaluated with respect to resource overhead, execution time in fault-free scenarios, and correct software behavior. The correctness of software behavior is demonstrated through simulation and on an FPGA platform. The adaptive lockstep architecture undergoes validation through random fault injection. In simulation, faults are injected using VHDL code, while on the FPGA platform, faults are introduced by extending the architecture. Injected faults manipulate results forwarded from the execution stage to the commit stage.

5.5.1 Fault Detection

The fault detection capabilities of the ALP are validated through both hardware simulation and a modified hardware implementation on an FPGA. Both methods demonstrate the fault detection and recovery capabilities. The evaluation of fault detection behavior also validates the functional correctness of context switches between the split and locked modes. The validation involves random switches between the modes, injecting faults whenever the process is in lock mode.

In the simulation-based validation, the RTL implementation of the ALP is used, and faults are injected using Mentor QuestSim. Signals are changed at random points in time, and faults are injected after dynamically reconfiguring processor cores into a DMR or TMR cluster. Various benchmark programs and a program implementing a heavy recursive test algorithm are used for this validation. The benchmark programs cover common problems of embedded applications, while the recursive test algorithm validates the critical phase of entering and leaving the trap handler.

The SPARC v8 ISA defines a privileged supervisor mode during trap handling, and recursive function calls are employed to generate window under and window overflow exceptions, ensuring reconfiguration switches during trap handling or between a switch from or to the supervisor mode.

Simulation-based injection faces challenges in terms of controllability and repeatability, as well as simulation time. To address this, a hardware extension is used to inject faults precisely into the processor pipeline. A *fault injection unit* is placed between the execute and commit stages, dedicated to one processor pipeline. Each pipeline is equipped with its own *fault injection unit,* and they work independently, depending on the associated pipeline state.

The architecture of the fault injection setup is illustrated in Figure 5.28, illustrating two AL cores. Each *fault injection unit* within the processor cluster is independently accessible, with the *fault manager* configurable through either the AHB bus or an external Universal Asynchronous Receiver Transmitte (UART) interface.

The *fault injection unit* supports the following fault modes:

- Transient fault or Single Event Upset (SEU)
- Intermittent fault
- Permanent fault.

Each fault mode defines the frequency of fault injection. In the case of a transient fault, the configured fault is injected once when the trig-

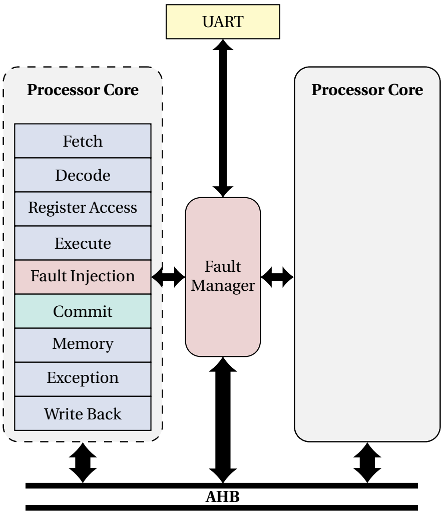


Figure 5.28: Fault injection architecture for the FPGA prototype

ger condition is met. For intermittent faults, the fault is periodically injected, with the injection cycle commencing when the trigger condition is first satisfied. Permanent faults inject the configured fault as soon as the fault condition is met for the first time.

Controlling the timing of fault injection, especially the moment when a fault or the begin of intermittent and permanent faults are injected, requires a fault trigger for the *fault injection unit*. Two methods for determining the timing of fault injection are as follows:

- Immediate
- PC-based.

The immediate fault injection injects the fault as soon as the *fault injection unit* is configured. The PC-based injection introduces a fault at a specific instruction, identified by its memory location and associated address. The address is utilized to set the PC where the fault should be injected.

To ensure that an injected fault is not masked and takes effect, the instruction type can be specified. The instruction type is not determined by the OP-code but rather by the execution type of the instruction. Thus, the following two types of instructions are considered:

- All instructions
- Only executed instruction (not annulled).

Fault triggers on annulled instructions are masked, as these instructions do not affect the processor state. To ensure that injected faults have an impact on the result of the *execute stage*, the instruction type must not be annulled. If the instruction is not annulled, it is executed and influences the system state.

The type of injected fault can be managed by the following three operations:

- AND
- OR
- XOR

One of these operations is applied to connect a defined and configurable fault vector with a target vector. The target vector represents the result of the *execute stage* where the fault is injected. The *AND* operation is used to force a *'0'*, and the *OR* operation is used to force a *'1'* in the target vector. Both operations can be employed in combination with a permanent fault to inject faults, covering latch-up faults (Stuck-at-'1' and Stuck-at-'0'). The *XOR* operation allows precise flipping of selected bits in the target vector.

This configuration, comprising fault mode, fault trigger, and fault type, enables the coverage and mapping of all relevant fault classes at the logical level.

The same programs used for the simulation-based evaluation are employed for the FPGA evaluation, with the exception of adaptation to switch between *split* and *locked* mode. Faults are periodically injected during lockstep execution. In a *lock configuration*, random faults are induced into the result of the execution stage. Faults are injected by bit inverting a result of the *master* or *slave core*. The fault injection is based on an intermittent fault mode, where the interval is used to simulate random faults. The faults are only injected into non-annulled and executed instructions, ensuring that the injected fault takes effect.

Permanent faults are detected for a DMR cluster; however, these faults result in a processor hang, as the fault is still present after re-execution. For the TMR cluster, these faults are masked, as the correct result of two processors forms the majority for the voter.

Furthermore, adaptive reconfiguration from a DMR into a TMR is investigated for permanent faults. Therefore, a threshold of faults is specified. When the threshold is exceeded, the ALP cores automatically reconfigures the DMR into a TMR cluster. The permanent fault leads to exceeding the configured threshold after re-executing the same flawed instruction.

Both the simulation-based and hardware implementations are used to demonstrate the effectiveness of the safety mechanism provided by the ALP and the context switch by state replication. When the procedure is configured into *lock mode*, fault injection does not affect correct execution, and correct functional behavior of the test program codes is shown. However, faults injected during the *split mode* to cross-validate the malfunctioning behavior of a fault lead to flawed execution. The executed and flawed programs differ from a flawless execution and can result into a processor hang.

5.5.2 Resources

A prototype of the AL processor, based on the LEON3 design, is implemented as a proof of concept. The prototype is developed by modifying the original source code of the LEON3 processor design, focusing primarily on functional verification. The design does not prioritize optimal hardware implementation with respect to resource utilization, and the presented values represent the upper bound of the hardware realization, allowing room for implementation improvements.

The hardware overhead of extending the additional pipeline stage is compared against the original seven-stage pipeline implementation of the LEON3. The implementation targets the Virtex UltraScale+ XCVU9P-L2FLGA2104E on the VCU118 evaluation board [151] from AMD.

The evaluation of resource utilization is conducted by synthesizing the design with AMD's Vivado [152]. For both synthesis and implementation, Vivado is employed with the standard strategy. The design runs on a clock frequency of 200 MHz without timing violations, aligning with the original seven-stage LEON3 design's frequency. The required resources are reasonable and are detailed in Table 5.3. The additional LUTs for the adaptive structure are low, and due to the duplication of processor registers and the extra pipeline stage, the design is primarily dominated by the additional registers.

Table 5.3: Implementation results of resource utilization

	LEON3	Lockstep	Overhead [%]
CLB LUTs	5224	5613	7.44
CLB Registers	2269	2887	27.24
CLBs	1003	1220	21.70
BRAMs	9	9	0

5.5.3 Runtime Overhead

The impact of the additional processor pipeline stage is assessed through HDL implementation. The RTL implementation of the 8-stage pipeline design not only serves to evaluate the behavior of the fault detection and recovery mechanism but is also synthesized to an FPGA prototype. As previously described, the prototype is based on the Virtex UltraScale+ VCU118 board. It is utilized to measured the runtime overhead introduced by the *commit stage*.

Five representative benchmark applications are chosen to evaluate the runtime overhead on the prototype platform. Each benchmark generates a Central Processing Unit (CPU) workload, covering specific aspects such as memory accesses, trap generation for window overflow and underflow, as well as branches.

Fast Fourier Transform (FFT) is an efficient algorithm for calculating the discrete Fourier transformation of a time-discrete signal. This transformation is employed to convert a signal from the time domain to the frequency domain and is widely used in signal processing applications. The implementation is based on the radix-2 Cooley-Tukey algorithm, which divides the problem into smaller sub-problems using a divide-and-conquer approach. The computation breakdown relies on complex number operation such as addition [153]. Due to the absence of a Floating Point Unit (FPU), the algorithm implementation is limited to integer operations.

Matrix Multiplication (MatMul) is a fundamental operation in linear algebra and a central operation in many applications, such as rigid body transformation [154] or Artificial Intelligence (AI). For instance, the convolution operation in Convolutional Neural Networks (CNNs) can be represented as a matrix multiplication. By utilizing the *im2col* algorithm, the data from any convolution can be transformed into a General Matrix Multiply (GEMM) operation [155]. The algorithm employs a highly iterative implementation with three nested loops.

Quicksort (QS) is a recursive, comparison-based sorting algorithm that employs the divide-and-conquer principle [156]. This sorting algorithm utilizes an in-place method, which introduces a small memory overhead. However, the recursive function calls require additional stack memory.

Mergesort (MS) is a comparison-based, recursive sorting algorithm that employs the divide-and-conquer principle [157]. Unlike the QS algorithm, MS is a stable sorting algorithm. The divide step of the algorithm introduces more complexity than the QS algorithm; however, the merge phase is simpler. It's important to note that the algorithm does not sort in-place, requiring additional memory allocation.

Black-Scholes (BS) is a financial mathematical model used in the dynamic derivative market [158]. While the financial market might not be a typical embedded domain, BS represents a class of different equations. The algorithm solves a parabolic partial differential equation, a common problem for embedded systems. The implemented algorithm, derived from [159], must compute results numerically, as there is no closed-form expression for the BS equation. Since the presented processor architecture lacks FPU support and integer solutions are insufficient, a software-based floating-point implementation is employed. This approach results in efficient data reuse.

The five benchmarks serve as workloads to assess the runtime overhead introduced by the additional pipeline stage. The inclusion of the *commit stage* results in delayed execution of load-store operations and late trap detection. Traps, encompassing errors like *data store errors*, *illegal instructions*, or *misaligned memory addresses*, also include events such as *window overflow* or *window underflow* [19]. *Window overflows* and *underflows* are associated with function calls; if a function attempts to access an already occupied *register window*, one of these traps is triggered. Typically, user-application programs are not directly affected by these traps, as they are handled by a supervised trap handler. This handler, executed in supervisor mode, returns control to the user-application after completing the trap handling. The LEON Bare-C Cross Compilation System (BCC)'s GNU GCC automatically integrates a trap handler for *window overflows* and *underflows* [160]. The user program and implemented workloads do not explicitly consider the trap handler and depend on the automatically integrated trap handler.

The results are obtained by running each benchmark for both 7-stage and 8-stage pipeline configurations and measuring the required clock cycles. Clock cycles is measured using a tightly coupled hardware counter, directly readable within the pipeline as a special register [18]. Each benchmark application program reads the counter value before and after the execution of the algorithm, and the difference yields the execution time of the algorithm. To mitigate external influences, each evaluation is performed 15 times, and the arithmetic mean is calculated. The difference between the minimum and maximum is small and negligible. The evaluation results for the five benchmarks are presented in Figure 5.29, given in the required clock cycles to execute the workload. All algorithms are evaluated with different input data sizes

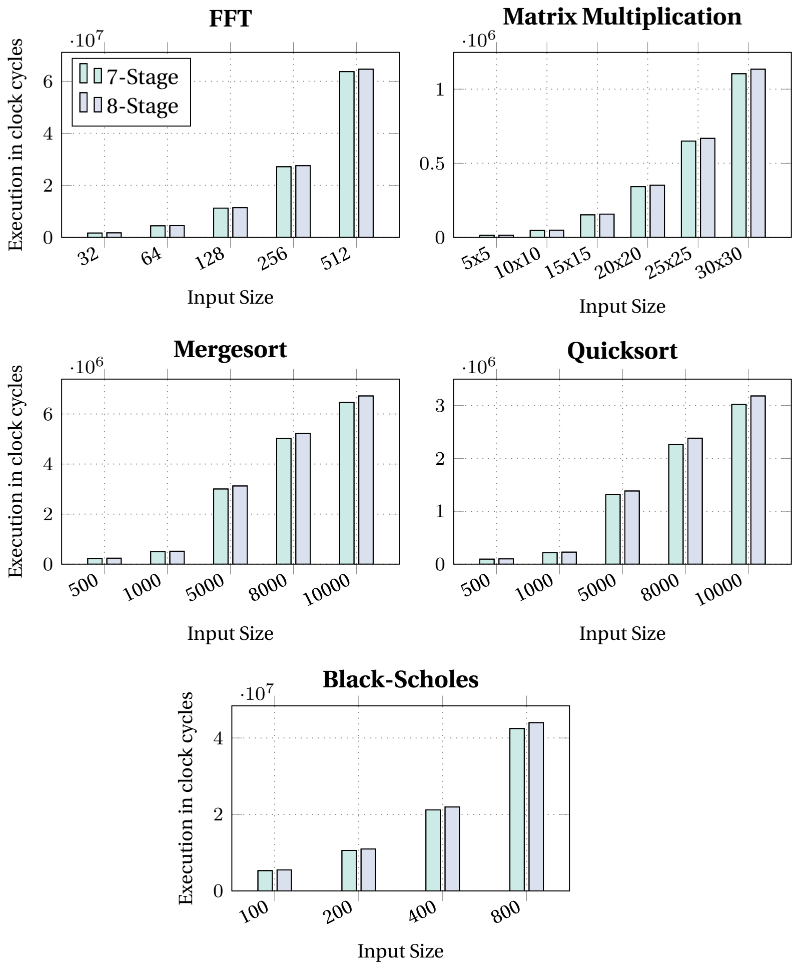


Figure 5.29: Evaluation of the absolute runtime overhead introduced by the *commit stage*

to generate varied workloads. The input for the merge and quicksort is identical to ensure repeatability across the 15 runs.

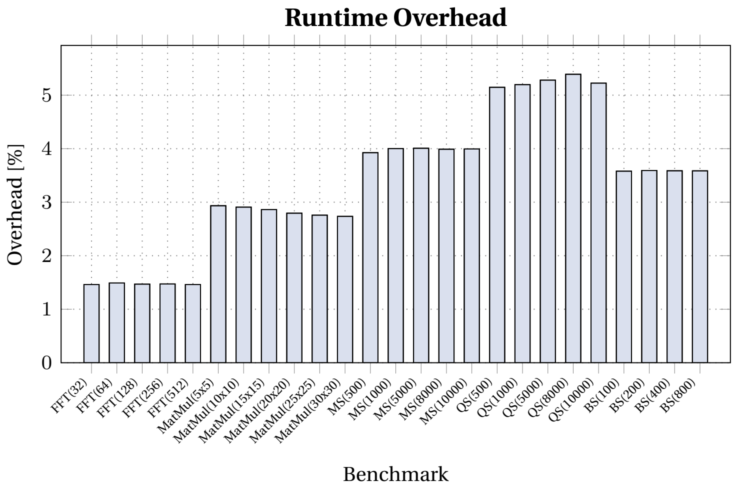


Figure 5.30: Comparison of the relative runtime overhead introduced by the *commit stage*.

Recursive algorithms are suitable for evaluating the trap-handling behavior of the extended pipeline. The SPARC v8 ISA uses the window system to pass function parameters, triggering window overflows and underflows and invoking the associated trap handler

The relative runtime overhead introduced by the additional pipeline stage is illustrated in Figure 5.30. The results reveal a modest overhead ranging from 1.46% to 5.28% when compared to the 7-stage implementation. Algorithms dominated by exceptions exhibit a higher runtime overhead, with both merge sort and quicksort generating more exceptions due to increased window overflow and underflow traps compared to matrix multiplication or FFT. The additional pipeline stage, positioned before the exception stage, results in a delayed exception handling process, causing a delay in the context switch from user software to the trap handler.

Matrix multiplication extensively operates on data that cannot be accommodated in the register file, necessitating multiple load and store operations. The extra pipeline stage introduces more read-after-write hazards, contributing to delays in overall execution time. Despite these factors, the total runtime overhead attributable to the additional pipeline stage remains small, being less than 5.3%.

5.5.4 Reconfiguration Time

The final evaluation investigates the time required for the reconfiguration of two processor cores. Specifically, the focus is on transitioning from the *split* mode to the *locked* mode. The evaluation quantifies the duration needed to switch both the *master* and *slave cores* into a single logical core. A hardware-based statistical unit is employed to measure this reconfiguration time, utilizing a hardware counter to count the clock cycles between the initiation and termination of the process.

The initiation condition is met when the *lockstepEn* signal is raised. When the signal rinses, both cores begin the reconfiguration process. The process concludes when both cores have entered the *locked* state and simultaneously fetches the same instruction.

This evaluation is conducted on the FPGA prototype. The two processor cores operate independently, executing a test application. A third core is exclusively designated to control the reconfiguration process, utilizing a pseudo-random process to initiate the transition from *split* to *locked* mode. The statistical unit is instrumental in measuring and recording the reconfiguration times.

The results of the measured reconfiguration times are presented in Table 5.4. For the executed application, an average time of 3.8 clock cycles is recorded. The observed reconfiguration time varies from a minimum of two clock cycles to a maximum of ten clock cycles.

	Reconfiguration time [clock cycles]
Min	2
Max	10
Avg	3.8

Table 5.4: Evaluation results of the reconfiguration time from switching from *split* into *locked mode*

As previously mentioned, the reconfiguration time is influenced by the instructions present in the pipeline stages after the commit stages. Particularly, instructions involving access to the main memory contribute to an increased reconfiguration time. Load and store instructions following the commit stage are executed and impact the necessary reconfiguration time.

To ensure the accuracy of the measurements, the test application intentionally excludes the deadlock prevention mechanism. The deadlock prevention mechanism disables the *slave's* reconfigure capability, leading to an increased reconfiguration time. This increase is applica"-tion-dependent rather than hardware-dependent.

5.6 Summary

This chapter introduces the concept of the ALP, which utilizes a hardware realization of the previously introduced AR concept. The analysis of the target processor pipeline with respect to failure modes and effects informs the placement of the commit stage within the pipeline architecture. The commit stage serves multiple purposes, with the three most crucial being the synchronization of processor pipelines, program replication from the master to the slave core, and automatic fault handling. Due to DCTI, additional logic is necessary for determining automatic fault handling in the context of automatic fault recovery.

The evaluation of the processor ALP architecture reveals minimal hardware and runtime overhead. The ALP demonstrates the ability to transition from a secure *locked mode* to a performance-oriented mode (*split mode*). To facilitate this transition, a pipeline reconfiguration concept is introduced. The commit stage, while introducing additional latency, can be bypassed during *split mode* to mitigate this latency.

Importantly, the introduced ALP concept is adaptable to various processor architectures. The modifications can be universally applied to all processor designs and are not confined to the example design. The concept is even applicable to out-of-order processors, necessitating the placement of the commit stage after the reorder unit and the memory and write-back unit.

In summary, the overall concept demonstrates a commendable balance between hardware abstraction, runtime overhead, and hardware resource utilization. Moreover, it fulfills all requirements derived from the AR concept.

Chapter 6

A Multi-Core Architecture based on the Adaptive Lockstep Core

This chapter describes a multi-core processor architecture based on the Adaptive Lockstep Processor (ALP) and its HDL. The chapter starts with the presentation of the tile-based processor architecture. An for real-time and safety-critical applications specialized Network Adapter (NA) is introduced afterwards. The chapter concludes with an evaluation and summary of the presented platform. Beforehand the Runtime Environment (RTE) used for the evaluation is introduce. It is used to evaluate the tile-based platform and the ALP by supporting the development of AR capable multi-core applications.

The concept, implementations and evaluation of the platform are published in [Kem22b] and [Red19]. The design and RTL impregnation of the NA are published in [Kem19]. This chapter extends these publications in further details.

6.1 The Processor Architecture

This section describes a tile-based multi-core architecture. It begins with an overview of the overall multi-core architecture and its specific requirements for Mixed Criticality Systems (MCSs). Following this, it offers a concise introduction to the Network-on-Chip (NoC) used for interconnecting the processor architecture's tiles, thus enabling system scalability.

The multi-core architecture is fundamentally based on the Adaptive Lockstep Processor (ALP) processor architecture. To enhance control over the individual processor cores and further enhance flexibility, the Redundancy Management Unit (RMU) is introduced. This unit is responsible to mange the redundancy of all AL processing cores within a tile.

In conclusion, this section delves into a detailed discussion of the RMU and its significance within the multi-core architecture.

6.1.1 The Overall Architecture

Modern processor architectures face the challenge of meeting growing requirements, particularly concerning the demand for increased computational power. Enhancing computational power can be achieved through two primary avenues: optimizing the performance of individual processor cores or increasing the number of cores. However, it is becoming evident that substantial improvements in single-core performance are increasingly limited, making scalability a central goal.

In designing a processor platform, it is imperative not only to address scalability but also to accommodate the stringent demands of safe mixed-criticality systems. Each of these requirements imposes distinct criteria for the platform's design. In systems with mixed criticality, there is a need to reliably determine upper bounds for critical tasks and ensure that these tasks keep to their individual Worst-Case Execution Times (WCETs). The underlying processor architecture plays a pivotal role in determining WCET, especially in the context of multi-core processors where shared resources are a primary concern. Shared resources, such as memory accessible by multiple processor cores, introduce complexities as concurrent access can lead to interference and potentially stall cores until conflicts are resolved. As the number of cores accessing shared resources grows, the likelihood of resource conflicts rises, resulting in intricate and inaccurate WCET estimations, often overestimating the WCET. Therefore, it's crucial for the processor platform to mitigate potential resource conflicts while still offering scalability to meet computational performance requirements.

Moreover, safety-critical applications impose additional demands on the processor system. Detecting and responding to faults necessitates additional hardware components or software routines. MCSs introduce additional sources of fault that are challenging to manage. A fault initiated by a non-critical task can impact the flawless execution of a critical task. Hence, the processor architecture should incorporate a mitigation mechanism to prevent fault propagation and support fault segregation, ensuring that undetected faults do not propagate uncontrollably.

The platform architecture has to fulfill the following properties:

- Scalable and Customizable
- Fault segregate
- Time predictability
- Simplification of the WCET determination.

Tile-based architectures offer a high degree of flexibility, making them well-suited for addressing complex and demanding requirements. In these architectures, one or more compute units, memories, and additional components are organized into discrete tiles, and these tiles are interconnected with one another. The scalability of such processors hinges heavily on the interconnect between individual tiles. The adoption of a (NoC) as the interconnect solution provides the essential scalability and the capacity to link together thousands of tiles [161; 162; 163].

Within the NoC, multiple routers are employed, with each router connected to a single tile. On one side of the tile, a Network Adapter (NA) serves as the interface between the NoC router and the internal tile structure. This NA acts to decouple individual tiles, playing a crucial role in memory architecture design. Notably, it enforces a No Remote Memory Access (NoRMA) memory architecture, which prohibits direct memory access from one tile to another. In this way, a tile cannot directly access the memory of another tile.

Both the NoRMA memory architecture and the decoupling property of the NA contribute to fulfilling the fault segregation requirement. They create a barrier preventing faults from propagating across the borders of a tile without traversing the NA.

Furthermore, a tile-based architecture with time-predictable interconnects proves highly suitable for time-critical applications. Decomposition approaches, in particular, are effective for determining Worst-Case Execution Time (WCET) efficiently. Individual timings of each component are precisely determined and collectively contribute to the overall WCET.

6.1.2 The Processing Tile

Each tile within the processor architecture is independently configurable, with a minimum configuration comprising a single processor

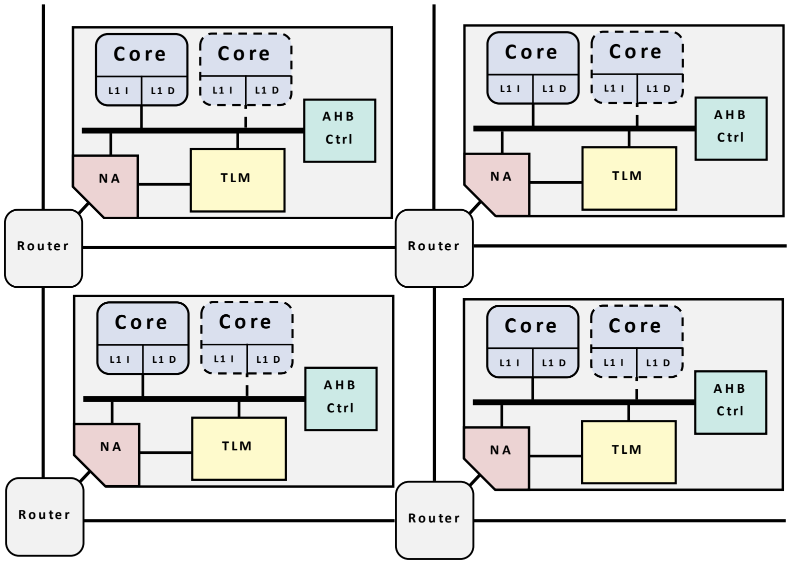


Figure 6.1: Basic platform architecture (refer to [Kem19])

core, Tile Local Memory (TLM), and the Network Adapter (NA). Additional components are optional and extend the local tile's capabilities. A tile operates as a self-contained subsystem within the complete processor architecture. The data processing of each tile is done by the ALP core with a RISC ISA. The underlying LEON3 processor architecture, which can be scaled up to accommodate up to 15 cores, equips each core with configurable Level 1 instruction and data caches. Program instructions, as well as static and temporary program data, are stored within the TLM. Access to these data occurs exclusively through a local interconnect, accessible only to the tile's Intellectual Property (IP) components. The processor cores and the TLM are interconnected through a local Advanced High-performance Bus (AHB) bus. Within a tile, each IP component is accessible through a linear address space, which remains invisible to external tiles and IP. The TLM and other IP components are protected against direct remote access, thanks to the implemented memory architecture.

The NA in each tile plays a critical role in implementing this memory architecture. It interfaces the external NoC with the internal Advanced High-performance Bus (AHB) bus, serving as the communication infrastructure. Messages and data moved trough the NoC by writing into registers within the NA. The NA offers two distinct methods for transmitting information to a destination tile, reflecting a hybrid architecture that combines transaction-based and buffer-based elements. The transaction-based architecture handles the transmission of individual data words to a destination NA, while for larger data quantities, the buffer-based architecture is better suited, employing a Direct Memory Access (DMA) configuration to transmit data. The basic platform configuration, as illustrated in Figure 6.1, encompasses all mandatory units. Each tile is situated on a grid topology and connected to one NoC router.

In addition to the mandatory components, a tile can incorporate supplementary elements connected to the AHB bus. If a tile comprises more than one processor core, an Multi-Processor Interrupt Controller (IRQMP) becomes necessary. The IRQMP manages the individual states of processor cores and can initiate, pause, and halt individual cores within a tile [18]. Typically, core 0 of each tile runs after a reset and can start additional cores within the same tile. Alongside the IRQMP, a Debug Support Unit (DSU) is an integral part of a tile. The DSU offers an external serial debug interface that is mapped to physical IO hardware pins, enabling external and independent debugging of each tile. The DSU unit provides a range of debugging features for the ALP cores, including setting breakpoints, stepping through instructions, tracing executed instructions, reading register data from the register file, and accessing status registers of each core. It supports software debugging using the GNU Debugger (GDB) and can also load an ELF file containing program code and data directly into the tile's memory.

The LEON3 processor core, IRQMP, and DSU are open-source IP components provided by Frontgrade Gaisler AB [18].

When ever the tile architecture is configured with more than one core, the previously introduced ALP architecture is employed. Alternatively, when using a single core configuration, the original LEON3 architecture is implemented. The ALP is built upon the LEON3 and maintains compatibility with the original Frontgrade Gaisler AB design, making it a seamless integrable to the broad ecosystem. Both the IRQMP and DSU serve as means to interact with the processor core, with particular emphasis on the importance of DSU. The DSU functionality is essential as it facilitates processor control and system debugging, encompassing the following features:

- Load program code to the platform
- Start, pause and stop program execution
- Debug interface (break points, GDB)
- Instruction trace
- Read out of process core state (register values).

When configuring a tile with more than two cores, the RMU comes into play. The RMU takes charge of managing the redundancy configuration of the local AL cores. An advanced platform is illustrated in Figure 6.2, where each tile within this processor architecture is comprised of six cores, a DSU, and an RMU.

6.1.3 Redundancy Management Unit

The ALP, as discussed in Chapter 5, provides a detailed discursion of the processor pipeline and core architecture. Nonetheless, it does not

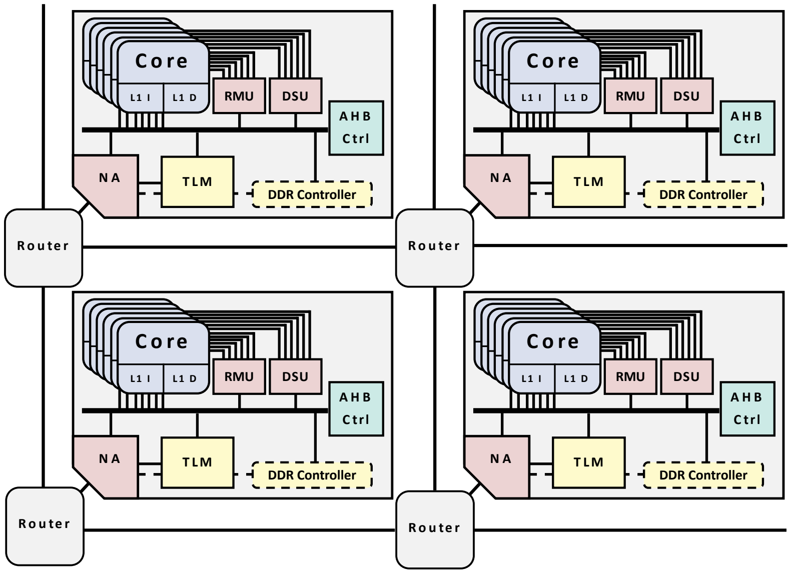


Figure 6.2: Advanced platform architecture utilizing tiles with six cores, DSU, and RMU (refer to [Kem22b])

considered how data distribution among the cores is managed. Particularly, the synchronization of the *commit stage* among two or three AL cores requires orchestration. This crucial task falls under the responsibility of the RMU, which takes charge for efficiently distributing all the necessary data among the cores within a single tile. Furthermore, the RMU introduces a layer of flexibility by allowing cores to be decoupled, providing a more versatile configuration of *logical cores*.

In Figure 6.3, it is illustrated a tile consisting of six cores, all of which are directly interconnected with a single RMU. The *commit stages* of these AL cores have a direct link to the RMU, which assumes the crucial role of supplying the proper data to all cores within a lockstep cluster. To facilitate this task, a routing complex is employed to seamlessly forward the necessary data between the cores forming a *logical core*.

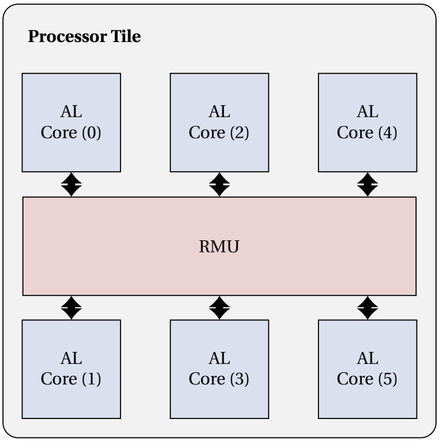


Figure 6.3: RMU manges six cores of one processing tile

Among the essential data handled by the RMU, the distribution of general-purpose register data and core results is of highest importance. As elaborated in Chapter 5, all cores within a lockstep cluster operate on a shared register file. The *master core* shares its general-purpose register file with all the slave cores. Owing to synchronized pipelines, every core simultaneously accesses the same register data. Consequently, the RMU can directly relay the output from the master core's register file to all slave cores. It is notable that only the master core is permitted to write to the common register file. Since the write accesses from both the master and slave cores are identical, these duplicate write requests are filtered out by the RMU, which the cores subsequently disregard. The data written to the register file by the master is considered faultless, as it has already undergone comparison during the commit stage.

The *commit stages* in a lockstep cluster necessitate access to results from all associated cores during the comparison or voting process. In a DMR configuration, the RMU plays an essential role in facilitating

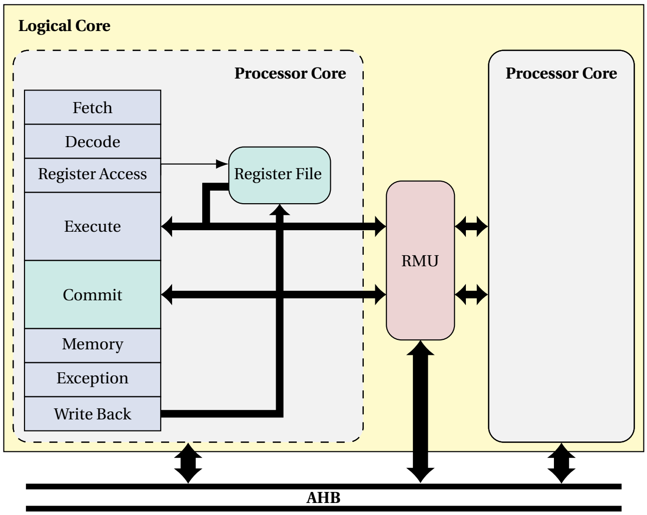
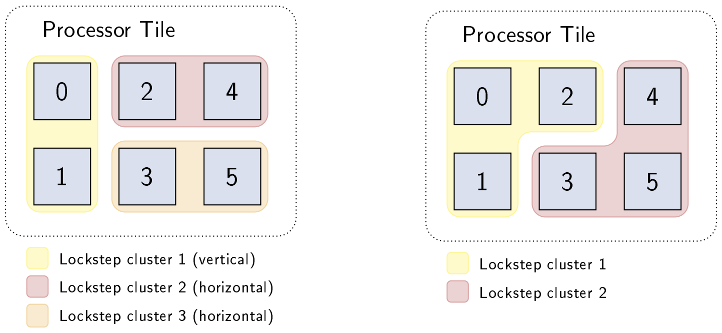


Figure 6.4: Interaction of the RMU with the processor Core.

the exchange of instruction results between the *master core* and the *slave core*, ensuring seamless bidirectional communication. However, if a TMR configuration is chosen for the logical core, each core receives results from the other two cores within the cluster. The schematic interaction between the AL processor core and the RMU is illustrated in Figure 6.4.

During split mode, the RMU omits from forwarding any data to the cores, and in turn, the cores disregard input data from the RMU. In this mode, the cores operate autonomously utilizing their local general-purpose register files. Consequently, the commit stage in this scenario discards performing any result comparisons.

The RMU is connected to the AHB bus and provides user-accessible control registers. This feature enables controllable access to the processor state of each core and individual configurations for creating


Processor Tile
0
2
4
1
3
5
Lockstep cluster 1 (vertical)
Lockstep cluster 2 (horizontal)
Lockstep cluster 3 (horizontal)
Processor Tile
0
2
4
1
3
5
Lockstep cluster 1
Lockstep cluster 2

(a) DMR configuration

(b) TMR configuration

Figure 6.5: Lockstep cluster configurations (refer to [Kem22b])

lockstep clusters. However, there are some constraints when it comes to selecting the participating cores within a lockstep cluster.

The cores within a tile are organized in rows and columns, allowing the master core of each lockstep cluster to be freely configured. Nevertheless, in a DMR cluster, the participating cores are limited to immediate neighbors. An example of a DMR configuration with one vertical and two horizontal lockstep clusters can be seen in Figure 6.5a. Furthermore, the arrangement of AL cores can allow for the formation of TMR clusters in an "L-Shape", as shown in Figure 6.5b, or in a row. These clusters can be formed and reconfigured individually during runtime.

Importantly, lockstep cluster configurations within a tile do not have to be uniform. A cluster can comprise a combination of horizontal or vertical lockstep clusters and four cores in split mode. This hardware flexibility allows programs to combine DMR and TMR configurations as needed. The cluster configuration is limited to neighboring cores to maintain manageable data routing via the switch matrix. Any faults within the switch matrix are detected during the voting process in the *commit stages*, thereby preventing the RMU from becoming a single point of failure.

The RMU serves a dual purpose. It not only offers flexibility in configuring *logical cores* but also allows for dynamic reconfiguration based on hardware-monitored conditions. The RMU employs performance counters to measure detected faults. Through a special configuration register, a threshold is defined, triggering the transformation of a DMR cluster into a TMR cluster. The user application can utilize the RMU's configuration register to specify both the threshold for reconfiguration initiation and which core should participate in forming the emerging TMR cluster. It is essential that the emerging TMR cluster adheres to the specified shape constraints.

6.1.4 The Network-on-Chip

The processor platform architecture is based on the iNoC to establish connections among individual tiles. The foundational NoC was originally introduced by J. Heißwolf and has seen subsequent feature enhancements over time [164; Ana19; 165; 166]. The iNoC design exhibits scalability, flexibility, and configurability. Moreover, for time-critical applications, it incorporates Guaranteed Service (GS) channels, guaranteeing data transmission latency. These attributes make the iNoC well-suited for meeting the specific requirements. This section provides an overview and configuration details of the iNo. Detailed and comprehensive information of the iNoC architecture is available in [164].

The NoC is set up in a 2D mesh topology with XY-routing. Tiles are positioned on a 2D grid, with each tile connected to the local port of a router. The north, east, south, and west ports of a router establish connections with neighboring routers. The router architecture employs a

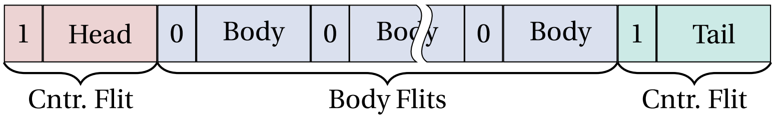


Figure 6.6: A stream of iNoC flits. The channel is established with the head flit and closed with the tail flit. Between head and tail flit all data flits are transmitted from the source to the destination. (refer to [164])

packet-switching approach with a five-stage pipeline design. Incoming data units (flits) are temporarily stored in a First in First out (FIFO) and are subsequently directed to the designated output port as per the scheduling.

The router uses a full-duplex interface, allowing flits to both enter and exit a router port simultaneously. Each input and output port provides four virtual channels, each equipped with its independent input buffers. To prevent flit loss due to full FIFO, a credit-based flow control mechanism is implemented. This mechanism operates by incrementing or decrementing the credit counter every time a flit enters or leaves the input FIFO. Each input FIFO maintains its credit counter, indicating the number of available storage slots. Consequently, when data is ready to be sent, the *req* signal is raised, leading to a corresponding decrement in the credit counter. As the virtual channels have their independent input buffers, the *vc* signal is employed to select the appropriate FIFO for data transmission.

When a flit leaves from the router through one of the output ports, it is dequeued from the FIFO. Simultaneously, the associated *ack* signal is raised, and the credit counter is incremented.

The iNoC operates using a network protocol that distinguishes between configuration flits and *data flits.* Configuration flits are employed for setting up individual router configurations. The initial *head* *flit* establishes the communication channel between the source and destination. All routers, following the XY-routing algorithm, are configured to route incoming flits to the correct output port. The initial *head flit* sets up the route and communication channel. All subsequent *data flits* within the same virtual channel make use of this established channel. To close the established channel, a *tail flit* is transmitted. Figure 6.6 illustrates a sequence of flits.

6.2 The Hybrid Network Adapter

The NA concept and its implementations are published in [Kem19]. This section provides an overview of the Network Interface (NI) and the core interface of the NA. The core interface adopts a hybrid approach by combining transaction-based and buffer-based architectures. The distinction between buffer-based and transaction-based architectures is established based on how IP is integrated and accesses the NA interface, as discussed in [167].

Buffer-based architectures employ FIFO-based or DMA-based structures to implement the NA. These architectures facilitate communication between the NA and the Network-on-Chip (NoC) through handshakes or credit signals. IP binding between the NA and IP is achieved using FIFOs or DMA along with local memory. The **DMNI** architecture [168] is an example of a buffer-based Network Adapter, seamlessly integrating FIFO-based and DMA-based architectures. The NA is directly configured by the IP, emphasizing the tight coupling of IP and local memory to the NA. Additional implementations of FIFO-based architectures are presented in [169] and [170], while DMA-based architectures are explored in [171] and [172].

In contrast to buffer-based architectures, transaction-based NAs operate independently of IP and local memory. This decoupling is achieved through bus interconnects that facilitate transactions between components, such as ARM's Advanced High-performance Bus (AHB), Advanced eXtensible Interface (AXI), the Wishbone bus, or the Open Core Protocol (OCP). Transaction-based approaches inherently ensure scalability and reusability of IP.

An adaptable NAs architecture with an interchangeable frontend is introduced in [173]. Authors in [174] and [175] present an NAs architecture designed for the AXI bus and compliant with the OCP standards.

6.2.1 The Network Adapter Architecture

The architecture of a Network Adapter (NA) comprises a backend and a frontend. The backend, also referred to as the Network Interface (NI), establishes the interface between the NA and the router of the NoC. Responsible for low-level communication with the NoC, the NI offers an abstraction service to the NA, as discussed in [161; 176]. The schematic representation of the NA is illustrated in Figure 6.7, highlighting the bidirectional link between the NI and the iNoC. The NI accesses the local port of the iNoC router, abstracting the interface and necessary protocols for the core interface.

The core interface functions as the frontend of the NA, providing a higher-level service interface to the IP components, as described in [161; 176]. In the context of the presented platform, the core interface is linked to the IP within the processing tile.

The NA employs a hybrid architecture approach, combining both buffer-based and transaction-based architectures. The key distinction lies in the frontend of the NA, where the core interface provides two interfaces to the tile IP: one for the buffer-based architecture and another for the transaction-based architecture. Both interfaces access

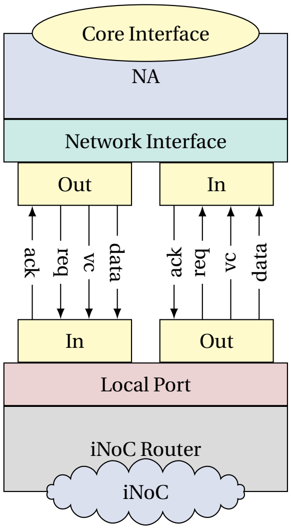


Figure 6.7: Schematic structure of the NA which is connected to the local port of the iNoC router.

the same NI, and the NA manages arbitration and access control for both interfaces to the NI.

These frontend architectures are tailored to specific tasks and address distinct requirements. The transaction-based architecture focuses on efficiently transmitting small data with minimal overhead. It is well-suited for lightweight tasks such as synchronization mechanisms. This architecture facilitates a lightweight Message Passing Interface (MPI) where individual and small messages are directly sent and received through the NoC. Data is transmitted from the source to the destination, written into local FIFO buffers for received messages at the target NA. The core interface of the transaction-based architecture allows access to this data. The interface covers basic MPI functionalities, supporting non-blocking functions for sending and receiving data through a memory-mapped interface.

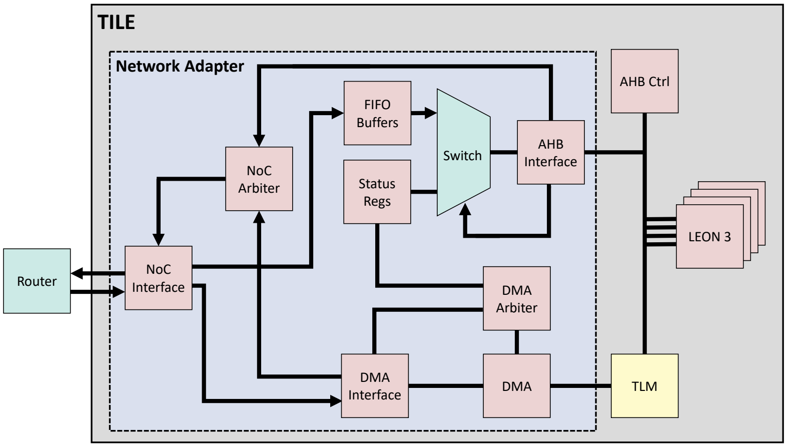


Figure 6.8: NA architecture (refer to [Kem19])

However, the transaction-based message passing does not permit direct reading and writing to the tile local memory, nor does it allow for remote memory access.

On the other hand, the buffer-based architecture of the core interface is designed for the efficient transmission of larger consecutive data. To achieve this, a DMA unit is integrated into the core interface of the NA. The DMA unit is directly connected to the TLM, enabling independent data movement without interfering with the Tile's AHB bus. This integration offers two significant advantages: first, it simplifies the WCET analyzability of the executed program, as communication does not directly interfere with local software execution. Second, it prevents interference on the local bus, resulting in improved average and worst-case performance of the processing system.

A Non-Uniform Memory Access (NuMA) is implemented by both NA architectures. This means there is no uniform and unique global address space. Additionally, the transactionbased architecture does not allow direct memory access to a remote tile, resulting in a No Remote Memory Access (NoRMA) architecture where remote data is only indirectly accessible. The NoRMA memory architecture is further constrained for the buffer-based architecture, as the DMA unit not only reads data from a remote tile but also enables writing local data to a remote tile location. Disabling this capability aligns both architectures with a NoRMA memory architecture.

The NA architecture manages outgoing and incoming data flits independently, due to the full-duplex router interface, complying with the iNoC router's full-duplex interface requirements. In accordance with the router interface, the NA implements credit-based flow control, with the NI handling flow control, router access, packetization, depacketization, and providing an internal interface to both the transaction-based and buffer-based architectures.

The internal structure of the NA is illustrated in Figure 6.8. The transaction-based architecture is implemented by the *AHB Interface* and *FIFO input buffers*. The buffer-based architecture is facilitated by the *DMA unit* and *DMA Interface*. The *DMA unit* establishes a basic interface connecting to memory and the internal *DMA Interface*. The role of the *DMA Interface* is to buffer incoming data from the *NoC Interface* and translate a DMA request into a network protocol. The translated network protocol is then forwarded to the *NoC Interface*, managing iNoC access.

During data transmission, an arbitration between the *NoC Interface* and the *AHB Interface* is necessary, as the NoC transmits only one flit per cycle. The *NoC Arbiter* facilitates priority arbitration, with the *AHB Interface* taking high priority.

The *AHB Interface* of the hybrid NA is divided into three distinct regions, each memory-mapped with a specific offset from the NA base address, as detailed in Table 6.1. Each region serves a unique purpose.

Table 6.1: Memory layout of the AHB Interface of the NA

Offset	**Regions**
0x00000	NA Status registers
0x10000	Message Passing Interface (MPI)
0x12000	DMA interface

The NA *status registers* house critical information about the NA, including the tile's unique ID stored as a hard-wired register value. Additionally, this region contains status information related to the Network Interface (NI). The MPI region is specifically associated with the transaction-based architecture, serving as the AHB interface to this architecture. Read and write accesses to this memory region correspond to MPI functionalities, such as sending and receiving data. The final memory region is intended to configuring and retrieving the status of the DMA unit within the buffer-based architecture.

6.2.2 The Transaction-based Architecture

The transaction-based architecture offers a streamlined message passing service tailored for communication between tiles, particularly optimized for brief exchanges involving a small data payload. In the realm of parallel software, effective operation requires synchronization and concurrency control mechanisms. These requirements are efficiently met by the minimal overhead transactions facilitated by the transaction-based architecture of the NA. Crucially, the NA service must protect the on-tile IP from the intricacies of network behavior.

The core interface of the transaction-based architecture must seamlessly integrate with the tile local bus, leading to the implementation of an AHB compatible slave interface. This interface provides a memory-mapped structure for MPI-related status registers, incoming messages from other tiles, and outgoing messages. Incoming data flits are buffered while awaiting retrieval by the local IP. The network interface forwards incoming flits to the core interface, where each flit is written into FIFOs accessible via the local bus, persisting until consumed.

The iNoC utilizes a packet-switching-based NoC, employing Virtual Channel (VC) to enhance network bandwidth [177]. The network router relies on a round-robin arbitration scheme for access control over shared physical links, with the arbitration dependent on the VCs associated with Quality of Service (QoS).

To harness the full capabilities of the iNoC, the NA must support VCs, involving the selection of both outgoing and incoming VCs. The outgoing VC is managed by the NI, aligning with the chosen virtual channel of the iNoC router's local port. Conversely, the incoming virtual channel is set by the local port and remains beyond control of the source tile, independent of the outgoing source VC.

A configured iNoC router with four virtual channel support enables four concurrent outgoing and incoming connections on each port, meaning the NA can establish four parallel outgoing and incoming connections simultaneously. Ideally, the NA can send and receive one flit per cycle for each of these VCs.

The provision of simultaneous outgoing and incoming links empowers application software on the local tile. By offering multiple logical links concurrently, resource conflicts are minimized, allowing for effective sharing of network resources among different software tasks running on local processing elements. These logical links, supported by the concept of VCs, can be mapped to physical resources. However, this mapping requires support from the transaction-based core interfaces.

While the NoC guarantees the order of flits within a link, the arrival order of messages from various tiles or links is unpredictable, depending on the complete history of prior and ongoing network traffic. For an established channel, the arrival time falls between the best-case and worst-case transmission times. However, flits from different links can arrive in a disordered fashion.

The route a flit takes is deterministic, with the router freely choosing an available virtual output channel. However, the virtual channel used to inject a flit into the network and the one used for its exit can differ. As a result, incoming flits cannot be directly forwarded to the receiving FIFO buffers. The transaction-based core interface introduces logical communication channels, decoupling them from the virtual channels of the network.

To facilitate this service, an interface capable of distinguishing between different incoming links is essential. The NA can control the used outgoing VC of the NoC router, but the incoming VC on the target side remains beyond the control of the iNoC network protocol. Therefore, the NA employs a protocol to assign an established link to a target channel at the destination. These target channels are FIFOs buffers accessible via the AHB bus interface of the NA, with each potential source tile and its VC having a designated target channel.

The logical communication channels are exemplified in Figure 6.9, illustrating multiple incoming connections from various tiles. Data is transmitted to the destination tile independently of the source and destination virtual channels.

The illustrated example features a 2x2 NoC configuration with four established links. Source tiles on the left side transmit data to the corresponding target tiles on the right side. *Tile 0*, for instance, has two established links routed to input VCs 0 and 1 on the destination. Sim-

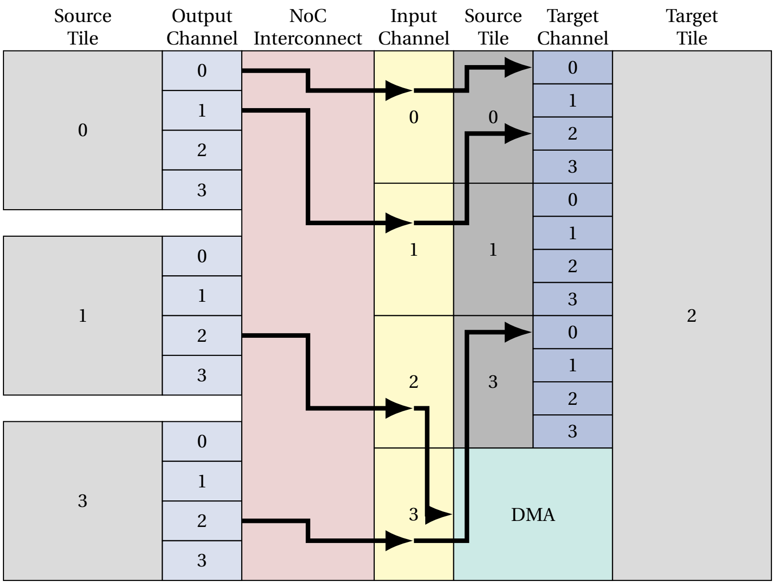


Figure 6.9: Channel assignment (refer to [Kem19])

ilarly, the connection from the output VC 2 of *tile 3* is directed to input VC 3 on the target.

In this scenario, the NI forwards incoming data flits from these connections to the transaction-based architecture. The data transfer from *tile 1* to *tile 2* employs the buffer-based architecture. Each of these connections is directed to the FIFO buffer of the target channel, with the designated target buffer associated with the input VC defined by a NA protocol.

Additionally, Figure 6.9 includes a connection associated with the buffer-based architecture. Data from the output VC 2 of *tile 1* is routed to the input VC 2 of target *tile 3*, and the input data of this VC is then forwarded to the DMA unit.

The NA supports four simultaneous incoming and outgoing connections, while the DMA unit is limited to a single connection. This limitation restricts the buffer-based architecture to handling one incoming and one outgoing data stream.

The NA facilitates communication through an *AHB Interface* to the local bus, enabling data transmission by writing to addresses assigned to the NA. The received data, forwarded from the *NoC Interface* to the transaction-based architecture, is stored in destination FIFOs, accessible via the AHB bus. The choice of FIFO depends on the source tile and the configured target channel, allowing different target channels to establish multiple connections from one source tile to the same target tile.

The transaction-based architecture employs a compact and efficient addressing scheme for sending and receiving messages. This scheme provides software with flexibility and control over the utilized resources. Consequently, the software can precisely manage the outgoing virtual channel used by the local router. The NoC link between the source and destination tile is automatically set up by the NI. An NI service automatically initiates connections, generating the necessary header flit. Each virtual channel offered by the router's local interface represents an independent link. The endpoints of these links can be located on different tiles within the network, enabling two tiles to establish multiple simultaneous links. The NI maintains these links until the software explicitly closes the connection.

Both the establishment and closure of a link are managed by the NI, and the service seamlessly conceals these configurations from the core interface and user software. It abstracts the generation of head and tail flits from the software, streamlining the communication process.

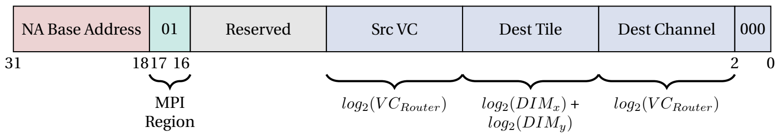


Figure 6.10: Structure of the MPI address scheme

Table 6.2: Fields of the target encoded in AHB address.

Field	Description	Size [bit]
DST X	X address of the destination	$log_2(DIM_x)$
DST Y	Y address of the destination	$log_2(DIM_x)$
VC	Virtual Channel	$log_2(VC_{Router})$
TC	Target Channel	$log_2(VC_{Router})$

Message reception involves the retrieval of data from the FIFOs. The address structure follows the pattern illustrated in Figure 6.10. Starting with the target channel ID and followed by the source tile ID, the address scheme is shown in Table 6.2 for field size reference. To ensure the extraction of valid data from the FIFO, the NA offers a register containing the current number of elements in the FIFO to the AHB bus.

For writing data to a designated destination, the AHB address space is also employed. This space configures the target tile, the source VC, and the destination channel where the data is to be written. The address structure follows a sub-address format, initiating with the target channel, followed by the target tile, and concluding with the source AHB. The decoded source VC configures the outgoing VC of the *NoC Interface*. The channel destination is defined based on the tile ID and the target channel.

Table 6.3: DMA Configuration Registers

Offset	Register Content	Access
0x0	Local Addr	R/W
0x4	Remote Addr	R/W
0x8	Remote Tile	R/W
0xC	VC Local Tile	R/W
0x10	Size	R/W
0x14	Write/Read	R/W
0x18	Send VC Remote Tile	R/W
0x1C	Input Valid	R/W
0x20-0x34	Reserved	-
0x38	Read Active (Status Reg)	R
0x3C	Write Active (Status Reg)	R

6.2.3 The Buffer-based Architecture

The buffer-based architecture is designed for the efficient transmission of large data blocks, operating independently of the AHB bus and the LEON3 or AL processor.

To achieve this independence, a DMA controller with direct access to the TLM is implemented. To facilitate simultaneous access by both the DMA controller and AHB, the TLM incorporates a dual-ported memory interface. One port is dedicated to the AHB bus, and the other serves as a dedicated interface to the DMA controller within the NA.

Ensuring time predictability is a key consideration for the DMA unit. To achieve this, a single one-way channel is utilized, guaranteeing predictable timing and minimizing interference. The DMA unit can be enabled for either sending or receiving data blocks, enhancing the analyzability of DMA transactions. This configuration supports the overall goal of maintaining efficient and predictable data movement within the system.

The *DMA Unit* offers extensive configurability, as outlined in Table 6.3. This encompasses adjustable parameters such as block size, and configurable read and write addresses. The block size is specified in words, and addresses follows an alignment with the word size (32-bit).

Key components of configuration include the *Local Addr* register, indicating the starting address of the data block in the local TLM. The *Remote Addr* register points to the start address of the data block in the remote TLM, with the remote tile specified by the *Remote Tile* register. The *Size* register defines the block size in words, and the *Write/Read* register determines whether data is written to or read from a remote tile. The outgoing virtual channel used is configured by *VC Local Tile*.

As illustrated in Figure 6.8, the DMA controller can be configured by either a local processor core or the *DMA Interface Unit*. While the local processor core configures the *DMA Unit* accessible via the local bus, the buffer-based architecture necessitates the use of *DMA Units* at both the source and target NA. The source *DMA Unit* fetches data from the TLM and sends it to the target NA. On the target NA, the DMA unit must be configured to receive and write the data to the correct TLM location.

When reading from a remote tile, the outgoing virtual channel of the remote tile must be specified, accomplished through the *Send VC Remote Tile* register. These 32-bit registers cannot be written simultaneously; hence, the *Input Valid* register confirms the configuration and initiates the data transfer. The *Read Active* and *Write Active* registers indicate the current status of the *DMA Interface*, specifying whether a read or write process is active.

While one *DMA Unit* is configured directly by the local processor, the other *DMA Unit* is configured through a buffer-based protocol. The *DMA Interface* generates configuration flits at the start of each data transfer, utilized by the target *DMA Interface* to configure the *DMA*

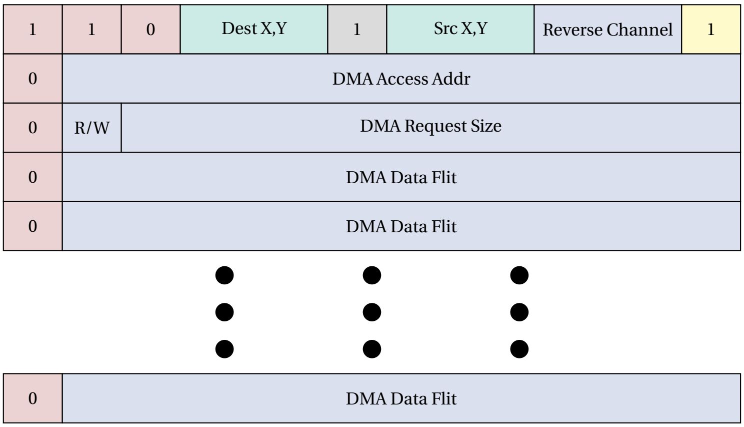


Figure 6.11: Protocol used to configure the NA and DMA on destination side.

Unit. This protocol is illustrated in Figure 6.11: the configuration body flit initiates the configuration of the target *DMA Unit*, followed by the data flits intended for writing.

To monitor the state of the DMA, registers detailed in Table 6.4 are employed. These registers, accessible through the AHB interface of the NA, provide valuable insights into the DMA state and are located in the memory region of the *DMA Interface*.

The *DMA Unit* employs a pipelined design, where data read by the unit is temporarily stored in a buffer. Writing operations involve reading the temporarily stored data and writing it to the output. If the DMA buffer is empty or the counterpart is unable to store the data, the writing process is stalled. This deliberate stalling ensures data integrity, as a stall in the writing process corresponds to a stall in the reading process, and vice versa. Read operations are halted when no data is available at the input.

Table 6.4: DMA Status and Debug Registers

Offset	Register Content	Access
0x40	Local Addr	R
0x44	Remote Addr	R
0x48	Remote Tile	R
0x4C	Remote VC	R
0x50	Size	R
0x54	Write/Read	R

Given that the DMA utilizes a unified interface for both TLM and NoC connections, the *DMA Interface Unit* serves as an intermediary between the DMA and the *NoC Interface Unit*. The DMA can read and write to the *NoC Interface Unit* in the same manner as it does with the TLM. Depending on the operation, the *NoC Interface Unit* responds accordingly. Reading data involves forwarding data flits as soon as they become available. On the other hand, writing data through the NoC necessitates a DMA protocol that configures the target DMA. The employed DMA protocol is lightweight, with the first flit containing the data block size and the second flit specifying the write address.

6.2.4 The NoC Interface

The *NoC Interface Unit* serves as a unified interface connecting the NoC router, the AHB Interface of the transaction-based architecture, and the DMA Unit. It functions as a Network Interface (NI), providing a service to the *core interface* while abstracting the intricate interactions between the *core interface* and the iNoC router. Its major responsibilities encompass:

- Packetization and depacketization
- Flow control

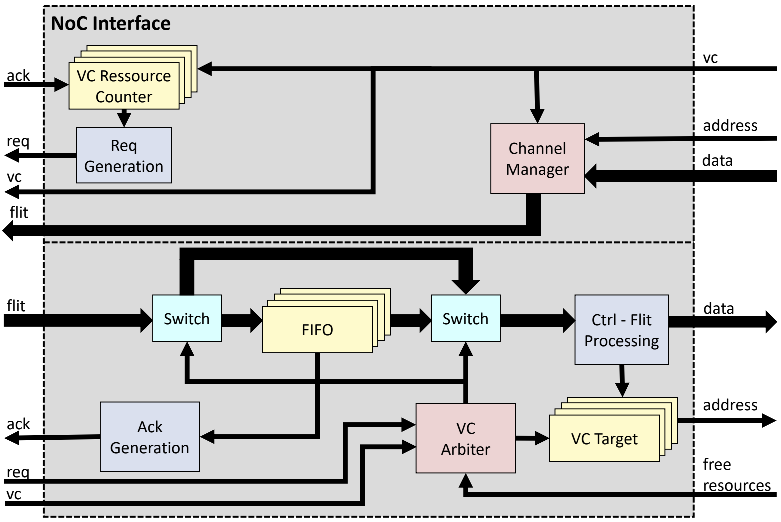


Figure 6.12: Architecture of the NoC interface (refer to [Kem19])

- Channel management.

The *NoC Interface* architecture efficiently manages the distinct tasks of sending and receiving data, aligning with the iNoC router's independent handling of flit transmission and reception. This structural division is evident in the *NoC Interface* architecture, as illustrated in Figure 6.12, where separate components handle sending and receiving operations. The upper section of the *NoC Interface* is dedicated to sending flits, while the lower section manages incoming flits from the router.

Received flits from a virtual channel are seamlessly directed by the *NoC Interface* to their designated destination target. The configuration of the destination target is determined by the head flit received during channel establishment, and this configuration is stored in the

1	0/1	0/1	X,Y	1	X,Y	Channel	0/1
Ctrl. Flag	Header Flit	Tail Flit	DST Address	Service Level	SRC Address	Target Channel	DMA

Figure 6.13: Flit structure of a ctrl flit (refer to [Kem19])

payload of the control flit, following the structure outlined in Figure 6.13.

The process of packetizing outgoing data is contingent on the current channel state, carefully monitored by the *Channel Management Unit*. The NoC interface autonomously terminates a channel connection for an outgoing virtual channel when the destination undergoes a change. This is accomplished by dispatching the tail flit of the established connection to effectively close the channel. Subsequently, when an outgoing virtual channel lacks an assigned channel, a new channel between source and destination is initiated by transmitting a head flit to the destination NA. This head flit configures both the routers in the NoC and the NA, ensuring the seamless establishment of the requisite connections.

To determine the readiness of the router to receive a new flit, the *NoC Interface* meticulously monitors virtual channel resource counters. These counters are decremented upon flit transmission (*req*) and incremented when acknowledgments (*ack*) are received from the router's virtual channel (VC). This mechanism ensures the required flow control necessary for the correct router's operation.

For incoming flits from the router, effective flow control mandates the availability of a FIFO buffer for each virtual channel. Consequently, the *NoC Interface* offers FIFO buffers to store incoming flits as needed. Depending on the free buffer space in the designated data target, the data is either directly forwarded or stored in the FIFO buffers. Direct forwarding occurs, bypassing the FIFO, if the destination buffer associated with the receiving virtual channel has available resources. In cases where no free resources are available for immediate data write, the FIFO buffers come into play, and no acknowledgment is sent to the router. As soon as free resources become available, a flit is dequeued from the FIFO buffer, relieved by the VC Arbiter Unit upon obtaining access. The VC arbiter utilizes round-robin arbitration, crucial for cases where flits are stored in the FIFO buffers. All VCs with received flits and available target resources are considered during the round-robin arbitration process.

The allocation of the destination buffer is managed by the head flit, processed in the *Control Flit Processing Unit.* This unit configures the virtual channel's target, and each incoming data flit is forwarded based on the VC target table. This table encompasses a destination address for each input virtual channel, guiding the transaction-based or buffer-based architecture in processing the data flit according to the designated destination.

6.2.5 Timing Analysis

The calculation of Worst-Case Execution Time (WCET) for application software involved in sending and receiving data depends on the Worst-Case Transmission Time (WCTT) of data packages. The WCTT represents the maximum time it takes for a data package to reach its destination. The analysis of WCTT employs a composition approach, integrating the worst-case latencies of individual network components participating in data transmission. The cumulative worst-case latency of each component contributes to the determination of WCTT. This analytical process necessitates the ability to establish a new channel from the source to the target tile. Therefore, the presence of available free NoC resources is essential for initiating a new GS channel establishment.

$$WCTT = L_{NAs} + L_{NoC} + L_{NAr} \tag{6.1}$$

Where L_{NAs} is the maximum latency of the sending NA, L_{NoC} is the maximum latency of the NoC and L_{NAr} is the maximum latency of the receiving NA. L_{NoC} donates and expresses the results of [164] with our NoC configuration.

$$L_{NoC}(S_{pkt}, H) = 4 * (5H + S_{pkt} - 1) \tag{6.2}$$

Let H represent the number of hops taken by the package, and S_{pkt} denote the package size.

The calculation of worst-case latency for L_{NAs} necessitates distinguishing between a transaction-based transfer and a DMA transfer. Given the prioritization of the transaction-based transfer, the latency $L_{NAs,trans}$ remains unaffected by DMA behavior. In this scenario, the NA incurs no additional latency as data from the AHB bus is directly forwarded to the *NoC Interface*. The *NoC Interface* adjusts the package size based on the channel's state. If the channel is already established, the package is directly sent. However, the process of closing old channels introduces an additional flit, as does the initiation of a new channel.

$$L_{NAs,trans} = 0 \tag{6.3}$$

The package size in a transaction transfer is contingent on both the size of the packages to be sent and the existing state of the channel. Especially, DMA transfers do not disrupt transaction-based transfers, thanks to their prioritization.

$$S_{pkt,trans}(S_{pkt}, c) = S_{pkt} + c_i \tag{6.4}$$

The term c_i is used to determine the current state of the channel i.

$$c_i = \begin{cases} 0, & \text{channel established} \\ 1, & \text{no channel established} \\ 2, & \text{wrong channel established} \end{cases} \tag{6.5}$$

The latency of DMA transfers is influenced by the time taken to read the data (L_{Read}), the arbitration latency ($L_{DMAarbiter}$), and the latency associated with the DMA pipeline ($L_{DMApipeline}$).

$$\begin{aligned} L_{NAs,DMA}(L_{DMAarbiter}, L_{Read}, L_{DMApipeline}) = \\ L_{DMAarbiter} + L_{Read} + L_{DMApipeline} \end{aligned} \tag{6.6}$$

The configuration uses a read latency of one, with $L_{DMAarbiter}$ set to 1, and $L_{DMApipeline}$ set to 1. The package size transmitted in a DMA transfer is contingent upon the amount of transaction-based data S_{trans}, the size of DMA data S_{dma}, and the current state of the channel. Additionally, the DMA network protocol introduces two configuration packages, while the channel establishment in the transaction transfer adds an extra two packages. Moreover, the reestablishment of the DMA channel contributes an additional two packages.

$$\begin{aligned} S_{pkt,dma}(S_{dma}, S_{trans}, c_i) = \\ 2 + S_{dma} + 5S_{trans} + c_i \end{aligned} \tag{6.7}$$

Receiving flits does not introduce any extra flits to the transfers. The latency is influenced by the *NoC Interface* and subsequent units. The *NoC Interface* latency is determined, in part, by the possible VC arbitration, which is contingent on the number of utilized VCs (vc). Transactions are then forwarded to the FIFOs, contributing a latency of one (L_{FIFO}).

$$L_{NAr,trans}(vc, L_{FIFO}) = vc + L_{FIFO} \tag{6.8}$$

In the case of DMA transfers, the*NoC Interface* latency must be considered and added by the buffer latency in the *DMA Interface* ($L_{DMAinterface}$ = 1) and the latency introduced by the *DMA Pipeline* ($L_{DMApipeline}$ = 1).

$$\begin{aligned} L_{NAr,DMA}(vc, L_{DMAinterface}, L_{DMApipeline}) = \\ vc + L_{DMAinterface} + L_{DMApipeline} \end{aligned} \tag{6.9}$$

6.3 The Runtime Environment

This section is based on the publication by [Kem22b] [1] Furthermore, the section builds on the work of Kühbacher et al. [64; 143; 178]. The original work focus on implementing a solution for the Kalray Massively Parallel Processor Array (MPPA). This section demonstrates the seamless integration of a coarse-grained dataflow approach into the presented platform and emphasizes the advantages derived from the ALP. The Runtime Environment (RTE) is introduced to provide a deeper understanding of the subsequent platform evaluation, utilizing dataflow graphs as inputs for algorithm descriptions.

[1] This represents a collaborative effort demonstrating a holistic hardware-software approach. It combines an adaptive and predictable platform with the Runtime Environment (RTE) based on coarse-grained dataflow. All contributions related to the hardware aspects of the platform and the Hardware Abstraction Layer (HAL) (e.g. driver of the Adaptive Lockstep Processor (ALP) and communication driver for the network adapter) are provided by the author of this thesis, while the software design and the Runtime Environment (RTE) are contributed by third parties.

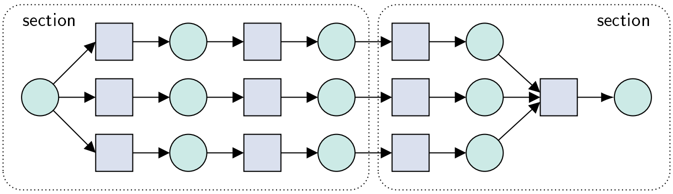


Figure 6.14: Example dataflow graph (refer to [Kem22b])

6.3.1 Dataflow Graphs

The RTE is specifically designed for tile-based processor systems, such as the Kalray MPPA or the multi-core system detailed in this section. It employs a static coarse-grain dataflow approach to define algorithms. In this approach, dataflow actors work with data collections, rather than individual values, and apply functions to process the data, rather than executing single instructions.

The application developer specifies the application programs for the RTE in the form of Directed Acyclic Graphs s (DAGs). These DAGs are compatible with the RTE and are structured as bipartite graphs consisting of actor and data nodes.

Actor nodes, which are referred as *actors* from this point onward, contain information regarding how data is processed. They encapsulate the functions applied to the input data, with the input data represented as data nodes within the DAGs. These data nodes explicitly model memory requirements and aid in data management. In the architecture being presented, the data within these nodes is allocated to the TLM. It is important to note that data can be distributed across multiple TLMs simultaneously.

Each actor node is associated with a function that has a corresponding number of parameters. Because the execution model is based on dataflow, these functions are inherently pure, meaning they are free from side effects and do not independently allocate memory. To overcome potential memory limitations on compute tiles, each function is typically attached to multiple actor nodes, enabling the accomplishment of more extensive computational tasks, such as a large FFT.

Dataflow graphs are particularly well-suited for redundancy in execution. Individual nodes can be duplicated (DMR) or triplicated (TMR), and the results of these nodes can be compared and checked for errors.

The model of the dataflow graph introduces sections to allocate non-functional like safety features during the execution of the graph. These sections are somewhat analogous to the program sections introduced in a previous Section 4.1.2, but the sectioning in the dataflow graph is more coarse-grained, encompassing a single actor or multiple parallel actors. In contrast, the section introduced in the earlier chapter is finer-grained and is limited to a single tile. The implementation of the ALP maps this to the redundant execution at a SoR on instruction level.

During the execution of the graph, the redundancy configuration of each graph section can be modified individually whenever the execution reaches a state between two graph sections. The decision to section a graph is a design choice that application developer of the RTE must make when specifying a graph.

In Figure 6.14, an example graph with two sections is shown. Data nodes are illustrated as green filled circles, while blue boxes represent actor nodes. Upon successful completion of an actor node, the associated data nodes are consumed.

6.3.2 Implementation on the ALP-based Multi-Core Architecture

The RTE necessitates the presence of a tile with higher memory requirements. In the context of the multi-core architecture being presented, one of the tiles is directly connected to a large DDR memory. This DDR memory serves as a central component for storing entire dataflow graphs because they are typically too large to fit within the TLMs.

As a result, TLMs are primarily responsible for temporarily storing specific data nodes and actors during the execution of the graph. The tile linked to the DDR memory is referred to as the *driver tile*. It serves as an orchestrator, responsible for distributing tasks to the compute tiles. The driver tile's role includes distributing the graph nodes stored in the DDR memory to the TLM memory of the compute tiles. Given that graph nodes are typically larger than just a few bytes, the RTE employs DMA transfers, utilizing the DMA unit of the NA (as detailed in Section 6.2.3).

The driver tile operates as a control entity, overseeing the progress of execution, managing all data transfers, and monitoring the execution of actors within the system. After an actor completes its execution, the compute tile collects the data from the distributed TLMs.

On the other hand, compute tiles are not self-initiating and must await MPI messages from the driver tile. The driver tile uses MPI messages to instruct the compute tiles. These messages can contain various types of information, such as signaling that the next actor is ready for execution, notifying that a DMA transfer affecting the TLM has been completed, or indicating that a data node should be removed from the TLM.

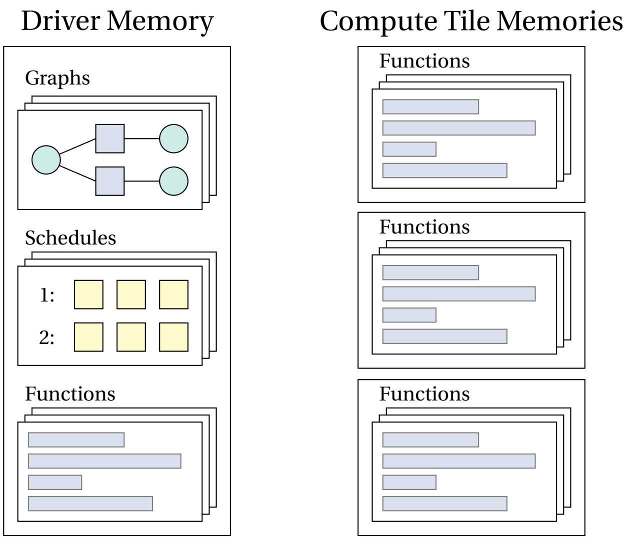


Figure 6.15: Information permanently stored in memory (refer to [Kem22b])

Given the limited size of TLMs, they can only accommodate a small number of data nodes with constrained sizes. When an actor processes substantial amounts of data, it is executed on the driver tile. While occasional actor executions on the driver tile are generally tolerable, frequent data processing on the driver can result in suboptimal performance and should be minimized. The RTE does not automatically partition extensive data portions, so it falls to the user to specify graphs in accordance with the hardware constraints.

Figure 6.15 provides an overview of the information permanently stored in the memories on compute tiles and the driver tile in a system with three compute tiles. Concerning actual data (i.e., the content of data nodes), each TLM has a statically allocated memory section, which remains consistent across all compute tiles. The driver tile knows the

the memory address and size of this memory section, it can initiate DMA transfers between the DDR memory and a TLM without requiring a complex communication protocol. The communication is completely abstracted and handled by the buffer-based architecture of the NA.

Due to the limited sizes of TLMs, dataflow actors are assigned to tiles rather than individual cores. To fully utilize all the cores within the hardware architecture, it is essential for the user's functions to inherently support parallelism. The AL approach emphasize the importance of dataflow actors being executable with varying core counts. Consequently, the RTE allows users to specify which segments of the actor code can be executed in parallel but not the precise core on which the code will run. The RTE offers a parallel for-loop construct well-suited for this purpose.

One significant advantage of the RTE is that application developer are relieved of managing synchronization between tiles, as the RTE automatically handles the necessary DMA and message transfers, abstracting these complexities from the user. However, it is upon the user to ensure the correct synchronization within functions executed by dataflow actors, particularly between the cores on a tile.

6.3.3 Fault Tolerance

The RTE provides support for both software and hardware fault tolerance. A designated section within the dataflow graph, marked with a redundancy property, prompts the RTE to apply the specified safety mechanisms.

When software redundancy is activated, each actor within the redundant section is duplicated and distributed across different compute tiles. The RTE leverages the fault segregation property of the tile-based architecture to prevent fault propagation from one compute tile to another. Each of these distinct compute tiles redundantly executes the same actor. Furthermore, the compute tiles compute a checksum in addition to the actor execution. Following the execution of each actor, the checksum is transmitted alongside the data to the driver for comparison. If the checksums do not match, error correction is applied, and the number of redundant executions determines the corrective action.

Similar to hardware redundancy provided by the ALP, the RTE supports both DMR and TMR configurations at the software level. In the case of DMR, the driver can initiate re-execution on the respective tiles by sending a small notification message. When an actor is executed on three different tiles, the driver identifies the correct checksum through a voting mechanism and instructs the tile with the incorrect checksum to discard its result. As mentioned in earlier sections, changes in redundancy can occur during runtime, but they impact entire sections of the graph and are only possible when the execution reaches a barrier between two sections.

Additionally, the RTE offers the option to execute dataflow actors with hardware redundancy, which is limited to processor cores within a single tile. The RTE utilizes the RMU to configure processor cores into DMR or TMR lockstep clusters when necessary. Similar to software redundancy, decisions regarding hardware redundancy are made by the RTE only between graph sections. In contrast to the sequential RTE management code, executing dataflow actors in a lockstep configuration reduces overall performance since the effective number of cores is halved (or reduced by a third in the case of TMR). This performance reduction is predicated on the assumption that actors offer sufficient parallelism to utilize all cores on a tile.

Table 6.5: Possible redundancy combinations supported by the RTE (refer to [Kem22b])

RTE redundancy		**actor execution redundancy**		
NO RED.		NO RED.		NO RED.
HW DMR	×	HW DMR	×	SW DMR
HW TMR		HW TMR		SW TMR

Furthermore, the RTE supports the combination of hardware and software redundancy. Hardware and software redundancy complement each other effectively, as the RTE management code, such as the preparation of transfers, which is not part of the software redundancy concept, can only be executed with hardware redundancy. This code is inherently sequential, and using two or three cores in a lockstep configuration has a negligible impact on performance.

Hardware and software redundancy within the RTE serve different purposes. Hardware redundancy operates exclusively within a tile (intra-tile), involving the reconfiguration of two or three cores on the same tile to create a lockstep processor. In contrast, software redundancy permits redundant execution of actors across two or three different tiles (inter-tile). As dataflow actors can concurrently utilize both hardware and software redundancy, the RTE can accommodate a range of redundancy configurations, providing redundancy levels of up to ninefold. For a comprehensive overview of all available redundancy combinations supported by the presented RTE, please refer to Table 6.5.

Table 6.6: Implementation results of resources and power (refer to [Kem19])

	LUTs	CLBs	FFs	BRAMs	Power (W)
Network Adapter	**3230**	**633**	**935**	**0.5**	**0.022**
NoC Interface	528	197	134	0	0.004
FIFO Buffers	253	92	84	0.5	0.004
AHB Interface	70	47	2	0	0.001
Status Register	204	100	9	0	0.002
DMA Arbiter	88	35	0	0	0.000
DMA Interface	437	118	99	0	0.000
DMA	1649	288	540	0	0.008

6.4 Evaluation

The tile-based multi-core architecture is evaluated in regard of its network performance and its WCET analyzability. The evaluation concludes with an evaluation of the RTE executed on the multi-core architecture. Different algorithm and redundancy configurations of the RTE are used to compare the runtime behavior.

6.4.1 Network Adapter

A VHDL implementation of the NA is used evaluated the required hardware resources and the DMA performance. Therefore, the developed prototype HDL designed is implemented for the AMD Virtex UltraScale+ Evaluation board VCU118, containing the XCVU9P-L2FLGA2104E FPGA.

For synthesis and implementation Vivado 2017.4 is used. For comparative reasons the standard settings are used. Specific optimizations strategies provided by the tool are not used. The NA is configured for a 4x4 NoC. The results of the implementation are shown in Table 7.1.

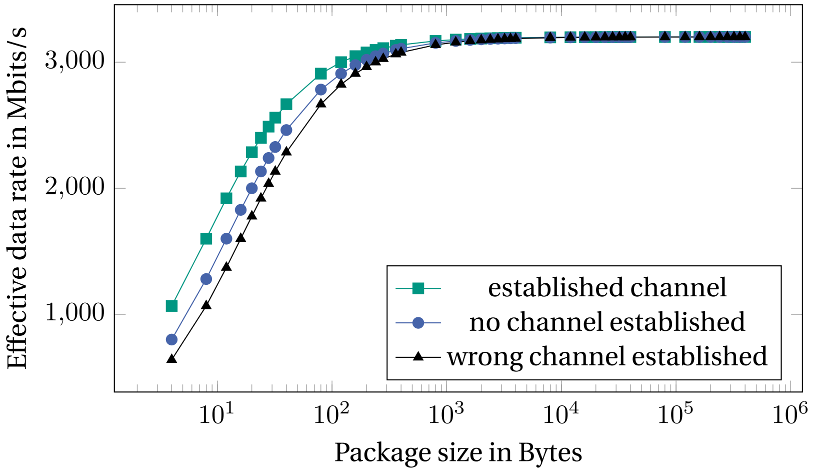


Figure 6.16: Effective best-case throughput (refer to [Kem19])

The implementation results shows that the NA design is dominated by the size of the DMA, while the transaction-based architecture including the *NoC Interface* requires approximately the same amount of resources. The mismatch of summed up resource and the complete NA design leads back to the connecting resources and is negligible. The lower amount of Configurable Logic Blocks (CLBs) comes from components sharing the same CLB.

Beside the required implementation resources the NA is evaluated regarding the DMA transfer capability. Therefore, the actual throughput of different data package sizes, under best case (Figure 6.16) and worst case (Figure 6.17) condition are measured. During a DMA transfer no further transfers are initiated from the source tile. For best-case conditions are converging to the theoretical maximum throughput of 3200 Mbit/s at a package size of 2 KB. The same behavior is seen under all conditions of the channel, as the overhead gets negligible. For

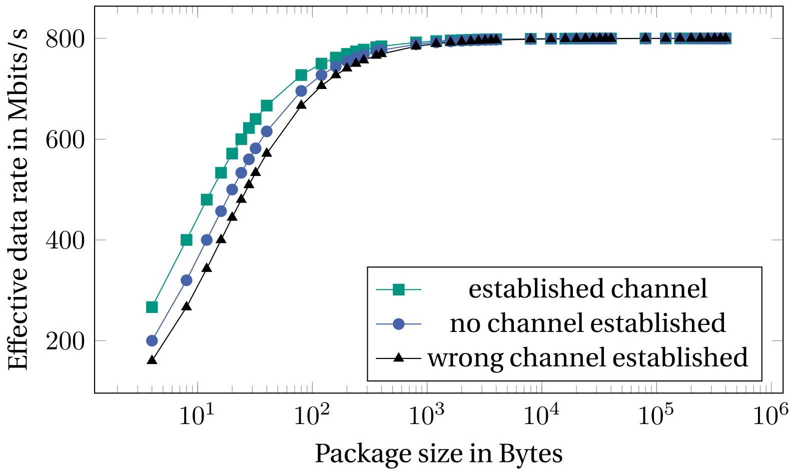


Figure 6.17: Effective worst-case throughput (refer to [Kem19])

worst-case similar results are observed, as the theoretical maximum throughput is 800 Mbit/s.

6.4.2 Worst-Case Execution Time

The platform is evaluated in regard to its WCET behavior. Therefore, the ARGO toolchain is used to a generated parallelized C code which can be deployed to the presented multi-core architecture. The ARGO toolchain can generate WCET-aware parallelization of a originally sequential C code [Red19; 179; 180]. The ARGO toolchain supports tile-based architectures like the presented one. For the communication between processing elements it assumes directed GS connections. These connection can be mapped to the MPI interface of the presented NA.

For application's WCET calculation the ARGO toolchain uses a timing-compositional approach. It analyzes the WCTT of the communication between processing elements and the WCET of processing elements independently. The WCTT is calculate for each used communication channels. The channels are individually analyzable and a WCTT is predicated. For the analysis it uses the presented timing model of Section 6.2.5 to determine the WCTT. The application code executed on the processing elements is analyzed by the aiT tool from AbsInt [181]. The final application's WCET is predicated by composing the individual WCTTs and program WCETs.

For the WCET evaluate a Terrain Avoidance and Warning System (TAWS) application is used. The TAWS is an algorithm used in avionics e.g. for vision-based aircraft navigation systems such as [182]. It is an safety critical application to warn the airplane pilot from dangerous approach to the terrain. Therefore, the TAWS uses different input parameters like rate of descent or altitude. For further implement details refer to [Red19; 183]. It is a dataflow dominated application suitable for parallelization.

The compiled program code is evaluation on a FPGA prototype. The VHDL implementation of the is implemented for the AMD Virtex UltraScale+ FPGA VCU118 Evaluation Kit. The platform is run at 100MHz and uses a 2x2 configuration. Each tile is equipped with a TLM of the size of 512 Kbyte and one processor core. The L1 instruction and data caches of the processor cores are configured as 8 Kbytes two-way associative cache with Least Recently Used (LRU) replacement strategy.

The aiT tool for the WCET prediction supports the LEON3 core [184]. However, the tool is based on a exacted model of the processor pipeline. The prevent the influence of the ALP modification to the LEON3 processor the platform uses the original processor implementation. The

Table 6.7: Execution times in µs for the TAWS application on the multi-core processor platform (refer to [Red19])

Scenario	WCET	Measured
Sequential code	7788 µs	672 µs
Parallel w/o comm. opt.	4792 µs	641 µs
Parallel code	4410 µs	524 µs

cores in the design do not have a hardware FPU to accelerate floating point operations.

Table 6.7 shows calculated WCET and the measured execution time in µs for the TAWS on the prototype platform. The ARGO tools generated a parallelization that utilizes all four cores of the platform while the data fields fitted into the TLMs. The tools were configured to generate two different parallelized c code versions. One version without communication optimizations and one with. This is done to demonstrated that the parallelized code does not exceed the calculated WCET. Further details about the ARGO toolchain and parallelized code generation of the TAWS can be found in [Red19].

6.4.3 Runtime Environment

The runtime behavior of the in Section 6.3 presented RTE is evaluated. The evaluation considers different hardware and software redundancy configurations. The for the evaluation used hardware platform uses a configuration with a 2x2 NoC. Each of the tiles is configured with six ALP. The mandatory TLM has a size of 1 MiB TLM. In addition to the TLM one tile directly accesses a 1 GiB of DDR memory. The driver tile is mapped to the tile connected to the DDR memory. The remaining three tiles act as compute tiles. The compute tiles receives computation tasks from the driver tile.

The multi-core architecture does not have a firmware distributing program code to other tiles and start the processor cores remotely. Therefore, the DSU of each tile is used to distribute the program code of each tile. The RTE program code of the driver and computational tile is loaded to the associated TLM and start the program execution of the tile.

From this 1 MiB TLM, the RTE uses 728 KiB to store up to eight data nodes with a size of up to 91 KiB, the rest is reserved for code, other RTE-related data and runtime stacks for the six cores.

To evaluate the supported redundancy configurations, three dataflow graphs for common computation tasks are constructed, namely matrix multiplication, Fast Fourier Transform (FFT) and bitonic sorting. For the matrix multiplication benchmark, a directed acyclic graph mimicking the data dependencies in Cannon's algorithm is created. The Cannon's algorithm is commonly used for distributed matrix multiplication [185]. To utilize all cores on the tiles, Cannon's algorithm is used in a block-wise fashion, i.e. matrices are divided into blocks and actor executions correspond to multiplications and additions of these blocks.

The evaluation uses two 600×600 integer matrices. These are divided into 16 square blocks so that each block had 150 rows and columns. For the two other benchmarks, similar approaches are used to split the input vector into smaller parts which are processed by the dataflow actors. To run the FFT benchmark, a graph based on the data dependencies of the Cooley-Tukey algorithm is constructed [153]. The input was a vector of 2^{16} complex numbers with 64-bit floating-point real and imaginary part which was split into 16 parts.

Because the ALP does not support a floating-point unit the RTE software is compiled with software floating-point arithmetic. As a result, the FFT benchmark is relatively slow compared to the other benchmarks which only use integer arithmetic. The software arithmetic to calculate the floating-point introduces non parallelized computation.

The bitonic sorting algorithm can be viewed as a dataflow graph. This dataflow graph is mapped to the RTE. The input vector of this benchmark consisted of 2^{19} integer numbers split into 32 parts.

The results of the evaluation results are shown in Figure 6.18. The benchmark applications are executed with different redundancy configurations.

First, it is also noticeable that, for example, some executions in DMR configurations take slightly less than twice as long. The reason for this is that only those parts of the execution which utilize all cores on a tile cause a performance drop when redundancy is switched on. Thus, only redundant actor executions affect the performance since the RTE does not transfer data redundantly and RTE management code only uses one core per tile.

Another noticeable factor is that software DMR is slow compared to the other configurations and takes as much time as software TMR. The reason lies in the structure of the used evaluation hardware platform. While three compute tiles work well for the TMR configuration, this number of tiles is rather unsuitable for the DMR configuration. The distribution of the computation task can not fully utilize all available hardware resources. The under-utilization lead for the larger number of compute tiles results that the performance is similar to hardware DMR like in case of TMR. Since the overhead of the software redundancy approach is low compared to the actual actor executions and the hardware maintains its clock speed in lockstep mode, both TMR approaches lead to similar performance. The measurements in Figure 6.18 show the execution times when no fault occurred.

Lastly, the three benchmark applications are used to measure the overhead caused by the additional commit stage in the pipeline. The left-

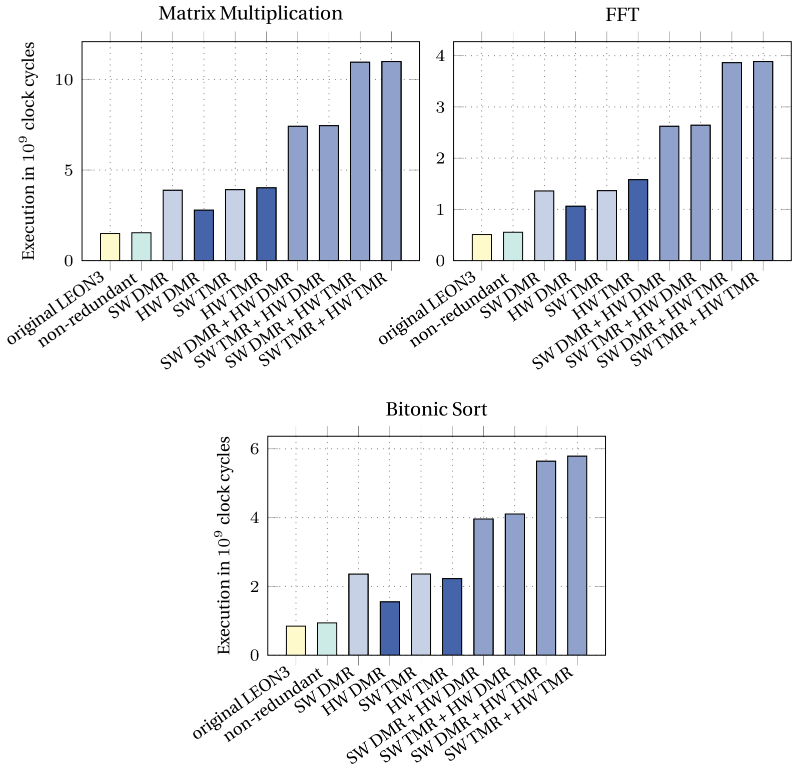


Figure 6.18: Dataflow execution times of benchmark applications for different redundancy configurations (refer to [Kem22b])

most bars (yellow) in Figure 6.18 show the execution times when standard LEON3 cores instead of the modified Adaptive Lockstep Processor (ALP) are used. The overhead is smallest for the matrix multiplication benchmark, where the execution took only 3.0% longer than on standard LEON3 cores, and largest for bitonic sorting with 11.2% the measured overhead of the FFT is 8.9%. This result is not surprising since bitonic sorting is a comparison-based sorting technique which relies on branch instructions to sort the data elements.

6.4.4 Evaluation Fault Injection

To ensure the proper functioning of the system in the event of a fault, the fault injection unit presented in Section 5.5.1 is utilized for the multi-core platform. Each tile on the evaluation platform is equipped with an individual fault injection unit, offering precise control over fault injection. Faults are injected directly into the result of the execution stage of ALP architecture. The placement of the fault injection between the execution stage and the commit stage allows emulation of both Single Event Upsets (SDC) and flawed control flow. In the execution of a jump instruction, the result from the execution stage is used, and an error in the control flow manifests as an incorrect result in the jump address. This emulates the behavior of a real fault that may occur during operation.

For the fault behavior benchmarking, single transient faults incur only a negligible overhead, even in the case of software DMR, which has the most expensive error correction mechanism as it necessitates a complete actor re-execution. This is primarily due to the size of the benchmark dataflow graphs. For instance, the matrix multiplication graph comprises 99 actors, the FFT graph contains 98 actors, and the bitonic sort graph includes 705 actors. In these scenarios, a single actor re-execution introduces only a marginal increase in overall execution time, especially since no additional DMA transfers are required.

The fault injection unit serves the purpose of validating fault tolerance behavior for both Hardware (HW) and Software (SW) redundancy. Faults are deliberately injected during program execution, and their impact is closely examined. In non-redundant execution, a misbehavior in the executed application is observed. However, correct execution is maintained for both SW and HW redundancy, even in the presence of injected errors.

6.5 Summary

This chapter integrates the ALP cores into a multi-core architecture, employing a tile-based design. Each tile in this architecture can be configured to encompass multiple ALP cores. To enhance the flexibility of logical lockstep cores, the RMU is introduced, which facilitates and manages various logical lockstep configurations. Communication between tiles and the memory hierarchy is orchestrated by the NA. Building upon the outlined tile-based architecture, a hybrid NA is presented. This NA offers an interface optimized for both MPI and another tailored for more substantial data movement. The MPI interface is designed for transmitting small control and synchronization data between compute tiles.

The platform undergoes evaluation concerning its WCET behavior and performance. In the performance assessment, hardware and software redundancy as well as a combination is compared. Both hardware DMR and TMR demonstrates advantageous runtime behavior when contrasted with software implementations.

Chapter 7

Adaptive Checkpointing

This chapter provides an in-depth Description of the Adaptive Cache Checkpointing (ACCP), explaining the Cache Checkpointing (CCP) concept and its associated HDL implementation. A comprehensive evaluation of the HDL implementation is presented in the concluding sections. The initial concept and implementations were previously published in [Kem22a], and this chapter serves as an extension, providing additional insights and details beyond the scope of the original publication.

7.1 The Concept

The preceding sections have explored the Adaptive Lockstep (AL) architecture, encompassing both its conceptual framework and practical implementation. This approach involves the close coupling of two or three cores within an associated DMR or TMR cluster. At its core, the AL architecture incorporates an instruction-level SoR, demanding

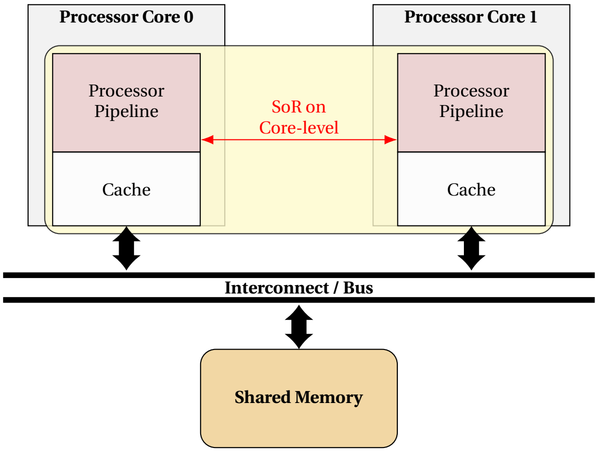


Figure 7.1: SoR of the Adaptive Cache Checkpointing (ACCP) architecture on a shared memory architecture.

a careful comparison of results during each clock cycle. The exchange between cores requires the sharing of results.

In contrast, the ACCP opts for a loosely coupled strategy. Here, not every instruction undergoes examination; rather, the system state is compared after the execution of multiple instructions. The ACCP employs a core-level SoR (refer to Section 2.2.4), committing results to the system state and main memory only after a comprehensive comparison.

7.1.1 The Adaptive Cache Checkpointing Principle

The foundation of the AR concept lies in the reconfiguration of independent *physical cores* into a unified *logical core*. All physical cores within the *logical core* execute the same program, as detailed in Section 4.1.2. The ALP achieves this by introducing a *split mode* for *non-*

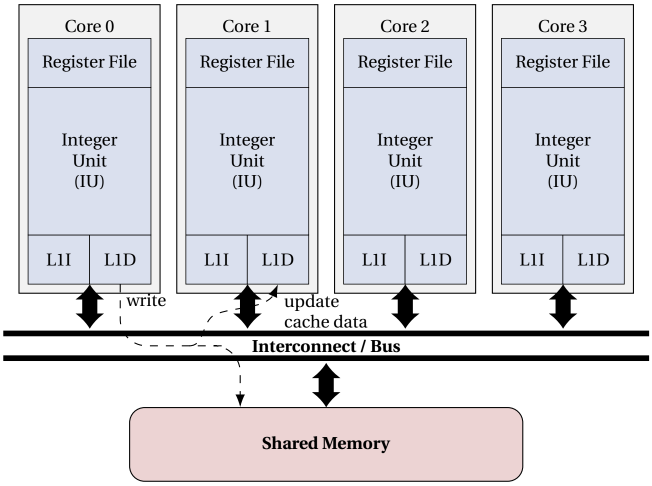


Figure 7.2: Processor and cache configuration of the *performance mode*. Processor cores uses a write-through cache with a bus snooping coherency protocol.

critical functionalities and a *locked mode* for *critical* functionalities. The activation of the *locked mode* gives rise to the creation of the *logical core*. However, the fusion of two or more cores into a single *logical core* does result in a reduction in the overall computational system performance.

The ACCP concept is based on the same fundamental principle, however with a different approach. Instead of orchestrating the simultaneous operation of two or more cores in lockstep, the ACCP concept employs a core-level SoR to construct a *logical core*. Since these cores are not tightly interlocked, the ACCP introduces distinct terminology. Instead of a *split* and a *locked mode*, the ACCP concept applies a *performance* and a *reliability mode*. The *performance mode* operates without a safety mechanism, while the *reliability mode* incorporates a safety mechanism at the core-level SoR.

Within the *performance mode*, a system with cache coherency is employed, illustrated in Figure 7.2. This configuration showcases a system with four processing cores. Cache coherency leads to an update of local cache data whenever the same data location is changed on another core. As illustrated in the figure, when *Core 0* modifies data in shared memory, *Core 1*, with a cached copy of the same data, undergoes an update to its *L1D* cache data due to the cache coherency protocol.

On the other side, the *reliability mode* operates within a system without cache coherency, as shown in Figure 7.3. The CCP employs *write-back* caches without of any cache coherency use to protect data within the SoR. Before data is written from the data caches to the shared memory, a state comparison between the *master* and the *slave core* is executed.

Both the utilization of a write-back write policy and the implementation of non-coherency play important roles in ensuring data isolation. The write-back policy acts as a safeguard, preventing the incorporation of corrupted data into the shared memory. Simultaneously, the non-coherent cache serves as a protective barrier, preventing external faults from directly infiltrating the cache data — where these external sources could include other processing elements, such as *Core 2* or *Core 3*.

The illustrated example shows the scenario presented in Figure 7.2, where one core writes to the shared memory without affecting the cached data of *Core 1*. In contrast to the prior example, the cached data remains unchanged. This data isolation mechanism prevents updates and potential data corruption.

In the context of the write-back cache within the *reliability mode*, each time a cache miss requires the replacement of a cache line, that line may be written back to the shared memory. When this write-back oc-

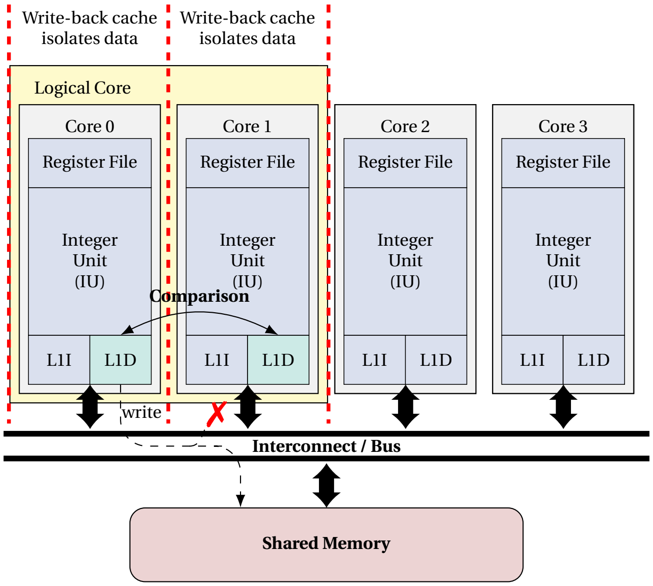


Figure 7.3: Processor and cache configuration of the *reliability mode*. Two processor cores are configured to one logical core. Both cores a write-back cache without coherency.

curs, a comparison of the processor states between the *master* and *slave* takes place. If both states are identical, signifying fault-free data, it is then committed to the system state by being written to the shared memory.

The comparison of system states involves a signature creation by a checksum calculated over all state-changing operations. Both the *master* and *slave* conduct identical checksum calculations. An identical program execution results in the same internal state transition and resulting signature. By monitoring these transitions with a checksum, any differentiation between the two system states can be detected.

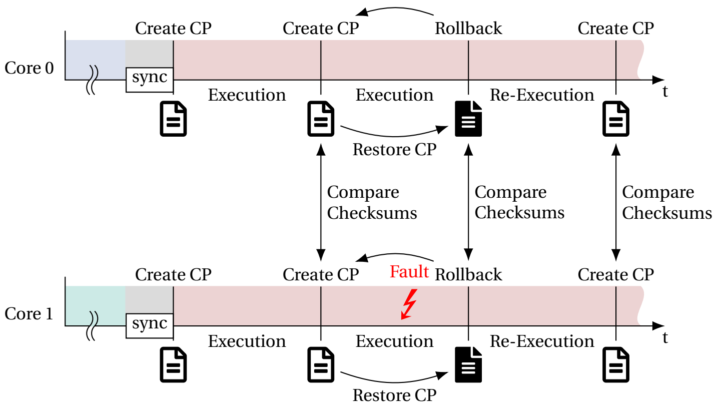


Figure 7.4: Timing diagram showcasing the checkpoint creation and reverting in case of an detected fault.

Figure 7.4 presents a timing diagram illustrating the cache checkpointing concept employing a BER. While the discussion so far has focused on the concepts of data isolation and fault detection, the subsequent analysis delves into the fault-handling aspect. In contrast to the ALP concept, a straightforward re-execution is not viable. As obviously shown in the timing diagram, the compression of checksums spans multiple instructions. Consequently, each time the compression of *master* and *slave* checksums is successful, both cores generate a Checkpoint (CP). This checkpoint serves as the fundamental for rollback in the event of a detected fault. When a fault is identified through different checksums, both master and slave cores restore the last CP. The restored CP makes the re-execution of the flawed program segment possible, as illustrated by the dark file symbol, contrasting with the white file symbol representing created CPs.

One notable limitation of the ALP lies in the shared register file. However, for the ACCP, this limitation is a specific focus.

Hence, the illustrated synchronization phase (*sync*) in Figure 7.4 differs from the synchronization phase of the ALP concept and design. In the ACCP, the synchronization phase is leveraged to transition from the *performance mode* to the *reliability mode*. Throughout this phase, the complete state is transferred from the *master* to the *slave core*, encompassing the internal state of both the IU and the state of the *register file*.

For the state transfer of the control registers of the IU, the same mechanism as previously presented for the ALP (refer to Section 5.3.2) is applicable. The separation of the *register files* aligns with the loosely coupled concept. However, all values from the master *register file* must be shared with the slave *register file*, a task accomplished during the synchronization phase.

Preserving the original slave state is necessary for a later resumption of the original slave program. Similar to the ALP concept, the ACCP saves its current state, encompassing the state of the IU and the *register file*. As illustrated in Figure 7.5, the slave accepts the master state while simultaneously retaining its own state. While the master state is replicated from the master to the slave, the slave migrates its own state (refer to Section 4.2.1). Upon leaving a safety-critical section with no need for safety mechanisms, the migrated state is restored. This saved state is used in resuming the original software on the slave core.

The ACCP employs state migration to uphold the integrity of the original state, a feature not required in the ALP concepts. Regarding the control registers of the IU, the same mechanism used in the ALP can be applied. However, the ALP concepts overlook the replication of the *register file* by sharing a common file between the *master* and *slave core*.

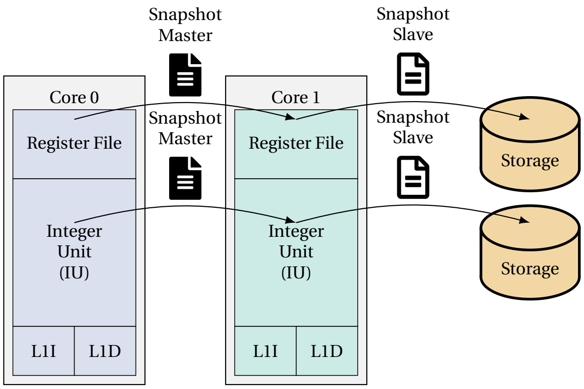


Figure 7.5: State replication of the *master core* (*Core 0*) to the *slave core* (*Core 1*). The slave core migrate its own state to an storage.

7.1.2 The Checkpoint Design

The design of checkpoints offers a degree of flexibility, with various design parameters categorized according to the taxonomy presented in Section 2.2.5. The checkpoint design is characterized by three key parameters:

- Sphere of checkpoint
- Separation of checkpoint and active data
- Checkpoint location.

The initial design parameter under consideration is the checkpoint's scope. The sphere of the checkpoint encompasses not only the checkpoint data but also all data within the lower memory hierarchy.

In the context of cache checkpointing, this extends from the cache data down to the register data. Encompassed within this scope are the memory of the cache itself and all register state memories within the executing processor pipeline. These processor registers encompass the *register file*, as well as the state and control registers of the processor.

For the LEON3, the processor registers, as defined by the SPARC v8 ISA, include:

- Multiply/Divide Register (Y)
- Program Counter (PC)
- Next Program Counter (NPC)
- Processor State Register (PSR)
- Window Invalid Mask (WIM).

The cache, situated as the lowest level of the memory hierarchy within the processor core, serves as the optimal location for implementing the targeted core-level SoR. Lower-ranked memory locations external to the core are considered unsuitable. To comprehensively cover all data within the memory hierarchy, the checkpoint must span from the cache data down to the register data. This inclusivity ensures that, in the event of a fault and subsequent error, a rollback to this system state is possible. Following the rollback, the system must be restored to its previous state, equivalent to the CP. The flawless IU state and *register file* are recovered from the checkpoint.

Moving to the second design parameter, the storage location of the checkpoint is considered. The checkpoint data can either be stored on the same memory hierarchy (as a dual-level checkpoint) or on a lower memory hierarchy. Leveled checkpoints involve an offloading process that consumes more time compared to the creation of a dual-level checkpoint. During offloading, data is transferred between memory hierarchies, with the checkpoint data moved and stored on a lower memory hierarchy level characterized by higher access times.

Leveled checkpoints, encompassing the cache state and register memories, can be stored in the shared system main memory. In contrast, dual-level checkpoints sidestep data transfer to a lower memory hierarchy by keeping the checkpoint at the same level as the active data. While this eliminates the need for data movement to a lower memory hierarchy, it requiers additional hardware for checkpoint storage.

The third design parameter consider the separation of the checkpoint and active data. This separation can be complete or partial, with the latter being more practical. Partial separation typically falls into three categories: buffering, logging, and renaming. Buffering involves buffering all results before committing to a new checkpoint, while logging updates the value and saves the original checkpoint value in a different memory location. Renaming, the chosen method for checkpoint design, modifies by redirecting, storing the updated value in a different location and updating the data mapping. The renaming method supports fast checkpoint creation and recovery times, with expected minimal hardware costs.

Figure 7.6 provides a comprehensive overview of the checkpoint design within the ACCP concept, utilizing the taxonomy presented in Section 2.2.5. In summary, the checkpoint's sphere extends up to the cache level, tracking data changes and maintaining both checkpoint data and active data at the same level within the memory hierarchy.

7.2 The Checkpoint Capable Register File

As explored in the checkpoint design discussion, the sphere of the checkpoint must encompass not only the cache data but also the register file. Consequently, the *register file* must facilitate the creation and management of CPs. This section introduces a *checkpoint-capable register file* designed to create checkpoints and perform rollbacks to the last created checkpoint.

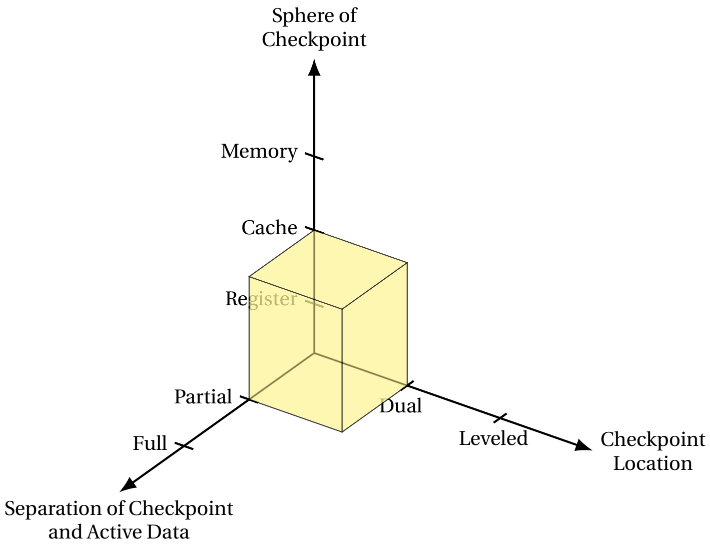


Figure 7.6: Visualization of the checkpoint design. The ACCP uses a partial cache checkpoint, which is dual leveled.

7.2.1 Checkpoint Management

The register file employ the checkpoint design principles previously outlined, concerning both checkpoint location and the separation of the checkpoint. Both the active data and the register file's checkpoint reside on the same memory technology. To distinguish the active data from the checkpoint, a renaming method is employed, defining two logical domains: one for the *modified* and *unsafe* data, and another for the *checkpoint* and *safe data.*

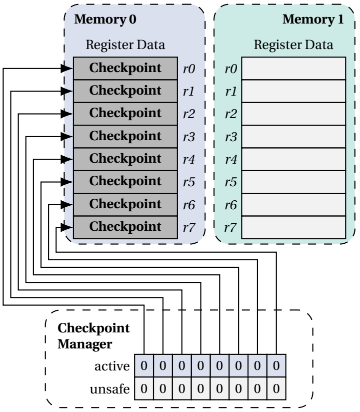
The *modified data* is initially considered *unsafe* as it has not undergone fault verification. Once the system state is checked, the *modified data* transitions to a *safe* state, subsequently becoming part of the new *checkpoint data*. All data included in the *checkpoint* must be fault-free and is, by definition, *safe*.

These logical domains represent potentially flawed *unsafe data* and flawless *safe data*. Through the renaming method, these domains are mapped to their corresponding memory locations. In a worst-case scenario between two checkpoints, the entire register file is modified, resulting in each register having both a *checkpoint* value and a *modified* and *unsafe active data* value. Consequently, the memory content of a register file must be duplicated, with the two domains distributed across both memories.

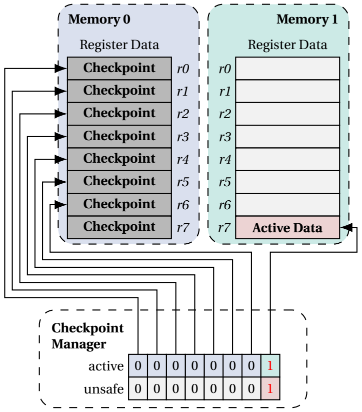
The access to the logical domain and its associated memory is controlled by two status registers: *active* and *unsafe*. Each access to the *unsafe* and *safe* domains is controlled by these registers. The *active* register indicates the concrete memory for read accesses, pointing where the current active data is located. This data can either be from a previously created CP or *modified data*. If the data has remained unchanged since the last checkpoint, the *active* register points to the current checkpoint. Otherwise, it directs to the *modified data*.

The *unsafe* register monitors write accesses and the ensuing data modifications. Whenever a register value is written, the *unsafe* register is set. The utilization of *active* and *unsafe* registers is illustrated in Figure 7.7, which shows four scenarios explaining the connection between the *active* and *unsafe* registers.

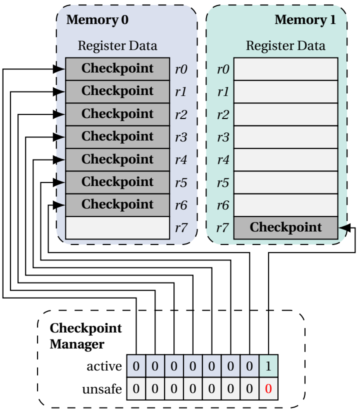
Figure 7.7a illustrates the initial state of the *register file*. All displayed data belongs to the CP and is located in *memory 0*. The *unsafe* register values for all registers are set to *'0'*, indicating that none of the data has been modified since the last checkpoint, and each element is *safe*. Each element of the *active* register points to *memory 0* by being set to *'0'*, ensuring that every read access is executed from the *checkpoint data*.



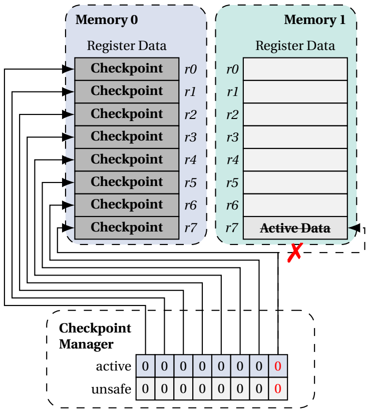
(a) Initial state of the *checkpoint-capable register file*. All register are *safe* and part of the *checkpoint*



(b) Write to *r7*



(c) Checkpoint creation after a write to *r7*



(d) Rollback after a write to *r7*

Figure 7.7: Illustration of different operations on the *checkpoint-capable register file*. Changes are highlighted in red.

From Figure 7.7a, a write access to the register *r7* results in a state as shown in Figure 7.7b. To prevent the modified data from overwriting and corrupting the checkpoint, the data is redirected and written to *memory 1*. This leads to an update of the corresponding values in the *active* and *unsafe* registers. After the write, the *active* register needs to point to *memory 0*, and the *unsafe* register needs to indicate data modification.

The combination of both register states assigns the correct locations for the checkpoint and the *modified data*. The set bit in the *unsafe* register ensures that all future write accesses are directed to the currently set active register. In this example, all future write accesses to the *r7* register are mapped to *memory 1*. Only after a checkpoint creation does the write location change.

At the start of the write access, the *unsafe* bit was unset and set to *'0'*. This redirects the write access and changes the *active* register. Instead of writing to the previous memory location with the checkpoint data, the *active* bit is inverted. For this example, the previously unset register value results in a set *'1'*. After the write access, both corresponding register values are *'1'*. Subsequently, all read accesses are performed from the *unsafe* data selected by the *active* register value.

As previously described, future writes are not further redirected as long as the *unsafe* bit is set. The *unsafe* bit is cleared either by a checkpoint reaction or a rollback to the last checkpoint. Both cases are considered next.

The flowchart detailing the recovery logic is presented in Figure 7.8. The hardware simultaneously checks each data element for the unsafe bit. In the event of unsafe data, corrective measures are initiated. Consequently, for affected data elements, a rollback to the last checkpoint data is executed. This rollback mechanism entails toggling the active register value.

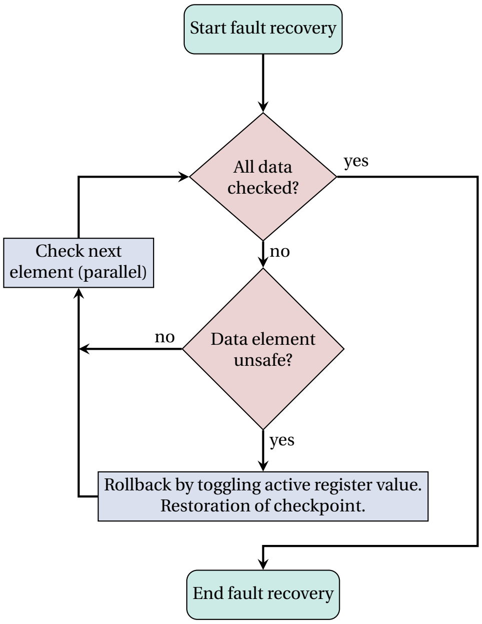

Start fault recovery
All data checked?
yes
no
Check next element (parallel)
no
Data element unsafe?
yes
Rollback by toggling active register value. Restoration of checkpoint.
End fault recovery

Figure 7.8: Flowchart showing the rollback process.

Figure 7.7c illustrates the scenario of a successfully established new checkpoint after a write access to *r7*. Before the checkpoint creation, the *unsafe* data was located in *memory 1*, with set *active* and *unsafe* register values. Creating a checkpoint involves clearing the *unsafe* register, indicating that all data within the register file is now *safe*. Consequently, the previously *unsafe* register value in *r7* is now safe and part of the newly created checkpoint. The former checkpoint data at *r7* in *memory 0* is no longer valid, allowing the location to be overwritten with a future write access.

Figure 7.7d illustrates the second case where the *unsafe* bit is cleared, applicable whenever a rollback to the last valid checkpoint is necessary. The checkpoint exclusively comprises *safe* entries of register

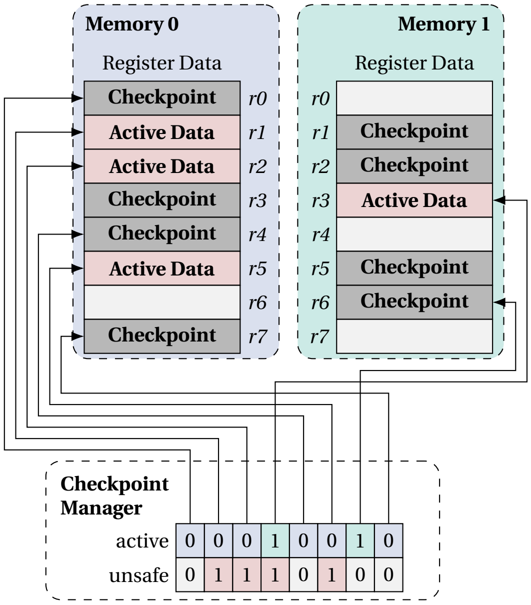


Figure 7.9: Exemplary state of a fragmented *checkpoint-capable register file*

data, requiring the complete clearing of the *unsafe* register. The prior changes and modifications to the register data must be undone by restoring the original access mapping. Hence, the *active* state at the time of checkpoint creation is recovered. The recovery of the original *active* register is relieved by the *unsafe* register. The changes tracked by the *unsafe* register need to be undone, achieved by re-inverting the *active* register values. All *active* register values where the *unsafe* bit is set are re-inverted, thus mapping the access direction back to the checkpoint data.

To summarize, the current state of the *active* register depends on both the write history of each register and the checkpoint creation history. The *unsafe* register state relies only on the write accesses, as it logs modifications since the last checkpoint creation. Both register states are essential for selecting the correct write or read domain and its associated location. Figure 7.9 illustrates an exemplary state of a *checkpoint-capable register file*, showcasing the fragmentation of the different logical *safe* and *unsafe* domains across two memories.

When the processor is in *performance mode*, only one logical domain is utilized. Once the processor core switches to *reliable mode*, both logical domains come into play. The performance configuration exclusively employs the *unsafe* domain, where all data is considered modified. In this configuration, rolling back to the last CP is not intended, as there is no valid CP available. Conversely, in *reliable mode*, the register file utilizes both domains, with the *safe* domain covering the CP and the *unsafe* domain handling the modified data.

7.2.2 Hardware Architecture

The control logic controlling the *active* and *unsafe* registers is shown in Figure 7.10 and designed for a minimal logic overhead. This logic implements the previously described behavior for the management and control of the checkpoint and its recovery. The logic is simplified, excluding the checkpoint creation, as this simply involves clearing the entire *unsafe* register.

The relevant signals include the indication of a data write, labeled as the *write access* signal, and the signal for checkpoint recovery, denoted as the *rollback* signal.

When the *write access* signal rises, the logic sets the bit of the *unsafe* register and inverts the *active* register. Conversely, the *rollback* signal clears the *unsafe* signal and toggles the *active* register back.

The interface of the presented register file is compatible with the original LEON3 register file. It defines two read ports, *rs1* and *rs2*, and the

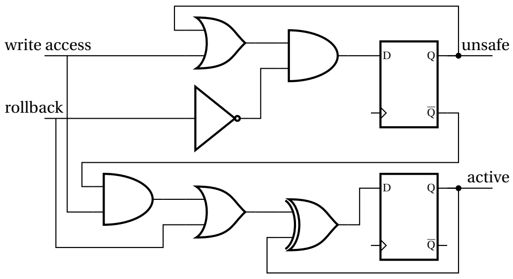


Figure 7.10: Checkpoint control logic (refer to [Kem22a])

write port, *rd*. Additionally, the interface is extended by two inputs for checkpoint creation or to initiate a rollback and recovery of the last checkpoint. The *createCP* signal creates the checkpoint by clearing the *unsafe* bits, and the *rollback* signal initiates the recovery of the last created checkpoint.

The presented register file behaves identically to the originally employed *register file*, as long as no rollback is performed. The creation of a checkpoint does not affect the data management nor externally visible behavior. Data access before and after the creation of a checkpoint behaves for the IU in an identical manner. The behavior is indistinguishable from the original register file. As the concept and design behave at the interface level identically to the original register file, no timing overhead needs to be introduced. This applies to both checkpoint creation and rollback. The previously presented concept introduces zero time overhead for both actions, CP creation, and recovery.

The architecture of the *checkpoint-capable register file* is illustrated in Figure 7.11. The register file interface includes three register ports: *rs1*, *rs2*, and *rd*, along with control signals responsible for managing the checkpoint, namely *createCP* and *rollback*. The illustrated register file consists of a *selection unit* and two *triple-ported Random-Access Memory (RAM)* units, which are duplications of the memory in the origi-

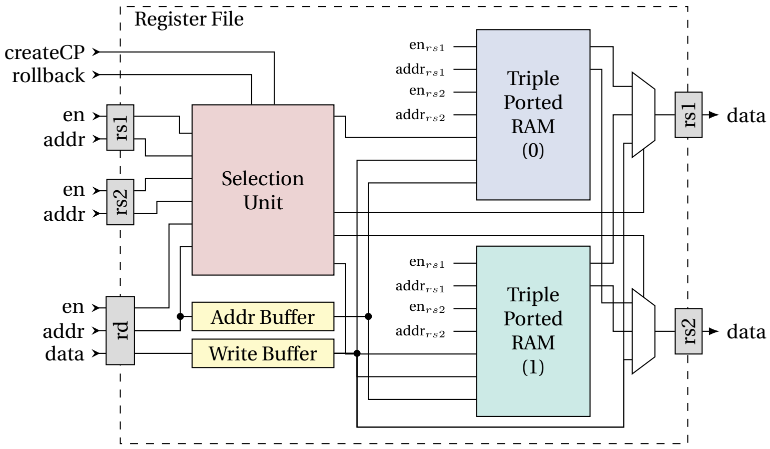


Figure 7.11: Internal structure of the *checkpoint-capable register file.*

nal register file. Additionally, the *checkpoint-capable register file* comprises two buffers, *addr* and *data write,* required due to the pipeline architecture of the *selection unit,* storing both the *write address* and the *data to write.*

The *selection unit* monitors and controls the selection of the appropriate memory for register access. To enable checkpoint functionality, the *selection unit* employs the non-invasive concept discussed earlier, selecting one of the two SRAM memory locations. This selection process introduces no observable timing overhead to the processor pipeline.

The *selection unit* controls both the output selection of the memories (*rs1* and *rs2*) and the target selection of the write access (*rd*). The read accesses of *rs1* and *rs2* are forwarded to both memories, with each memory providing its stored value on the associated output port. The register value is not immediately provided but is available in the next clock cycle. Simultaneously, the *selection unit* determines the cor-

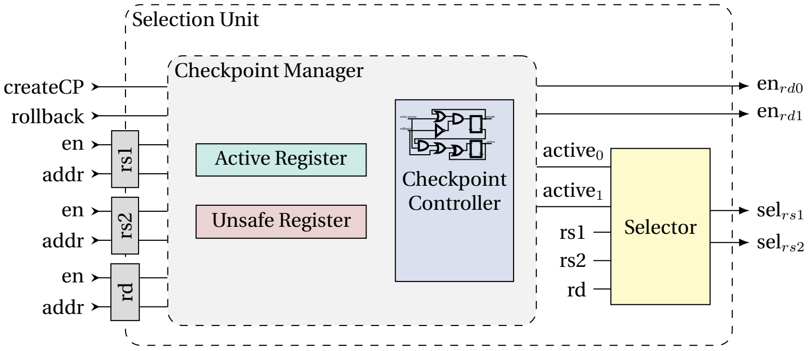


Figure 7.12: Internal structure of the *selection unit*

rect location of the register value using the previously presented algorithm. Based on the selection result, the output multiplexers of the *checkpoint-capable register file* are configured.

The architecture of the *selection unit* is presented in Figure 7.12. It consists of two main components: the *checkpoint manager*, responsible for managing the checkpoint location, and the *selector*, which chooses the appropriate memory for register access.

The *checkpoint manager* implements the previously presented register file checkpoint concept and its associated logic. It utilizes the *unsafe* register to track all registers that have changed since the last checkpoint, and the *active* register points to the currently active memory location. Each read access depends just on the value stored in the *active* register since it indicates the memory location of the current active data. Both the *unsafe* and *active* registers determine the target location for write access. By coordinating the *unsafe* and *active* registers, the *selection unit* ensures that the checkpoint data is not overwritten. Whenever the *createCP* signal is raised, a checkpoint is created, and the changes tracked by the *unsafe* register are cleared. When the *rollback* signal requests a rollback to the last checkpoint, the changes tracked by the *unsafe* register are undone.

The *selector* unit configures the output multiplexers. The values of sel_{rs1} and sel_{rs2} determine the correct data output for the register file ports rs_1 and rs_2. When the sel_{rs1} value is *'0'*, the register value from *memory 0* is selected. A value of *'1'* selects *memory 1*, and the write buffer is selected with a value of *'2'*.

From the perspective of the pipeline, it appears as a regular register file without explicit support for checkpoints. The timing behavior of checkpoint creation and management is completely abstracted and hidden from the pipeline. One of the memories stores the register value of the checkpoint, while the other is used to store modifications made since the last checkpoint. The selection between these memories depends on the write history and is managed by the *selection unit*.

Figure 7.13 illustrates the timing behavior of the *checkpoint-capable register file*. The timing diagram is simplified to one read port, rs_1, and one write port, *rd*. The shown behavior of the rs_1 port is also representative for rs_2. Both ports behave identically. The two read ports are completely independent of each other and do not interfere with each other. The *rs* port consists of the signals rs_{ADDR} for register selection and rs_{EN} signal to enable a read access. The rs_{DATA} signal returns the value of the selected register one clock cycle later. The *rd* port consists of the signals rd_{ADDR} indicating the target register, rd_{EN} to signal the valid write access, and $rs1_{DATA}$ containing the new register value.

The stored values of the *write* and *addr buffers* are illustrated with the signals $Buffer_{WRITE}$ and $Buffer_{ADDR}$. The signals en_{rd0} and en_{rd1} enable writing to *memory 0* and *memory 1*, respectively, using the current values of the *write buffer* at the address specified by the *addr buffer*. When the en_{rd} signal is raised (*'1'*), a new register value is stored in the associated memory.

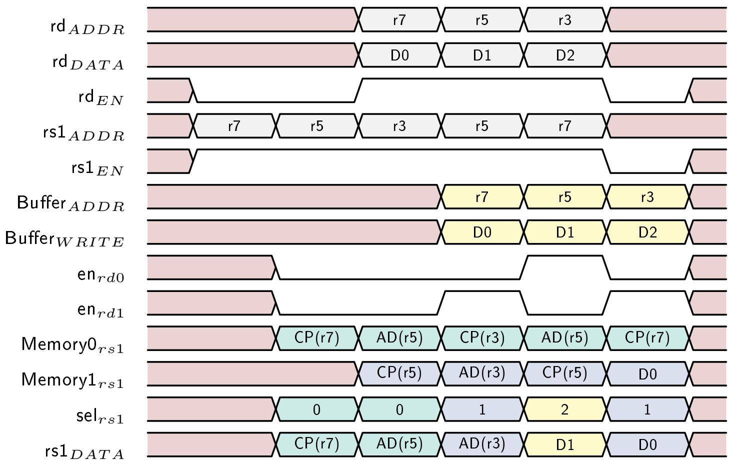


Figure 7.13: Timing diagram if the register file. Multiple read and write operations are shown.

The selection signals *sel*$_{rs1}$ determine the output data of the *rs1* port. The *rs1*$_{DATA}$ value is selected from one of three sources. The first source is *Memory0*$_{rs1}$, the output of *memory 0*, selected when *sel*$_{rs1}$ is *'0'*. The second source is *Memory1*$_{rs1}$, chosen when *sel*$_{rs1}$ is *'1'*. In this case, the value of *Memory1*$_{rs1}$ is used as *rs1*$_{DATA}$. The final source of the *rs1*$_{DATA}$ value is the stored value of *Buffer*$_{WRITE}$. This value is used as the output whenever *sel*$_{rs1}$ is *'2'*.

The timing diagram is based on the *register file state* as shown in Figure 7.9, which represents the initial state for the timing diagram. All write accesses considered in the timing diagram affect the *register file state*.

1. A read from register *r7* is performed. The requested register value is located in *memory 0* and is a *Checkpoint (CP)* entry. The second request reads the *r5* value from the *active data* located in *memory 0*, where the *CP* is placed in *memory 1*.

2. The third access reads and writes data in parallel. The value read from *r3* is located in *memory 1*. Therefore, the output selection sel_{rs1} is *'1'*, choosing $Memory1_{rs1}$ as the value for $rs1_{DATA}$. The write access request writes the value *D0* to register *r7*. As seen before, the register *r7* only consists of a *Checkpoint (CP)* entry located in *memory 0*. Therefore, the *D0* value is written to *memory 1* by raising en_{rd1}. The internal state is updated as seen in Figure 7.7b.

3. The next register file access demonstrates the simultaneous read and write access of the same register. The read value needs to be identical to the written register value. The *selection unit* requires one clock cycle to calculate the target memory. The values stored in $Buffer_{WRITE}$ and $Buffer_{ADDR}$ are used as input for the target memory the following clock cycle. Therefore, when the read and write ports are identical, the *selection unit* chooses the $Buffer_{WRITE}$ as the output value for the $rs1_{DATA}$ signal. When sel_{rs1} is *'2'*, the value stored in $Buffer_{WRITE}$ is forwarded to the $rs1_{DATA}$ port. In this example, the *D1* value from the write port is stored in $Buffer_{WRITE}$, which is forwarded to the $rs1_{DATA}$ port. The stored *D1* value in $Buffer_{WRITE}$ overwrites the *Active Data (AD)* in *memory 0* by raising en_{rd0}. After the write access to *memory 0*, the *AD* is updated to *D1*.

4. The final register access simultaneously reads from the *r7* register and modifies the data stored in the *r3* register. The *r3* register has *AD* located in *memory 1*. This *AD* is updated by raising the en_{rd0} signal. The previous read access to the *r7* register resulted in providing the *CP* from *memory 0*. In the meantime, the *r7* register was modified and updated with a write access. The current *AD* of the register is located in *memory 1*. Therefore, the *selection unit* configures the $rs1_{DATA}$ value of $Memory1_{rs1}$ by setting

sel_{rs1} to *'1'*. The previously written and, in the meantime, *AD* value *D1* is provided to the processor pipeline.

7.3 Adaptive Cache

This section describes and discusses the adaptive cache and its implementation, forming the fundamental for enabling the ACCP concept. The adaptive cache requires a change in the write policy, transitioning from *write-through* to *write-back*. Additionally, the discussion comprises the specifics of where and how a checkpoint is positioned and separated from the active data.

7.3.1 Adaptive Write Police

The fundamental of the adaptive cache lies in its configurable write policy. The LEON3 employs a cache coherent write-through design as the foundation of its basic cache structure, where each store instruction directly updates the main memory.

Bus snooping is utilized to monitor the shared bus, ensuring that whenever a cached word is written to the main memory, the cache updates the corresponding local data. The use of the *write-through* write policy is essential for enabling cache coherence through bus snooping. Without a *write-through* cache, more intricate cache coherence protocols such as MSI or MESI would be required, introducing additional complexity to the cache system [4].

Consequently, the cache employs two distinct policies. In the *performance mode,* where the processor core and the cache subsystem operate, cache coherence is achieved through bus snooping, and a *write-through* write policy is employed. In the *reliable mode,* neither bus snooping nor a *write-through* write policy is suitable. Therefore, the cache utilizes a non-coherent *write-back* write policy in this mode.

The *write-back* policy keeps possible error-prone data with in the shaper of the core, preventing it from leaving and potentially corrupting the system state. This policy withholds the data until its correctness is confirmed.

The adaptive cache checkpointing concept presented here is based on the L1 cache of each LEON3 processor core. It employs a *write-through* policy for the *performance mode* and a *write-back* strategy for the *reliable mode*. These strategies differ in cache line allocation and the timing of data being written to the lower memory hierarchy.

Under the *write-through* policy, data is directly written to the lower memory hierarchy, and the cached data is simultaneously updated. Vice versa, the *write-back* policy only writes modified data to the lower memory hierarchy when the cache line is replaced or flushed. Therefore, a *dirty bit* is required to indicate cache lines that must be written back to the system memory. These *dirty bits* are exclusive to the *write-back policy*, signaling whether a cache line needs to be written back before replacement. The *dirty bit* is set when the processor pipeline writes data to the cache line, indicating a difference between the data in the cache line and the system memory. This data is then written back to the system memory before replacement.

A cache line is replaced when the processor encounters a cache miss while reading or writing to a non-present cache line. In this scenario, the cache has to write back the dirty lines before storing the data from the system memory in the cache line.

However, in the *reliable mode* with the *write-back* policy, a *write-allocate* is required for every single write access that creates a cache miss. Before writing data to the cache, the required line is fetched from the lower memory hierarchy, ensuring the entire cache line is valid and can be completely written back if needed.

The write-back cache introduces two additional states, *write-back* and *write-allocate*, to the state machine of the write-through cache. The *write-back* state is used to write back a cache line with an enabled *dirty bit*, indicating that the content of the line has been manipulated. A write-back is initiated by a read miss before a *write-allocate* of a dirty line is performed, or when the write-back cache is flushed.

Before replacing a cache line, the *dirty bit* of the cache line is checked. If the *dirty bit* is not set, the data can be directly written to the cache line. Otherwise, the data of the cache line is written to the memory before fetching the new data. Every time there is a cache miss for a dirty line, the cache has to write back this line. In the case of a cache flush, all lines are invalidated, and the dirty lines are written back to the lower memory hierarchy.

Similar to the *write-through* policy, all cache lines are invalidated during a flush. For the *write-back* cache, it is necessary to write back all dirty lines to the lower memory hierarchy. To invalidate the complete cache, the cache controller iterates through all lines, checking the *dirty bits*. If the *dirty bit* is set, the line is written to the lower memory hierarchy.

When the cache switches from a *write-back* to a *write-through* policy, a cache flush is performed to ensure data integrity. This transition prevents data corruption or loss, as the *write-back* cache lacks hardware-based coherency support. In the incoherent *write-back* mode, cache lines are incoherent with the system memory, and cache data differs from system data. Therefore, all dirty cache lines must be written back to update the system memory. The *write-through* cache guarantees that the cache data and the system data are identical, as data is always

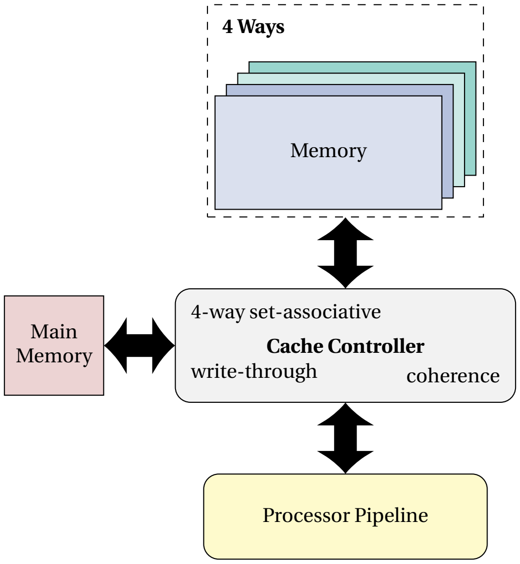


Figure 7.14: Cache configuration of the performance mode (refer to [Kem22a])

written to the lower memory hierarchy. Hence, the shift from a *write-through* to a *write-back* policy can seamlessly occur.

7.3.2 Checkpoint Location

The ACCP employs a write-through cache with cache coherency in its *performance mode*. The configuration illustrated in Figure 7.14 shows a performance setup with four ways. The cache controller orchestrates the access to the individual cache ways and sets. In this *performance mode*, the entire memory of the cache system is fully utilized.

When transitioning to the *reliable mode*, the cache system initially switches its write policy from write-through to write-back. As discussed earlier, the write-back cache policy does not depend on nor uses a cache coherence protocol.

For cache checkpointing, a record of the cache state needs to be created for a subsequently required BER. As previously outlined, there are two potential methods for storing this data. One approach involves storing the checkpoint on a lower memory hierarchy as a leveled checkpoint, while the alternative is to store it as a dual-level checkpoint on the same memory hierarchy.

Opting for a dual-level checkpointing scheme proves advantageous. This choice facilitates both a rapid checkpoint creation and recovery process, reducing memory access to shared resources. Consequently, the checkpoint management process becomes time-predictable and independent of other system components. Nonetheless, the use of the dual checkpoint scheme requires additional storage for checkpoint data.

The same principle employed for the *checkpoint-capable register file* can be applied to the cache checkpoint scheme. However, duplicating the memory incurs considerable costs. The register file only needs to provide space for the processor's internal registers. For example, in the case of the SPARC v8 ISA, there are between 40 registers for two register windows and 520 registers for 32 register windows. The standard configuration includes eight register windows, resulting in 136 registers. Each register has a size of four bytes per word and entry, totaling a register size of 544 bytes [18; 19].

The LEON3 cache sub-system of the L1-Cache can be configured with 1 to 4 ways, where each way has a size ranging from 1 KiB to 256 KiB, resulting in a cache size of at least 1KiB up to 1 MiB [18]. For instance, an L1 data cache with two ways and a size of 4 KiB has a total size of 8 KiB, which is more than 15 times the size of the register file. Similarly, a L1 data cache with four ways and a size of 4 KiB per way has a total size of 16 KiB, exceeding the register file size by more than 30 times.

Fully replicating a cache memory incurs significant hardware expenses that remain unused during the *performance mode* and are only fully utilized during the *reliable mode*. In contrast, the hardware costs associated with replicating the register file are justifiable and manageable. Compared to the costs incurred by duplicating a cache memory, these hardware costs are negligible.

Cache checkpointing, however, has the potential to change the memory appearance. Unlike the register file, where the number of registers is predefined, and values are stored within the register file, the cache buffers data from the lower memory hierarchy. Consequently, the cache memory can be downsized without changing the functional behavior of the processor or the executed program.

In the *reliable mode*, the memory allocated in the *performance mode* is halved and divided. One half of the cache memory is dedicated to checkpoint storage, while the remaining half is reserved for the *active data*. Figure 7.15 illustrates the cache configuration post-reconfiguration. In the reliable mode, the original four-way associative cache undergoes a reconfiguration, transforming into a two-way associative cache.

The cache ways are partitioned into two distinct *memory domains*. In the example of the four-way associative cache, two ways are situated in *memory domain 0*, and the remaining two ways are allocated to *memory domain 1*. The mapping of the *safe* and *unsafe* domains corresponds to one of the two virtual *memory domains*.

In the reliable mode, the memory dedicated to storing the checkpoint is fully utilized, just as in the *performance mode*. The entire cache size is employed during the *performance mode*, resulting in improved cache and computational performance. The larger cache size in *per-*

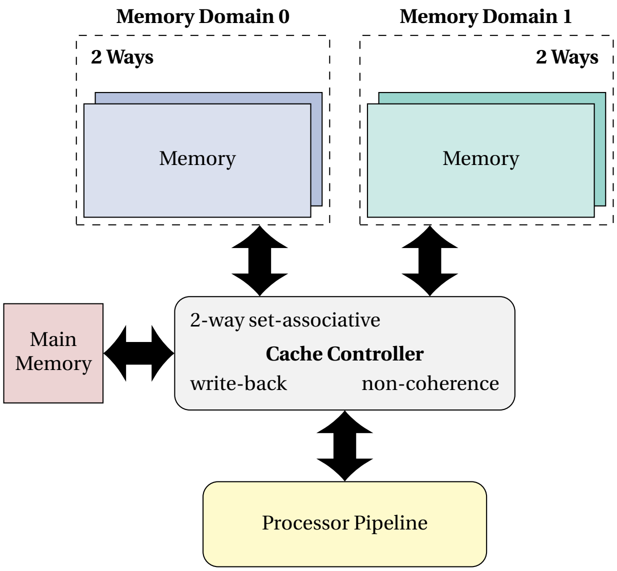


Figure 7.15: Cache configuration of the reliable mode (refer to [Kem22a])

formance mode minimizes cache misses, thereby enhancing overall processor computational efficiency.

While the use of only half of the cache in *reliable mode* may degrade expected computational performance, the implemented safety mechanism significantly enhances dependability. This duality in performance characteristics contributes to the naming scheme of *performance mode* and *reliable mode*.

7.3.3 Checkpoint Management

Checkpoints are exclusively generated in the *reliable mode* when the cache operates with a *write-back* write policy, and half of the cache is allocated for storing checkpoint data. Employing a dual-level scheme, the checkpoint remains within the cache, facilitating rapid checkpoint creation and rollback mechanisms. This strategy prevents the offloading of checkpoints to a lower memory hierarchy, mitigating latency introduced by reading and writing each checkpoint entry. Additionally, accessing the lower memory level through a shared bus introduces latency and interferes with other participants. Such interference can impact the offloading or restoring process of the cache by introducing unpredictable delays.

To address this, the *unsafe* data is segregated from the checkpoints in different memory spaces. The cache checkpointing mechanism, similar to the *checkpoint-capable register file,* is divided into *safe* and *unsafe* domains. Here, the checkpoint is located in the *safe* domain, while all modified data is assigned to the *unsafe* domain. Instead of relying on two dedicated physical memories, the cache checkpointing scheme utilizes the two virtual *memory domains.*

In Figure 7.16, an illustrative cache showcases both *safe* and *unsafe* data. The checkpoint management aligns with the previously discussed approach for the *checkpoint-capable register file.* The *cache controller* employs a logic similar to that used for the *checkpoint-capable register file.* The utilization of the *active* and *unsafe* controls supervises access to the respective *memory domain.*

The checkpointing scheme implemented in the cache controller differs from the register file's checkpoint manager, primarily because not all data in the cache is inherently valid. Unlike each register entry in the register file, which contains valid data, the current available cache data is only considered valid upon a cache hit. A cache hit happens when the tag address matches the data address, and the cache line is marked as valid. Each cache line possesses its own valid bit, indicating the validity of that particular cache line. It is noteworthy that the

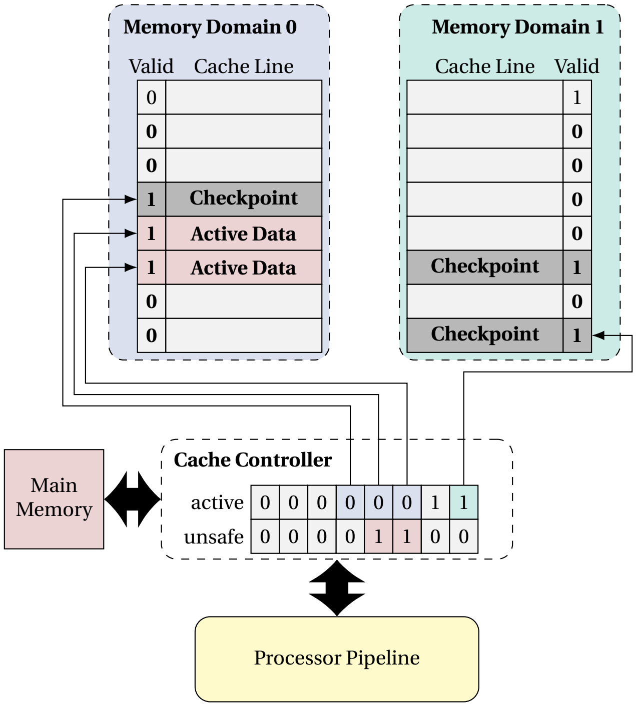


Figure 7.16: Cache controller managing the fragmented active and checkpoint data over two memory domains. (refer to [Kem22a])

cache may also contain cache lines that are neither associated with the *safe* nor the *unsafe* domains. The example in Figure 7.16 illustrates the four possible states of a cache line.

The first state represents an empty cache line in both memory domains. The second state mirrors the register checkpointing scenario, where one memory domain houses a checkpoint, and the other domain is available for storing *unsafe* data. The third state is similar to the *checkpoint-capable register file* discussion, where both entries in *memory domain 0* and *memory domain 1* contain data. In the example, cache line 5 holds *unsafe* and active data in *memory domain 0*, while *memory domain 1* contains the checkpoint data for potential later rollback and checkpoint recovery. The last and fourth state is distinct, as the *unsafe active data* of cache line 4 has no corresponding checkpoint in the opposite memory domain.

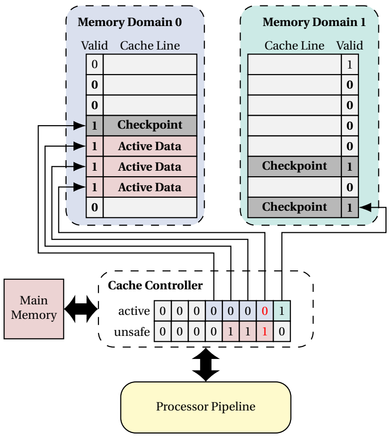
Two cache accesses do not change the *active* and *unsafe* registers. For both types of access, a cache hit is essential. Reading from a *checkpoint* or writing to *active data* does not modify either of the registers. In a read access, the present data from the checkpoint is used to forward data to the processor pipeline. In a write access, *unsafe* data is modified, leaving both the *active* and *unsafe* registers unchanged.

Using the example as a basis to consider four different cache accesses that change the *active* and *unsafe* registers, starting from the initial state shown in Figure 7.16.

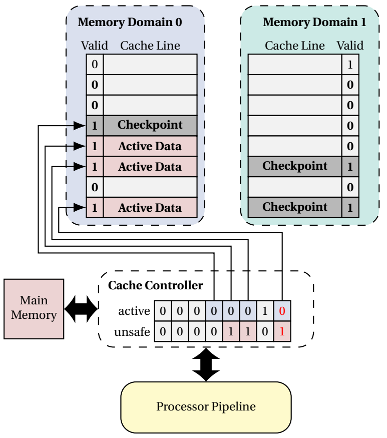
Scenario 1: Read Miss without Checkpoint (refer to Figure 7.17a)

In the case of a read miss, occurring when the processor pipeline attempts to read data not present in the cache, the *active register* points to the cache line of the active memory domain. For instance, when the processor aims to read data associated with cache line 6, and the *active register* points to *memory domain 1* where no valid cache line exists, a read miss occurs. This leads to fetching data from the lower memory hierarchy, comprising entirely of *safe* data. Although this ensures fault-tolerant checkpoint recoveries with low appearance, a write hit on a cache line with a checkpoint introduces latency, degrading computational performance. Consequently, the *active register* is updated, and the *unsafe register* is set to *'1'*, marking the fetched cache line as unsafe for performance reasons.

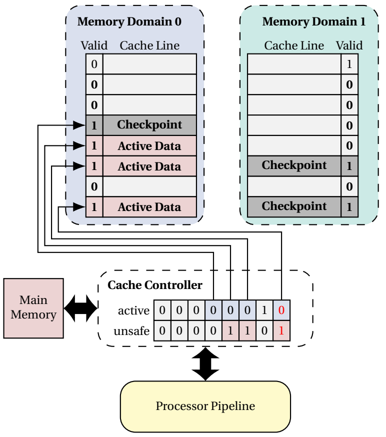
Scenario 2: Read Miss with Checkpoint (refer to Figure 7.17b)



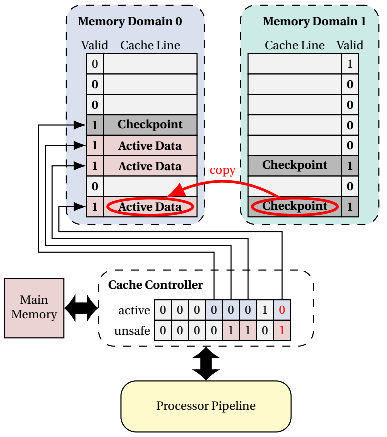
(a) Read miss without checkpoint



(b) Read miss with checkpoint



(c) Write miss with checkpoint



(d) Write hit with checkpoint

Figure 7.17: Comparison of four different cache access scenarios. Changes to the cache controller state are highlighted in red

Similar to the first case, this scenario involves a read miss, but with a cache line in *memory domain 1* containing a checkpoint. In this case, the *dirty bit* of the checkpoint data is checked, and if set, the cache line with the checkpoint must be written back to the main memory.

Scenario 3: Write Miss (refer to Figure 7.17c)

This scenario covers a write miss, where the *active register* points to a cache set with checkpoint data at *memory domain 1*. Similar to the read miss cases, the *dirty bit* of the checkpoint data needs to be checked first. If necessary, the data must be written back to the main memory, and the *dirty bit* is cleared. The write-allocate fetches the cache line from memory before the store instruction completes, marking the cache line as unsafe for potential flaws in the stored values.

Scenario 4: Write Hit (refer to Figure 7.17d)

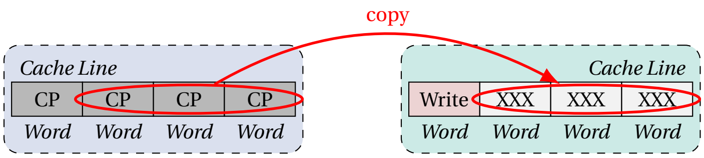
In a write hit scenario, where the processor pipeline intends to store a data word on cache line 7, the *active register* points to a checkpoint in *memory domain 1*. To preserve the checkpoint, the write is redirected to *memory domain 0*, flipping the *active register* value and setting the *unsafe* bit. The *data checkpoint* of the checkpoint does not need to be written back, ensuring the data remains valid after the write.

Data Consistency of Unmodified Data in Checkpoints

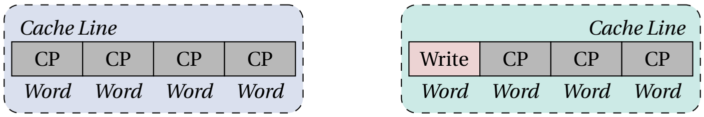
To ensure data consistency of unmodified data in a checkpoint, the data is copied from the checkpoint to the *unsafe* domain. This step is essential for maintaining the integrity of checkpoint data. The need for this copying process is further explored by considering the difference between a write access of a register within the *checkpoint-capable register file* and a cache line. Unlike a register in the register file, which consists of a single value (as illustrated in Figure 7.18a), a write access of a cache line updates partial data, requiring careful management to ensure data consistency.

Register
Checkpoint
Register
Write Data

(a) Write access of a register within the register file



(b) Write access of a cache line with an available checkpoint (before data is copied)



(c) Write access of a cache line with an available checkpoint (after data is copied)

Figure 7.18: Compression of different write accesses.

Unlike a register in the *checkpoint-capable register file*, a cache line poses a distinct challenge. A cache line comprises multiple words, typically four or eight. In the example shown in Figure 7.18b, four words are used. During a data store operation, only one word is updated, and in a double store, only two consecutive data words are modified. Consequently, at least two words within the cache line remain unchanged. The state of these unmodified words is unknown and depends on the old memory values.

To ensure data consistency for the unmodified data in the cache line, a copy mechanism becomes necessary. Therefore, during a write access to a cache line where a checkpoint is present, the process involves copying the checkpoint line to the new active data line and afterwards updating the data words in the active line. This strategy minimizes data transfers, as only the unchanged data is copied from the checkpoint. After this data transfer, the cache line assumes the configuration illustrated in Figure 7.18c. This mechanism guarantees data consistency, addressing the challenges posed by the multiple words within a cache line.

7.4 The System Architecture

So far, the concept of ACCP has been introduced and discussed. Building upon this concept, the discussion extends to the two fundamental components: the *checkpoint capable register file* and the *checkpoint capable cache*. Both play essential roles as subsystems of a checkpoint-capable processor core, contributing to the overall architecture.

In the subsequent sections, the focus shifts to the processor architecture. Following that, the attention is directed to fault detection mechanisms and the checkpoint creation process.

7.4.1 Processor Architecture

The processor architecture smoothly incorporates both the *checkpoint-capable register file* and the *checkpoint-capable cache*. These components can be seamlessly integrated into the LEON3 pipeline as both subsystems follows the interfaces of the original components.

To support checkpoint creation and fault recovery mechanisms, the processing system needs to go beyond the register file and cache subsystem. The processor pipeline state must be a part of the CP. The pipeline should have the capability to create a CP and restore the internal state in case of CP recovery.

Micro-architectural modifications within the pipeline are necessary. The modifications related to program and internal state replication align with those presented for the ALP (refer to Section 5.3.2). Additionally, the micro-architecture must support the replication of the register file and the CP management of the internal state. Task replication comes into play when the processor switches from a non-reliable to a reliable execution, resulting in a reconfiguration from the *performance* to the *reliable mode*.

The AR concept supports the re-establishment of the suspended program, requiring the restoration of the complete internal state from the original program execution, including the data of the register file. The register file uses a dual-level checkpointing scheme, making it unsuitable to keep the original state within the register file. Instead, the state is offloaded to an external memory, while the register file state of the master core is taken.

All these features are implemented in the *CP Management stage*, positioned at the same location as the *commit stage* of the ALP. Placed between the *execution* and *memory stage*, both *commit* and *CP Management stages* complement each other and can be implemented within the same processor pipeline. The *CP Management stage* is responsible for task replication and checkpoint handling. The hardware handles checkpoint creation, rollback, fault detection, and CP organization, fully abstracting the reconfiguration and hides it from the software.

Figure 7.19 illustrates the pipeline and the processor architecture, encompassing two cores and the Redundancy Management Unit (RMU). The RMU orchestrates the reconfiguration process, forwarding register file data from the *master* to the *slave* core and receiving the slave register file data. The *snapshot* of the register file is stored in a secure memory located within the RMU, with the entire checkpointing process supervised by the RMU.

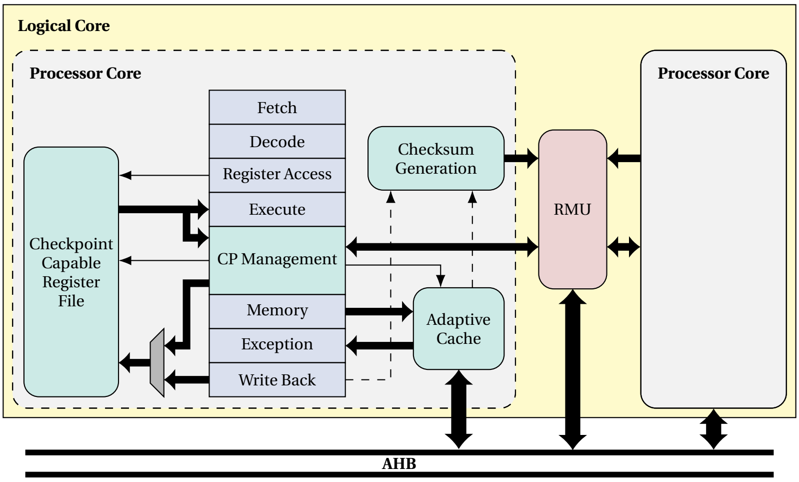


Figure 7.19: Processor and pipeline architecture of the ACCP (refer to [Kem22a])

A pipeline checkpoint does not necessarily need to encompass all processor registers. As discussed in the context of the ALP (refer to Section 5.3.1), individual pipeline registers are recoverable by re-executing the program code. To minimize the required checkpoint data, the pipeline registers are excluded, as they are not essential for a rollback and can be reconstructed by program re-execution. The *CP Management stage* offers registers specifically for storing the *snapshot*, consisting only of relevant pipeline registers identified during the state transfer discussion (refer to Section 5.3.2). In the *reliable mode*, the *CP Management stage* stores two states: one for recovering the original state and the other for the checkpoint. The pipeline employs a dual-level checkpointing scheme, mirroring the approach used in the register file and cache subsystems.

The state replication from the *master* to the *slave* core, involving the pipeline state with the control register, follows a process similar to that of the ALP. Both cores synchronize their pipelines to achieve two completely flushed pipelines. The flushed pipelines are suitable for replicating the *master* state to the *slave* and migrating the *slave* state. Unlike the synchronization phase in the ALP, the ACCP requires the transfer of register file data, necessitating a sequential load of each register value. During replication and migration, both processor pipelines are put on hold and stall. After successful replication, both cores create a checkpoint, and the pipelines resume independent operation. Processor cores synchronize and compare their internal states only when the conditions for creating a new checkpoint are met.

The scalability of the number of cores is not limited to two, as shown in the example configuration. The ACCP adopts a loosely coupled concept, where processor data does not need to be directly forwarded by the RMU. Data between RMUs can be tunneled through various interconnects, extending beyond centralized buses to include NoCs. This flexibility allows for various configurations, and the tunneling and distribution of *physical cores* associated with one *logical core* is feasible due to the reduced compression frequency.

7.4.2 Checkpoint Creation

A checkpoint is generated whenever *unsafe* data attempts to leave to the SoR. In the context of the ACCP, this occurs when the cache writes an *unsafe* cache line to a lower memory hierarchy, typically during the replacement of a *dirty* cache line. This replacement can happen in the case of a read miss, as discussed earlier, or a write miss. In the case of a write miss, the *dirty* cache line is initially written back, followed by the write-allocate and the write operation.

During checkpoint creation, the cache keeps all data internally and postpones the write-back process. Before the write-back occurs, both processors validate their respective states by comparing checksums over their processor states. If the checksums match, the data is written back, and a new checkpoint is established. Performing compression before the write-back is essential to prevent corrupted data from leaving the SoR and causing irreversible changes to the system state.

The frequency of necessary write-backs depends on the executed program and may vary, leading to a checkpoint creation frequency and period between checkpoints that are code-dependent. Consequently, controlling the maximum fault detection latency becomes challenging, as it depends on the unpredictable checkpoint creation frequency.

To address this and enhance control over fault detection latency beyond write-backs, additional conditions for creating a checkpoint are introduced. These conditions mitigate fault detection latency based only on write-backs.

A performance counter, incrementing with each executed instruction, serves to control the maximum fault detection latency. This upward counter resets each time a checkpoint is created and measures the instructions executed since the last checkpoint. It provides a deterministic and less software-dependent approach to checkpoint creation, based on the total number of executed instructions. The software defines the upper threshold. Every time the performance counter reaches the runtime-defined threshold, a new checkpoint creation is triggered.

Similar to the checkpoint creation triggered by a write-back, checksums of the *master* and *slave* core are compared in advance. If both checksums match, a checkpoint is established, and the program continues. Otherwise, the last checkpoint is restored, and the program is automatically re-executed.

As the threshold for the performance counter is runtime-configurable, it allows for adjustable maximum fault detection latency.

The process of creating checkpoints and detecting potential faults during execution is illustrated by the flowchart depicted in Figure 7.20. The initiation of checkpoint creation can be triggered either by a cache miss in the presence of unsafe data requiring write-back to main memory, or by surpassing a threshold of executed instructions. Checkpoints are generated whenever no fault is detected, and the watchdog timer remains within its limits.

7.4.3 Fault Detection

Safety-critical sections are redundantly executed in the *reliable mode,* where both the *master* and *slave* cores run identical programs. In this mode, the state of the master is replicated to the slave core, resulting in precisely matched state transitions. Ideally, flawless execution ensures that the *register file, cache,* and *pipeline state* undergo identical changes on both the *master* and *slave cores.*

Upon entering the *reliable mode,* a checksum is calculated over all state transitions. Faults are detected by identifying mismatches in write accesses to the register and memory files during the write-back stage and cache accesses. The checksum calculation can detect from a single to multiple faults. Each time a checkpoint is created or the processor rolls back to the last checkpoint, the checksum is reset to its initial value.

The checksum used for fault detection must be effective for error detection and straightforward to implement. Various checksums are known in the literature, such as the *xor checksum* [186], Cyclic Redundancy Codes (CRC) checksum [187], or *Fletcher's checksum* [188].

The *xor checksum* is easy to calculate and involves minimal hardware resources. However, its probability of undetected errors is higher compared to other considered checksums. A CRC checksum has a low

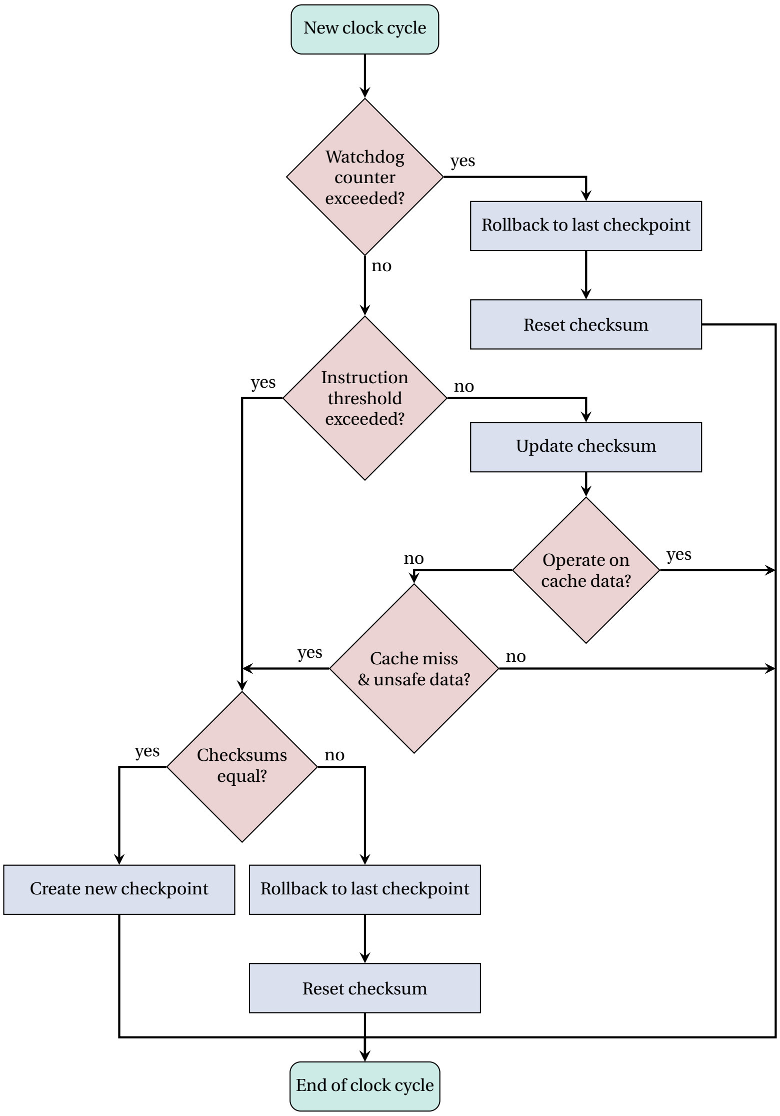


Figure 7.20: Flowchart illustrating the checkpoint creation process.

probability of undetected errors, but its calculation involves costly operations. Hardware realization of a CRC code based on shift registers is not suitable for high bandwidth and parallel data words [189]. A less hardware-intensive option, with a slightly smaller error detection property, is the *Fletcher's checksum.* For a detailed investigation of different checksums, Maxino et al. provide comprehensive comparison [190].

The *Fletcher's checksum* is based on two sums, as seen in Equation 7.1 and Equation 7.2. $sum1$ is updated with the new data word $data$, while $sum2$ is updated with the value from $sum2$. Both sums are then calculated modulo $M = 2^{32}$. The hardware implementation of the Fletcher checksum is simple and hardware coste-effective.

$$sum1 = (sum1 + data)\, mod\, M \tag{7.1}$$

$$sum2 = (sum2 + sum1)\, mod\, M \tag{7.2}$$

After discussing the type of checksum, the next consideration is the location of checkpoint creation. As previously mentioned, the checkpoint should cover the state transfers of the *register file, cache,* and *pipeline state.* Two approaches can achieve this. The straightforward method is to calculate individual checksums for each of the three components. The integrated way involves placing the checkpoint creation only in the *write-back stage* of the pipeline. All state transfers must pass through the *write-back stage,* encompassing not only the data written to the processor internal registers and register file but also the cache access. Since the address and data of a store instruction are split into separate phases, a single checksum can cover both. The address and data of the cache access traverse the entire pipeline. Calculat-

Table 7.1: Implementation results of required resources

	CLB LUTs	**CLB Registers**	**CLBs**	**BRAMs**
ACCP	14219	4364	3431	10
Cache	6792	2048	2069	8
Integer Unit	7073	1966	1209	0
Register File	354	350	153	2

ing the *Fletcher's checksum* at the *write-back stage* effectively covers all three components.

7.5 Evaluation

The ACCP architecture undergoes evaluation in terms of resources, fault-free execution time, and correct execution behavior. The correctness of execution is demonstrated through simulation and emulation on an FPGA platform. The investigation covers both fault-free scenarios and cases with injected faults. The validation of the ACCP architecture involves random fault injection, following the methodology described in Section 5.5.1.

7.5.1 Resources

A prototype of the ACCP based on the LEON3 design has been implemented as a proof of concept. The development involved modifying the original source code of the LEON3 processor core and the cache design. The primary focus of the proof of concept was on functional validation, with less emphasis on optimal hardware implementation and resource utilization. The presented values represent the upper bounds of the hardware realization, allowing room for potential implementation improvements. Similar to the AL processor, the implementation targets the Virtex UltraScale+ XCVU9P-L2FLGA2104E on the VCU118 evaluation board from AMD [151].

Resource utilization is evaluated by synthesizing the design using AMD's Vivado [152]. Standard strategies are employed for both synthesis and implementation in Vivado, with the design running at a clock frequency of 100 MHz without encountering timing violations.

The required resources are reasonable, as shown in Table 7.1. The cache configuration used is a 2-way associativity for the *performance mode*, with a cache size of 8 KiB. Each cache way has 4 KiB and four words per line.

7.5.2 Runtime Overhead

The runtime behavior of the ACCP architecture is evaluated using the same five benchmarks as for the ALP architecture (refer to Section 5.5.3). The benchmark applications are utilized with varying input sizes to assess the runtime overhead introduced by the write-back cache configuration without checkpoint creation and the additional time needed for checkpoint creation.

The evaluation is conducted on an FPGA prototype, involving three different configurations synthesized and implemented for the AMD VCU118 evaluation board. The first configuration represents the reference time, measuring the execution time of the processor design in *performance mode*. In *performance mode*, the cache uses a write-through write policy, and the complete memory is utilized. The second configuration employs the write-back configuration with half the cache size but with disabled checkpointing, showcasing the impact of the copy operation to ensure correct data in the active data region.

The third configuration uses the write-back cache with enabled checkpointing. The runtime evaluation of the second and third configurations includes the reconfiguration process and task replication from the primary to the secondary processor core.

Each of the three configurations is based on the previously presented architecture, utilizing a 2-way associative cache for the *performance mode* with 4KiB per way, resulting in a total 8KiB cache size and four words per line.

The observed runtime results are shown in Figure 7.21. The overhead is detailed in Figure 7.22 and Figure 7.23.

The runtime overhead introduced by the checkpointing mechanism is minimal and only applies to execution in the *reliable mode*. The measured runtime overhead difference between the write-back cache without and with enabled checkpoint creation ranges from 0% to 6.19%. For the Quicksort evaluation with small input sizes (100 and 500 elements), the overhead is 0%. The worst-case overhead is measured for the Quicksort algorithm with 10,000 input elements (6.19%) and matrix multiplication with an input size of 5x5 (5.4%). On average across all evaluated workloads, the overhead is measured at 2.04%.

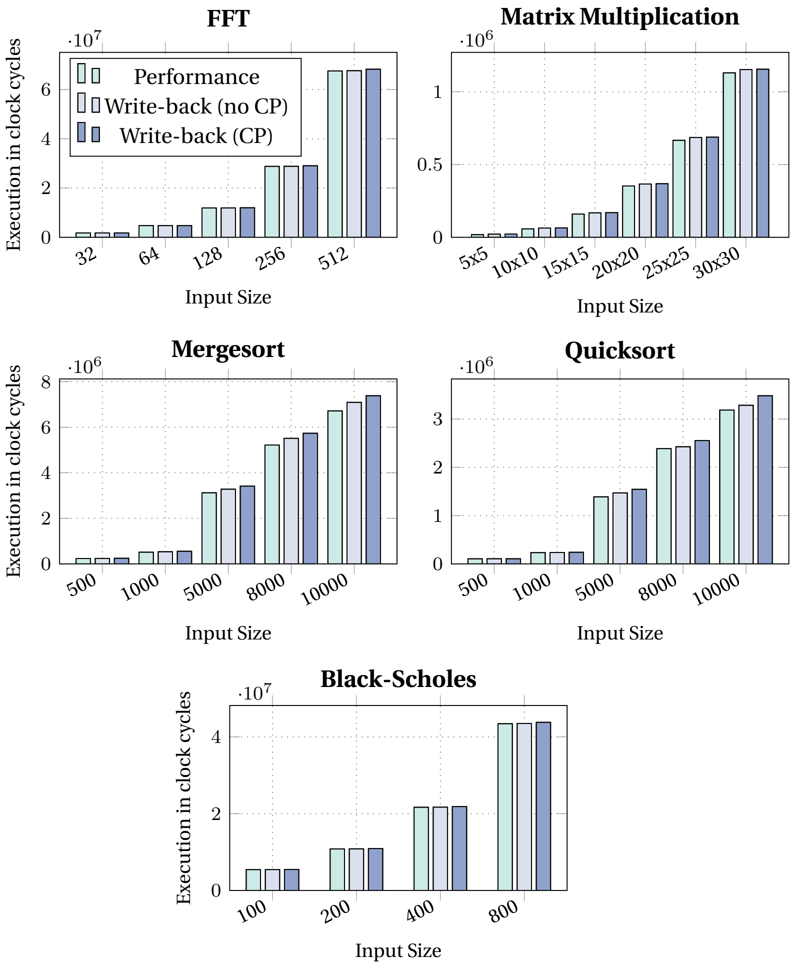


Figure 7.21: Execution times of benchmark applications for different cache configurations with various sizes of the input sets. Performance mode (green), write-back cache with disabled checkpointing (light blue), write-back cache with enabled cache checkpointing (blue)

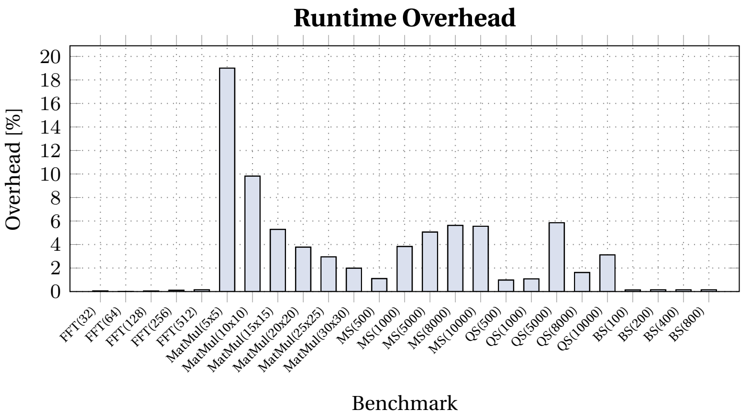


Figure 7.22: Relative runtime overhead introduced by the write-back implementation of the *reliable mode*. It is compared to the *performance mode*.

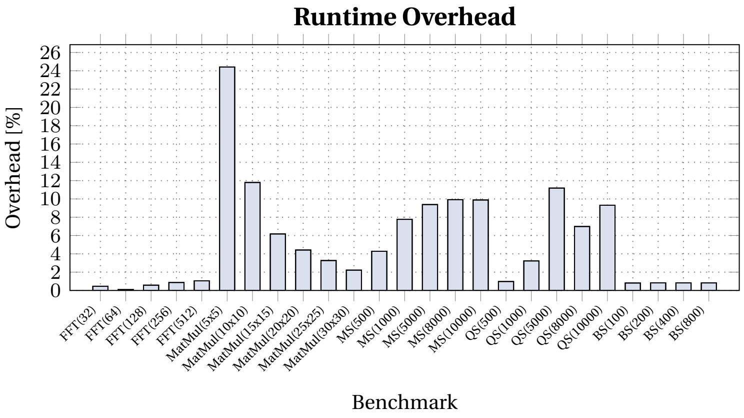


Figure 7.23: Relative runtime overhead introduced by the write-back and checkpoint management implementation of the *reliable mode*. It is compared to the *performance mode*.

Table 7.2: Time between checkpoint creations by write-backs

Algorithm	Clock cycles between write-backs		
	Min	**Max**	**Avg**
FFT	13	39,910	10,131.3
Matrix Multiplication	15	10,031	1,272.7
Mergsort	13	635	158,7
Quicksort	13	126,861	663.1
Black-Scholes	13	5,376,601	251,259.1

7.5.3 Fault Detection Latency

For the same cache configuration as previously, the error detection latency is evaluated by measuring the time between two consecutive checkpoints. The evaluation of error detection latency is dependent on the automatic checkpoint creation, considering only checkpoints created by write-back operations of the cache, as the RMU can configure an individual checkpointing interval. Hence, the performance counters of the RMU are disabled. The results for different benchmark algorithms are presented in Table 7.2. It is obvious from the results that the checkpoint dependency is closely related to the executed software. The Black-Scholes algorithm, benefiting from high data locality due to a substantial number of software floating-point operations, exhibits a larger interval between checkpoints compared to the other algorithms.

7.6 Summary

This chapter discusses the coarse-grained ACCP processor architecture, with the SoR situated at the cache level. The checkpoint data, crucial for rolling back to the last checkpoints, must encompass the cache and all memory elements at that level of the memory hierarchy. Consequently, the ACCP concept extends to the cache, the register file, and the processor pipeline to make the creation and management of checkpoints possible. The hardware implementation not only demonstrates the feasibility of the concept but also showcases a minimal introduced runtime overhead.

In general, this concept and the resultant modifications are applicable to various processor designs, including out-of-order, and are not limited to the example design.

Chapter 8

Interleaved Processor State Transfer

This chapter introduces the concept of interleaved processor state transfer and explore its corresponding hardware realization. The ACCP employs a time-consuming state transfer mechanism designed to replicate the master state onto the target slave. Throughout the entire replication process, both the master and slave temporarily halt to ensure a coherent processor state. The interleaved state transfer explores parallelizing the state transfer alongside program execution. The chapter systematically examines the fundamental concepts, investigates a comprehensive analysis of various aspects. Building upon these concepts, an architectural extension for the ACCP processor is discussed. The chapter concludes with a HDL evaluation of this proposed architecture. The fundamental for the interleaved processor state transfer, including its conceptualization and implementation, was initially published in [Kem23a]. This chapter serves as an extension and elaboration of the concepts presented in that publication.

8.1 Interleaved State Migration and Replication

Before a target core can execute a migrated or replicated task, a snapshot of that task must be transmitted to the target processor. In a shared memory system, where both the source processor core and the target core access a common main memory, there is no need to physically relocate the associated memory contents. In such systems, it suffices to migrate or replicate only the internal processor state.

Migrating or replicating a software task involves capturing the internal program state of a processor core and transferring that state to the target. The internal processor state encompasses values stored in general-purpose registers, dedicated processor state registers, and, potentially, pipeline states. Processors commonly employ an SRAM-based register file to implement general-purpose registers and locate processor state registers within the core structure.

In the case of the SPARC v8 RISC ISA, these processor state registers are designated as control and status registers. The ISA specifies these as 32-bit registers, and they are enumerated below:

- Program Counter (PC)
- Next Program Counter (NPC)
- Processor State Register (PSR)
- Window Invalid Mask (WIM)
- Trap Base Register (TBR)
- Multiply/Divide Register (Y).

In addition to the specified state registers, the SPARC v8 ISA specifies a range of 40 to 520 32-bit general-purpose registers. Due to the substantial number of registers, these are conventionally implemented using an SRAM-based register file. A register file employs ports to access individual registers, though unlike flip-flop based registers, not all register values are accessible simultaneously.

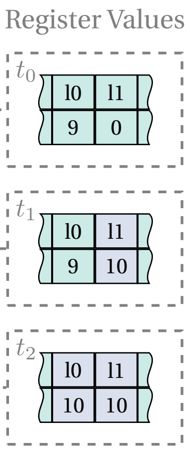
The migration or replication of processor state registers can be accomplished in parallel, given their small number and simultaneous accessibility. However, copying or transferring data of a general-purpose register to or from a register file must be performed sequentially. This is a time-intensive process, depending on the complexity and structure of the processor architecture and the quantity of registers.

Throughout the migration or replication process, the source processor must maintain a consistent internal state that is readable. Otherwise, discrepancies in the behavior of the target core may arise. The C code snippet in Listing 8.1 illustrates how ongoing program execution can impact the replication process. The target core commences execution from the statement following the marked line where the replication process initiates. Continued program execution has the potential to change register values, thereby influencing code execution on the target core. Consequently, the C code snippet presents register values at three distinct points in time, emphasizing that the executed code's behavior depends on the actual register values read from the source core.

When the register data is read from the source core at t_0, both the source and target cores enter *block 1*, show identical behavior. The same holds true when the data is read at time t_1, as changes made by the source core to the register *l1* do not impact the target core's execution. By having the target core overwrite the register value, the loaded value becomes insignificant to the software execution post-replication. However, the scenario at time t_2 results in unequal and different behavior between the source and target cores. In this case, the target core sees the value of *l0* incremented twice - once from the source core and once from itself - causing it to execute *block 2* while the source core continues with *block 1*.

Listing 8.1: Demonstrate the impact of continued program execution on the replication process. Depending on the register values copied, the target core enters either block 1 or block 2.

```
...
l0 =9;
l1 =0;
// (Start Replication Process)
l1 = 10;
l0++;
if(l0 > l1)
{
  // Do Something (Block 1)
} else
{
  // Do Something Else (Block 2)
}
...
```



This example illustrates that the source core cannot seamlessly resume code execution, and task replication or migration must carefully manage the reading of register values from this core. Depending on when these values are read, the target core's behavior can vary. While cases where the transferred program overwrites register values result in identical behavior for both cores, in most situations, the target core operates with distinct values, depending on different outcomes from the source core. Therefore, the snapshot must be write-protected during the replication or migration process.

Ensuring correct behavior on the target core requires that the register values transferred from the source to the target core remain consistent with the data at the start of the migration or replication process.

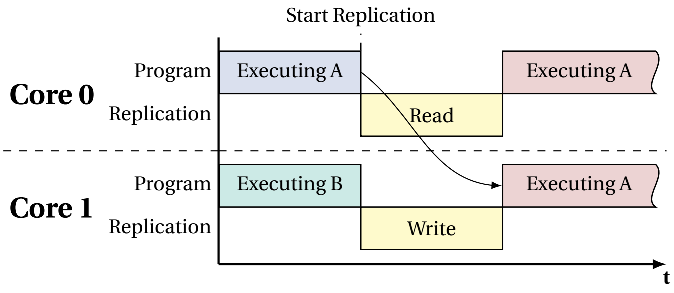


Figure 8.1: Task replication of program *A* from *core 0* to *core 1*. During the replication process both cores pauses the code execution. (refer to [Kem23a])

One approach involves the source core pausing execution until the migration or replication process finishes, as illustrated in Figure 8.1 and used by the ACCP architecture in Section 7.4.1. Here, *Core 0* (source core) runs program *A*, set to replicate to *Core 1*. *Core 1* takes over the state of *Core 0*, discarding the execution of program *B*. After completion, both *Core 0* and *Core 1* run the same program *A*.

Throughout the replication process, register values are read from source *Core 0* and transferred to target *Core 1*, with the latter writing these values into its internal registers. Both cores are halted during this process, ensuring *Core 0*'s internal state remains unchanged. This pause and block mechanism guarantees identical functional behavior for both cores.

The period during which both processors are paused and idle is exclusively dedicated to the transfer of processor state data. For the guarantee of internal state consistency, no additional instructions are executed on the source core during this time, preventing progress in program execution. However, if the source core is capable of establish and provide the original register values from the start of task replication or migration, it can resume execution concurrently. To achieve this, the core must be enabled to store a snapshot of its state. Throughout the migration or replication process, the snapshot data is retrieved from the source core. When task migration or replication does not disrupt code execution and the source core provides a consistent state, it can be seamlessly interleaved with normal program execution. This interleaving of state provision and concurrent program execution minimizes performance penalties on both source and target core.

In Figure 8.1, not only does the source core pause, but the slave core does as well. The execution of the acquired program on the slave core requires all relevant register information. This is ensured by pausing until the complete system state is written to the internal process registers. However, running a program does not instantly require all state information; it is sufficient if program-relevant data is available when needed. For a migrated or replicated task, this implies that the full internal pipeline state does not need to be accessible before executing the task. To ensure correct execution of the replicated task, it is sufficient to provide the correct register values of the source processor or the snapshot only when necessary. In the absence of this required data, the target core must guarantee correct program execution. Speculative execution results into performance gains for the target core, as program execution and task migration or replication can proceed in parallel. The migrated or replicated software runs speculatively, assuming the required data is available, while simultaneously, the internal processor state is migrated or replicated to the target core.

Figure 8.2 illustrates an ideal scenario of interleaved task migration, where both processor cores remain active without idling or pausing, seamlessly transitioning from program *B* to program *A* on *Core 1*, while *Core 0* continues executing program *A*. At the replication process's outset, a snapshot of *Core 0's* state is captured. During execution, the snapshot state is read from *Core 0*, transmitted to *Core 1*, and written to its internal registers. The interleaved read and write operations

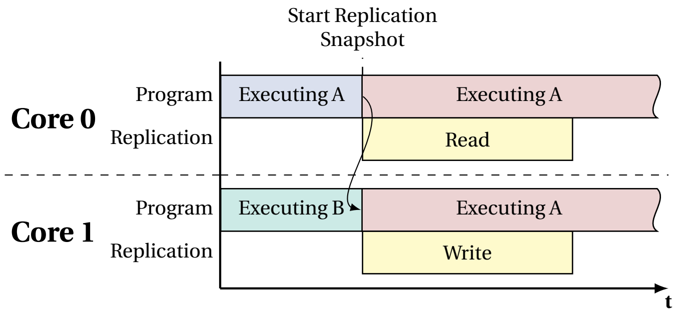


Figure 8.2: Task replication of program *A* from *core 0* to *core 1*. The the replication process is done in parallel. (refer to [Kem23a])

might take longer than a sequential procedure due to processor architecture and register access restrictions. However, the essential metric is the execution time, not the duration of transferring the complete system state.

Achieving seamless execution of the transferred task on the target core depends on the availability of necessary data. Whenever an instruction reads register values, those values must be present on the target core. Otherwise, a hazard management strategy is applied, halting the processor core until the correct data is available. Therefore, the migration or replication process should anticipate and predict the next required state value.

Anticipating the next required state value involves the state-keeping process predicting the next register access of the slave core. With perfect anticipation, each register value is accurately predicted, allowing executing instructions on the slave core to read those values.

Concurrent program execution and task migration or replication can minimize the time that the source and target processors stall, enhancing overall performance. However, parallel program execution on both cores may disrupt the interleaved task takeover process. Even with perfect prediction of the next required state values, architectural limitations may prevent the processor cores from providing or taking over the state value.

Therefore, it is essential to identify when it is feasible to concurrently run code while reading or writing the processor state.

8.2 Analysis of Suitable Time Slots

Understanding when the source core and slave core can concurrently transmit a system state alongside regular code execution is essential. Specifically, it is important to determine when the processor is accessing a register and when the overtaking process can access those registers without disturbance and interference.

Moreover, it is significant to identify when each register is accessed, the frequency of access, the number of simultaneous accesses, and the dependencies of register accessibility. Certain register values, such as the PC, are essential for the processor; without the correct PC, incorrect instructions are fetched, leading to erroneous code execution. These critical register values must be available when the migrated or replicated task's code is executed and are typically stored in dedicated registers within the processor. These dedicated registers have a high degree of accessibility, enabling simultaneous reading of all these values. In contrast, general-purpose registers are often located in restricted-access register files, and the number of simultaneous accesses depends on the register file's ports. Consequently, these general-purpose register values become essential candidates for parallel task replication and code execution. Register values are exclusively read from the read ports and written to the write ports, with each port independently accessing a specific register.

Both the context switch process and code execution tries to access a common and shared register file, necessitating arbitration to resolve potential conflicts arising from concurrent accesses. The frequency of conflicts generated by the task migration or replication process and parallel code execution depends on various factors. These factors include:

- Maximum quantity of simultaneous register access of the processor
- Actual number of register accesses by the software
- Number of register-file ports.

The register file represents a essential bottleneck in transmitting the system state. Consequently, beyond comprehending the register file's functionality, it becomes necessary to understand its integration into processor architectures. Additionally, understanding how and when a register file is accessed proves to be another pertinent factor.

The design of a register file deeply influences processor performance, as it features dedicated read and write ports that dictate its accessibility and, consequently, overall efficiency. When multiple registers are accessed simultaneously, and the register file lacks sufficient parallel ports, a structural hazard emerges. Resource contention arises when the register file provides fewer ports than an instruction demands - for instance, if an instruction requires access to two registers simultaneously but the register file only has a single read port.

In Figure 8.3, a processor configuration with a single-port register file is illustrated. Arbitration is necessary not only for read accesses but also for write accesses, and simultaneous read and write accesses introduce a structural hazard.

To resolve this resource conflict, the processor must stall while loading both operands of the instruction sequentially from the register file.

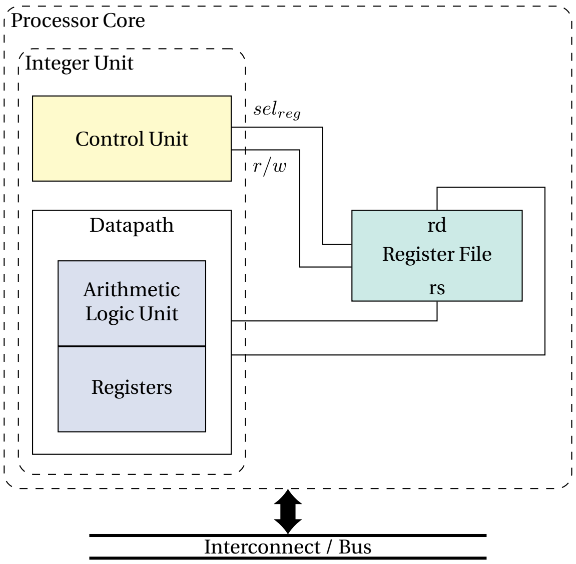


Figure 8.3: Processor core with a single ported register file

Sequential loading effectively mitigates structural hazards and the associated resource conflicts.

An alternative approach to address structural hazards caused by a register file is prevention. This involves ensuring a sufficient number of read and write ports for the register file. The maximum number of parallel register accesses by the processor is constrained, with the ISA specifying this limit for single-issue processors. ISAs define the actual number of operands and destination registers for each instruction, directly determining the maximum values for operands and destination registers. Typically, RISC ISAs specify a maximum of two operands and one destination register. To prevent structural hazards, the number of register file ports should align with the maximum operands and destination registers stipulated by the ISA.

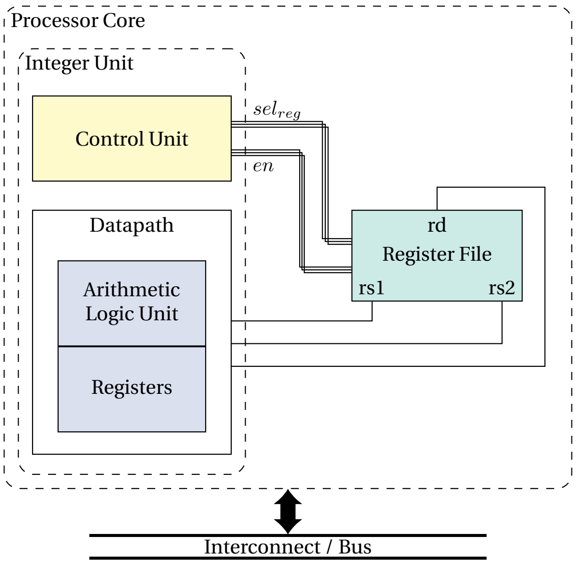


Figure 8.4: Processor core with a triple ported register file

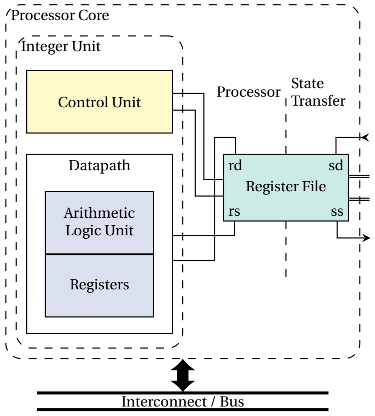
In Figure 8.4, an example processor architecture is illustrated with a register file offering two read and one write port. The read ports (*rs1* and *rs2*) and the write port (*rd*) are activated by corresponding *en* signals. The actual register is addressed and selected through the sel_{reg} signal. The interaction of sel_{reg} and *en* controls whether register values are read or the input of *rd* is written to the selected register.

In the context of parallel code execution and task transfer, enhancing the register file can involve the addition and extension of dedicated ports reserved explicitly for the task transfer process. This modification aims to prevent structural hazards in the processor's register file. The design ensures concurrent access to register values by both the processor and the transfer task, facilitating simultaneous read and write operations.

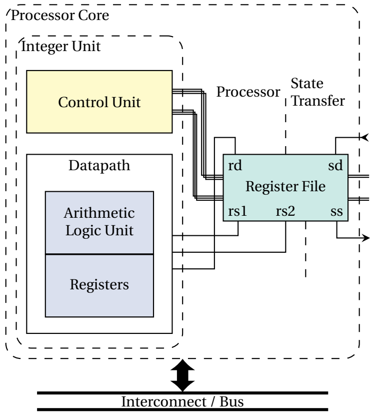
However, simultaneous read and write accesses, both from processor ports and a dedicated task transfer port addressing the same register, may lead to conflicts. The register file must resolve these conflicts, requiring a contention resolution strategy, especially in a pipelined processor. This strategy, similar to those for handling conflicts between processor reads and writes, must also address conflicts arising between the processor and the task replication or migration process. The chosen strategy should strike a balance, ensuring uninterrupted code execution and accurate task transfer.

A dedicated port, whether for reading or writing status values to or from a register within the register file, suffices for a single task transfer process. However, when a task is replicated or migrated to a target core, simply overwriting the target's state is insufficient. Typically, the target's state needs either transfer to another core or preservation for subsequent continuation. Therefore, it becomes essential to save the current state before accepting the new state. A register file equipped with a dedicated port for task transfers can either read or write state. Importantly, the read port of the register file can provide the current state of the register addressed by the write port, enabling the preservation of the original state of the target core.

Both Figure 8.5a and Figure 8.5b illustrate a register file configuration tailored for a processor architecture featuring dedicated task transfer ports alongside ports for either one read and write port or two read and one write ports. In both cases, an additional register file port is introduced for reading and writing. The status source port (*ss*) reads the register's status, while the status destination port (*sd*) writes the status into a register. The task transfer process can utilize either port for reading or writing, with the *ss* port providing the current state of the register addressed by the write port, as previously described.



(a) Configuration for a single ported register file.



(b) Configuration for a triple ported register file.

Figure 8.5: Comparison of two different state transfer realization for a original single and triple ported register file.

The utilization of extra dedicated ports for task transfer facilitates nested task transfers alongside regular program execution. However, it's essential to note that increasing the number of ports augments the logical size of the register file. Additional hardware resources are typically costly and should be minimized whenever feasible.

Consideration must be given to how the processor architecture and infrastructure can be leveraged to interleave task assignment with program execution. For instance, not every instruction execution requiers the full utilization of all available register file ports. When a register file read port is unused during an instruction, the task replication or migration process can read a register value without interfering with code execution. Similarly, the same principle applies when the processor pipeline does not write to general-purpose registers.

Understanding how and when the processor architecture accesses the register file is essential, closely linked to the ISA and its specific processor implementation. To seamlessly nest interleaved task transfers into regular program execution, register accesses must be scheduled without disrupting the ongoing program execution.

In the subsequent discussion, this concept will be explored based on the underlying LEON3 realization of the SPARC v8 ISA. TheSPARC v8 ISA specifies source registers (*rs1* and *rs2*) from which operands are read, and a destination register (*rd*) to which results are written.

The LEON3 implementation employs a register file featuring two read ports and one write port dedicated to handling *rs1*, *rs2*, and *rd*. The specific number of operands and the corresponding register accesses depend on the sort of the executed instruction. The accessed general-purpose registers can range between two, one, or none, contingent on the instruction being executed. The following SPARC v8 assembly instructions illustrate accesses to two, one, and no source registers:

- **add rs1**, **rs2**, **rd** (two operands, one result)
- **add rs1**, **imm**, **rd** (one operand, one result)
- **sethi** %**hi**(value), **rd** (no operands, one result)
- **st** %**rs1**, [%**rd**] (two operands [rs1, rd], no result)

The results of the first three instructions are stored in the register file, with the write port accessed by the pipeline's write-back stage to write the results into the designated destination registers. The fourth instruction involves writing the value of the *rs1* register to the memory location specified by the address stored in the *rd* register.

Figure 8.6 illustrates the fundamental structure of the LEON3 pipeline, highlighting the register file access. The integer unit is comprised of a pipelined architecture that interacts with the register file. This figure illustrates the interconnections between individual pipeline stages

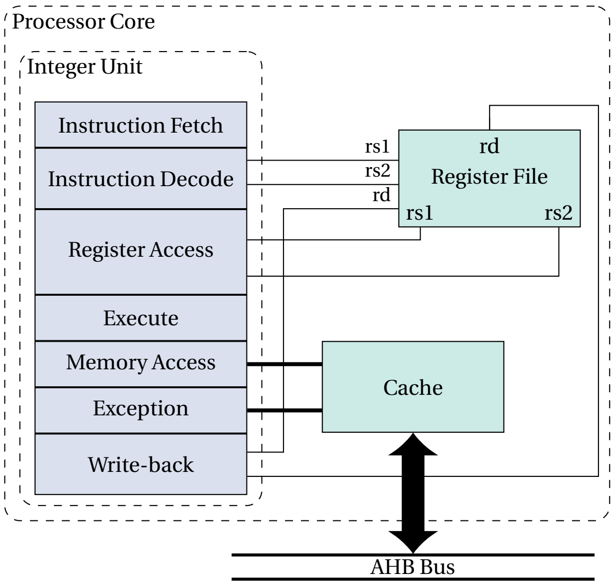


Figure 8.6: LEON3 core architectures accessing the triple ported register file to prevent a *structural hazard*.

and the register file. As previously mentioned, the register file is equipped with two read ports (*rs1* and *rs2*) and one write port (*rd*).

The read ports are linked to the register access stage of the integer unit. In this stage, operands are fetched from the register file and subsequently forwarded to the execution stage. The write port, on the other hand, is accessed by the write-back stage, where the results of an instruction are written back into a register.

The pipelined architecture of a processor introduces the potential for a Read-After-Write (RAW) hazard—a pipeline hazard resulting from the structure of the pipeline and the partial completion of instructions. A RAW hazard arises whenever an instruction needs to read a result from an incomplete instruction, where one instruction's operand relies on the result of a previously issued instruction.

Listing 8.2 illustrates a **Read-After-Write (RAW) hazard**. The *sub* instruction is dependent on the register values *l2* and *l3*, but the value in register *l2* is dependent on the outcome of the preceding *add* instruction. The result of the *add* instruction is stored in the *l2* register. Without any conflict resolution mechanism, the *sub* instruction must wait until the *add* instruction completes the write-back stage and updates the *l2* register. For the LEON3 processor with seven pipeline stages, the *sub* instruction can only be executed once the *add* instruction has finished writing to the register during the write-back stage.

Listing 8.2: Example of Read-After-Write (RAW) hazard. The operand l2 of the seconde add operation depends on the result (l2) of the first add operation. The processor architetcure has to ensure a correct functional behavior.

```
add l2 , l0 , l1  /* l2 = l0 + l1 */
sub l4 , l2 , l3  /* l4 = l2 - l3 */
```

Addressing a Read-After-Write (RAW) hazard by stalling the processor pipeline until the register value is written proves to be inefficient. During the processor stall, no further instructions are processed, despite having the required results already present within the pipeline. In the earlier example, the result of the *add* instruction is available after the execution stage. Instead of waiting until the register value is written during the write-back stage, the result can be forwarded within the pipeline.

Data forwarding serves as a pipeline optimization specifically targeting at mitigating pipeline stalls caused by RAW hazards. This approach involves directly forwarding intermediate results within the pipeline to the execution stage, thereby reducing pipeline stalls resulting from RAW hazards.

An additional benefit of data forwarding is the reduction in register file accesses. When an operand is forwarded from a pipeline stage, it does not need to be read from the register file. Consequently, the read port of a register file is less utilized, leading to a decrease in register file accesses.

The full read and write access capability of a register file is rarely fully utilized. As previously discussed, particular instructions involve one or no register operands, there is a data forwarding structure that diminishes RAW hazards, or instructions do not use general-purpose registers as target registers. Whenever the processor pipeline does not access all of the register file's ports, these available ports can be repurposed for other tasks.

These tasks may involve reading or writing the system state. Whenever the read port is not fully utilized, the system state can be read. Similarly, if the write port of the register file is not accessed, it is possible to overwrite the system state. Importantly, these register file accesses can be scheduled without interfering with the register access of regular program execution.

Figure 8.7 illustrates the observation of exemplary register accesses from the LEON3 IU to the register file recorded during an actual program execution. The example includes three forwarded operands, marked with brackets. Various points in time are highlighted in the image. Time slots marked in red are suitable for reading data from the register file. Green-marked time slots are appropriate for both reading and writing data from or to the register file. During the blue-marked time slot, no register access occurs.

In the context of a concurrently executed context-switching task, additional register accesses are generated that are unrelated to the cur-

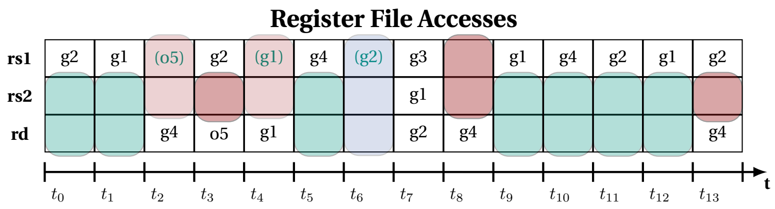
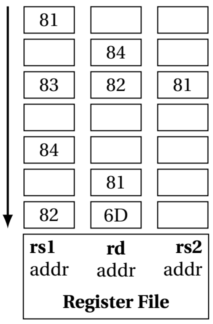


Figure 8.7: Exemplary register file generated by a executed program. Values are the accessed registers. Values in brackets are discarded register accesses due to data forwarding. Highlights in green are one simultaneous available read and write slot, in red one or two read slots and blue no register port is accessed.

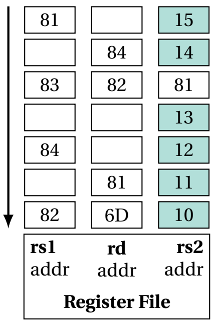
rent program execution. However, it is essential to ensure that these additional accesses do not collide and disrupt the regular program flow. As discussed earlier, one approach involves employing an additional register file for this purpose. Alternatively, the temporal underutilization of a register file can be leveraged. Instead of a spatial hardware extension, a temporally optimized hardware approach proves to be cost-effective.

The previously illustrated example highlights time slots suitable for interleaving a task transfer with the regular register access of program execution. Scheduling the read and write access of the context switch during these opportune points in time allows for a harmonious integration of the task transfer without impeding the regular program flow.

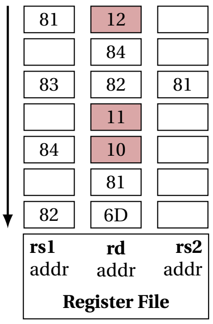
In Figure 8.8, various scenarios of interleaved task transfer are illustrated. Figure 8.8a displays the baseline of register access, generated by the executed code, similar to the example presented earlier. Actual register addresses are used in this representation, translating the register names as required by the SPARC ISA.



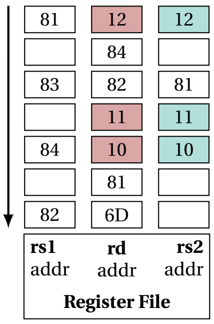
(a) Register access generated by the program



(b) Register access generated by the program interleaved with read accesses of the task transfer process.



(c) Register access generated by the program interleaved with write accesses of the task transfer process.



(d) Register access generated by the program interleaved with simultaneous read and write accesses of the task transfer process.

Figure 8.8: Comparison of different temporal interleaved register file accesses.

In Figure 8.8b, the register accesses of the regular program execution and the interleaved state reads are illustrated. The green-highlighted read accesses of the *rs2* port are related to the task transfer. Feasible state write accesses, nested within the ordinary program accesses of the *rd* port, are shown in Figure 8.8c. The red write accesses of the *rd* port indicate the writing of the system state. As previously discussed, the system state may not only be overwritten but also offloaded—either transferred to another core or saved for later resumption.

In Figure 8.8d, a simultaneous read and write of the register is illustrated, assuming a read-first strategy in the register file. The read-first strategy outputs the register value before it is overwritten with the new value, in contrast to the write-first strategy that outputs the newly overwritten value.

For simultaneous read and write of the system state, the register file ports *rd* and either *rs1* or *rs2* must be available. The availability is contingent on regular program execution. However, due to additional constraints on register port availability, there may be fewer time slots available than individual and independent read and write accesses.

The low utilization of register file ports enables the scheduling of additional read and write accesses interleaved with the regular stream of register file accesses generated by the executed program. Dedicated time slots allow task transfer accesses to be nested into the register access stream without interfering with code execution. Therefore, interleaving program execution with task migration or replication does not necessitate additional register file ports, thanks to the temporal utilization. However, it's important to note that not every time slot is feasible.

8.3 Analysis of Register File Accesses

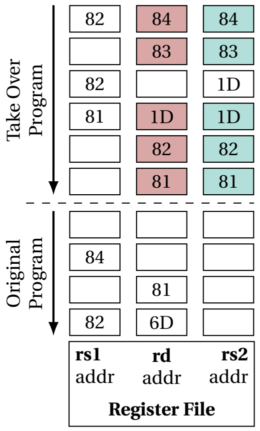
So far the concept of interleaving the task migration or replication task with a regular code execution is presented. Further, suitable mechanism for a nested context transfer into the regular register file accesses is discussed. Both additional hardware ports or scheduling accesses between program generated read and write accesses are feasible solutions.

However, the overtaken program required registers accesses has not been discussed in detail yet. It is necessary to that the processor state transfer process provides register values in time. It is necessary for a correct program execution of the overtaken program, that the required values are already present in the register. As discussed the bottleneck of the register transfer are register values allocated within the register file.

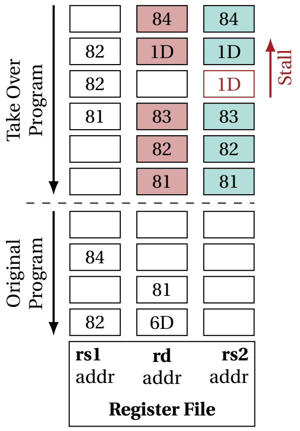
The process transferring or copping the register values from the source processor to the target processor has to write the target register before a instruction operand accesses the register. In Figure 8.9a this is illustrated. In this example every read register is beforehand written by the state transfer process. The overtaken program executes uninterrupted. Thereby, processor stalls and the resulting performance losses are prevented.

However, it is not always possible to provide the register values in time. In these cases the processor has to stall until the register value is overtaken. Figure 8.9b demonstrates such a case. The processor stalls until the not available register value is written by the state overtaking process. The processor stalling results in performance degradation.

The difference between both examples is the order of overtaken register files. The ideal case provides register values just in time. While the non ideal case provides the value of register associated to *1D* not



(a) Ideal provision of register values. The processor can executed the code uninterrupted.



(b) Non ideal provision of register values. The processor has to stall until a register value (highlighted in red) is provide by the task transfer process.

Figure 8.9: Ideal and non ideal register provision for the overtaken program execution. In the non ideal case the processor has to stall and can not fetch further instruction. (refer to [Kem23a])

in time. The processor stalls as long as the value of *1D* is written. The stall lead to a degree of performance degradation. In these example the performance degradation is visible by a later access of the register associated with the address *82*. In the non-delegated case in Figure 8.9a the last exemplary register access of *rs1* is *82*. However, due to the stall and performance degradation in Figure 8.9b the register access of *82* is no longer present. Whereby, both example illustrates the same number of register accesses.

Providing the necessary register values to the overtaken program requires unknown knowledge of the next register accesses. The state transfer processes has to anticipate the next required register value. This implies future knowledge, which usually is not present in the system. Future register accesses depends on the executed instruction of the overtaken program. Neither the correct control flow of the program nor the selected operands can be accurately predicted.

A accurate prediction of the control flow is impossible due to conditional statements. Conditional statements are dependent on various factors like previous executed code or input date. The necessity to perform control flow instruction can not be bypassed. Therefore, it is impossible to known the future control flow without executing each instruction. Because the exact program flow cannot be predicted exactly. It is not possible to know which registers are accessed next.

However, it is possible to estimate the next register access. The probability a register is accessed is not equally distributed. There are register which are more likely to be accessed. As already discussed the PC register is frequently accessed and must be available at the start of the execution of the overtaken program.

There are register within the register file which could be more likely to be accessed. For example the *SP* and *FP* are stored in the register file. Certain register could more likely to be accessed but there is also a correlation between register accesses.

Listing 8.3 demonstrates the exemplary the correlation between register accesses. It shows how results are directly reused by the next instructions. The example also shows dominating usage of *global* register. Other registers defined by the SPARC ISA like *local* or *in* are not used by the code snippet. However these register may have a stronger usage in other program routines. Therefore, the correlation of register accesses depends on the current PC. These correlations are difficult to model and would require additional logic or look up tables. Another solutions are runtime trained predictor. Which would require further additional hardware and introduces additional complexity. Because of the complexity, no online training of a predictor is considered at first. The prediction of the next needed register value is done by a static predictor. This is based on the access probability of each register file register.

Listing 8.3: Code example from the newlib memcpy function. It demonstartes a correlation of register accesses. Compiler optimze for data locality and register reuse. This creates a local correlation between register accesses. The code demonstartes the correlation between results and reuse of this.

```
ld     [%g2 + 8], %g4
ld     [%g2], %o5
add    %g1, 0x10, %g1
st     %o5, [ %g1 + -16]
add    %g2, 0x10,  %g2
st     %g4, [%g1 + -8]
ld     [%g2 + -4], %g4
cmp    %g3, %g1
bne    0x80009848
```

In the following the register accesses of the register file are discussed. The register access probability heavily on the compiler and conventions where data is placed. The SPARC v8 defines a software conventions which unifies register access. Compilers transforming software into SPARC v8 instructions follows these conventions. A static analysis of different compiled programs reveals frequently accessed registers.

8.3.1 Static Analysis of Register Accesses

The static analysis of register accesses is based on the executable program which is independent of the original programming language. The executable file is the output of the compilation and linking process and can be executable by the processor. It follows the structure and organization as presented in Section 4.2.2.

The input is a disassembly of the code section which is loaded into the main memory. The disassembled code consists of individual instructions which follows the SPARC assembler syntax as previously used.

The static analysis do not consider an actual execution model. Instructions are considered individually without context information. During the analysis process the program behaviors is neglected. For example loop estimations are not consider. This results into an abstract and simplified model of the program behavior. However, the target of the static code analysis is to understand which registers are likely to be used by the executed program. Furthermore, the static code analysis is used to estimate the register accesses.

The static code analysis allows rough estimation of which register is access. Further, it is able to determine how many instructions simultaneously accesses *rs1* and *rs2*. Without an actual pipeline model it is not possible to make a statement about simultaneously accesses of *rd* and *rs1* or *rs2*.

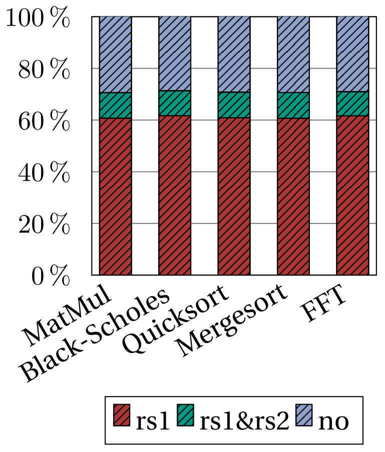
The static code analysis just considering individual instructions without considering previously instructions. Possible data forwarding is excluded. The static code analysis is used to quickly estimate register accesses. An actual pipeline model would add additional complexity. A exact determination of register accesses is measured in Section 8.3.2. In this section the static analyzed code is executed on a modified LEON3 processor. The processor is extended by performance counters monitoring the register accesses.

In the static code analysis, only the accessed registers and the associated register file port are investigated. It considers beside the SPARC ISA specifications additional implementations details of the LEON3 processors.

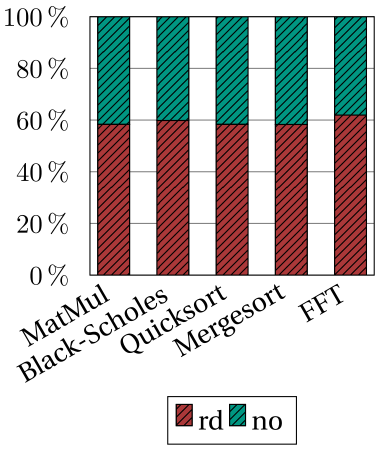
Listing 8.4: Examplary instructions for satic code analysis.

```
add  %g1 ,  %g2 ,  %g3
ldd   [%g  2+  8] ,  %g4
st     %o5 ,  [%g2]
std   %g4 ,  [%g2]
```

The LEON3 processor uses the triple ported register file to simultaneous read and write. The register write of the instruction does not occur at the same time. The write access occurs with delay. Listing 8.4 consists some example instructions which should be used to explain the code analysis. The **add** %**g1**, %**g2**, %**g3** increments the access of the *g1* register of the *rs1* port and the *g2* register of the *rs2* register file port. ldd [%**g2** + 8], %**g4** is a double load, which accesses the *g2* of the *rs1* port only for the address generation. The *8* is decoded as *simm13* of the instruction and is used as offset. The address calculation is done ony once and two values from that calculated address and the following are written into register *g4* and the consequent *g5* register. The store instruction **st** %**o5**, [%**g2**] writes the value of *o5* at the address pointed by the register value of *g2*. The LEON3 uses a multi cycle instruction to store the value. The first cycle is used for the address phase. The second cycle writes the value to the address location. The instruction has two accesses to the register file. However both are in different cycles and uses the same register port (*rs1*). The same applies for **std** %**g4**, [%**g2**], where the register values *g4* and *g5* are written in two consecutive clock cycles.



(a) RS Ports



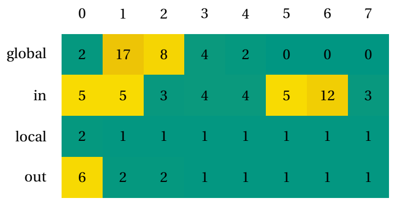
(b) RD Port

Figure 8.10: Static analysis of register accesses. It shows how often read register ports rd1 and rd2, as well as the write register port rd are accessed by the static program codes

Figure 8.10 illustrates the evaluation results obtained from static code analysis. The analysis encompasses five benchmarks employed in evaluating both ALP and ACCP, utilizing the maximum input dataset for each evaluation. Specifically, Figure 8.10a details the breakdown of instruction accesses for *rs1*, both *rs1* and *rs2*, or no access to the *register file*. Remarkably, the results exhibit striking similarity across all five benchmarks, with differences being less than 2%.

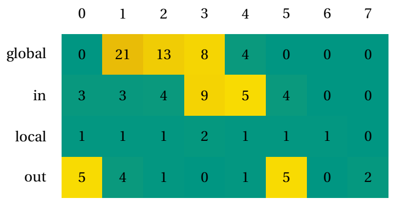
Examining Figure 8.10b, it becomes evident that approximately 60% of instruction accesses target the write port of the register file. Mirroring the pattern observed in the *rs* ports evaluation, the results display a minor deviation, not exceeding 4% for any of the benchmarks.

Concluding the analysis, the investigation turns to individual register accesses. In Figure 8.11, the results for the most frequently accessed



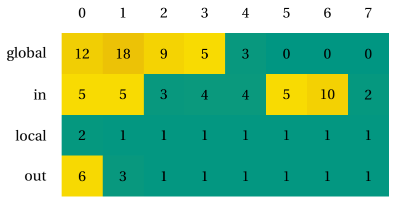
	0	1	2	3	4	5	6	7
global	2	17	8	4	2	0	0	0
in	5	5	3	4	4	5	12	3
local	2	1	1	1	1	1	1	1
out	6	2	2	1	1	1	1	1

(a) RS1



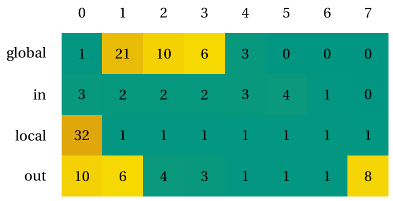
	0	1	2	3	4	5	6	7
global	0	21	13	8	4	0	0	0
in	3	3	4	9	5	4	0	0
local	1	1	1	2	1	1	1	0
out	5	4	1	0	1	5	0	2

(b) RS2



	0	1	2	3	4	5	6	7
global	12	18	9	5	3	0	0	0
in	5	5	3	4	4	5	10	2
local	2	1	1	1	1	1	1	1
out	6	3	1	1	1	1	1	1

(c) RS



	0	1	2	3	4	5	6	7
global	1	21	10	6	3	0	0	0
in	3	2	2	2	3	4	1	0
local	32	1	1	1	1	1	1	1
out	10	6	4	3	1	1	1	8

(d) RD

Figure 8.11: Evaluation results of the static register access analysis.

registers are presented. These outcomes illustrates the average register access distribution across all five benchmarks, with distinct listings for both *rs1* and *rs2* register ports. The cumulative read accesses are summarized as *rs*, while the write accesses via the *rd* port are listed.

The visualizations show a predominant focus on the *global* and *in* registers within the static program code.

8.3.2 Runtime Measurements of Register Accesses

Following the static program code analysis, the investigation delves into the determination of actual register accesses. As previously highlighted, the register accesses generated by the processor pipeline may deviate from those identified in static code analysis. Two primary factors contribute to this distinction. Firstly, control transfer instructions play a significant role, leading to the exclusion of certain program sections and the repetition of basic blocks. Secondly, the pipeline architecture of the original LEON3 pipeline, retained by the ACCP processors, introduces another layer of complexity. In order to mitigate the impact of RAW hazards, the pipeline architecture incorporates operand forward logic, resulting in not all register values being read from the register file and not accessing the associated port.

To quantify the actual register accesses, a statistical unit has been implemented. This unit is tasked with logging the accesses of a register. Similar to the static code analysis, it measures individual register accesses as well as simultaneous accesses of the three register file ports. The statistical unit monitors the interplay between the processor pipeline and the register file, featuring a performance counter for each register that increments upon access.

The initial assessment focuses on determining the practical feasibility of interleaved state transfer. It analyzes register port accesses generated by the program, without considering the specific target register. The efficacy of the interleaved processor state transfer mechanism heavily depends on identifying suitable time slots for uninterrupted register access. Specifically, the source core's register file requires a free *rs* port, and on the target side, both *rs* and *rd* ports must be available.

This evaluation is conducted across five benchmarks, collecting data on register file accesses while varying input sizes. The results, depicted in Figure 8.12, offer insights into the characteristics of register accesses. To ensure a comprehensive analysis, the arithmetic average across input sizes is calculated.

A noteworthy finding is the minimal impact of algorithm input size on the results, indicating consistent behavior of register accesses across diverse problem sizes. Furthermore, the analysis reveals that in approximately 20% of the time for each application, no ports of the regis-

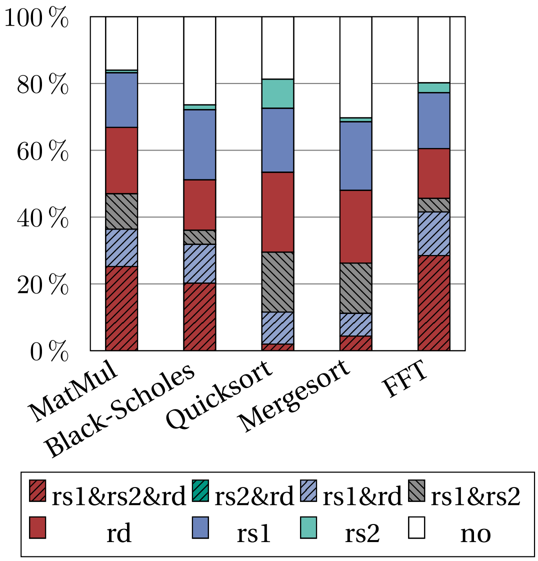
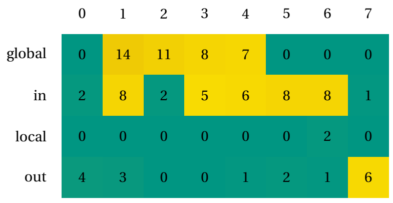


Figure 8.12: Evaluation results of register accesses. It shows how often register ports are accessed during the code execution. (refer to [Kem23a])

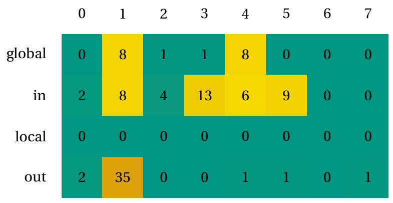
ter file are accessed. These idle time slots present opportunities for executing additional operations, such as reading or storing register data. Effectively leveraging these idle time slots is essential for maximizing the utilization of the interleaved state transfer process.

The second evaluation focuses on the registers that are actually accessed. In this assessment, the statistic unit translates the exact physical register address to the logical register. The SPARC v8 ISA incorporates a register window (refer to Section 2.1.1) that maps a logical window to a physical register in the register file. For instance, after a window slide, *i1* is mapped to the same physical register as the logical register *o1* was previously. The results regarding to the accessed reg-



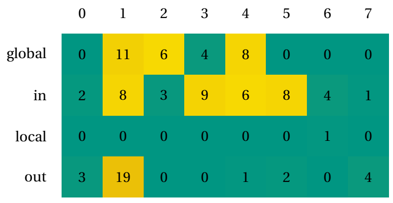
	0	1	2	3	4	5	6	7
global	0	14	11	8	7	0	0	0
in	2	8	2	5	6	8	8	1
local	0	0	0	0	0	0	2	0
out	4	3	0	0	1	2	1	6

(a) RS1



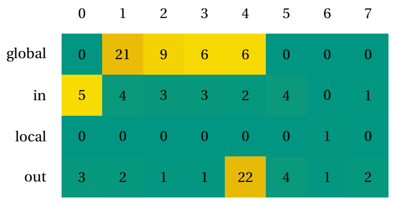
	0	1	2	3	4	5	6	7
global	0	8	1	1	8	0	0	0
in	2	8	4	13	6	9	0	0
local	0	0	0	0	0	0	0	0
out	2	35	0	0	1	1	0	1

(b) RS2



	0	1	2	3	4	5	6	7
global	0	11	6	4	8	0	0	0
in	2	8	3	9	6	8	4	1
local	0	0	0	0	0	0	1	0
out	3	19	0	0	1	2	0	4

(c) RS



	0	1	2	3	4	5	6	7
global	0	21	9	6	6	0	0	0
in	5	4	3	3	2	4	0	1
local	0	0	0	0	0	0	1	0
out	3	2	1	1	22	4	1	2

(d) RD

Figure 8.13: Evaluation results of the dynamic register access analysis.

isters are presented in Figure 8.13, representing an average across all relating register accesses in 27 benchmark runs.

The outcomes closely align with the results from static code analysis, with local register results seemingly approaching zero. However, this apparent zero value is attributed to rounding. The count of global register *g0* accesses is zero for the runtime measurement. That is due to the pipeline impletion, which uses a hardwired zero implementation.

8.4 Hardware Architecture for Interleaved State Transfers

The interleaved processor state transfer mechanism is embedded into the ACCP processor design of Chapter 7. A fundamental feature of the

ACCP concept is the generation of checkpoints, which are a snapshot of the current processor state.

8.4.1 The Processor Core Architecture

The ACCP processor pipeline has the capability to provide or take over control and state registers. These registers are implemented as FF and support parallel read and write operations without the need for additional specialized logic. However, in order to establish these state registers, the pipeline architecture needs synchronization to achieve an explicit and consistent state. This synchronization is critical because a pipeline processor executes instructions partially at each pipeline stage, thereby consistently changing the processor's state. Through synchronization, a well-defined state is reached, which is transferrable from the source core to the target core.

To enable this transfer, the source core and the target core creates a checkpoint before initiates the interleaved state transfer process. For the standard ACCP processor design, the creation of checkpoints on both cores is not mandatory to be done in advance of the state transfer. However, during the interleaved state transfer the values of a register are changeable by the executed program. The checkpoint ensure the data consistency of the transferable processor state.

The improved implementation of the processor core architecture empowers the capability for interleaved state transfer, as visually shown in Figure 8.14. The highlighted components and signals in red are used to realize the interleaved state transfer.

Within the architecture of the ACCP processor, the processor pipeline is connected to the *load unit* which forme a close interface with the register file. The *load unit* is responsible for the direct transmission of the pipeline's register accesses to the register file. Its primary function

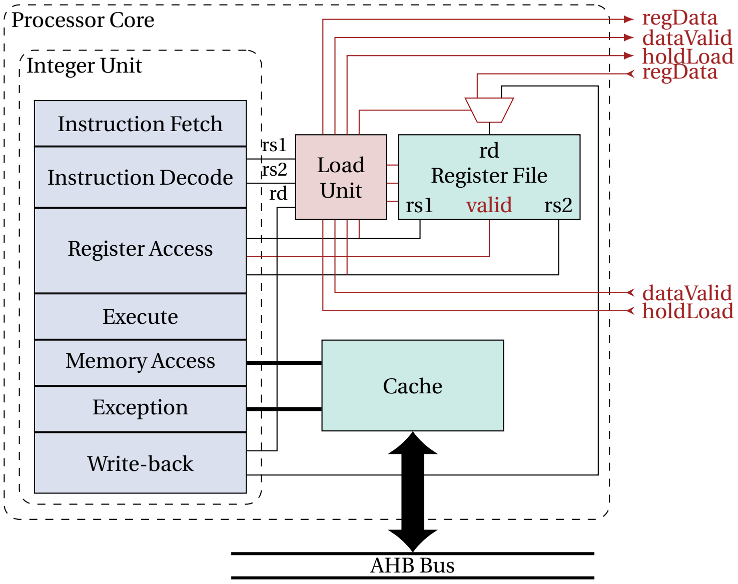


Figure 8.14: Processor core architecture for the interleaved processor state transfer. The red marked signals correspond to the logic realizing the state transfer mechanism. The load unit manages the interleaved access of the register file, while the valid signal indicates validity of the read register values. (refer to [Kem23a])

revolves around identifying opportune time slots within the software-generated register access stream, enabling the strategic scheduling of interleaved state transfers. When an appropriate time slot is available, the *load unit* seamlessly employs the read and write mechanisms as previously discussed to execute the transfer of the processor state.

The *load unit* not only serves to supply register data from the source core but also indicates the accessed data as valid or not through the use of the *dataValid* flag.

On the target core, the *dataValid* assumes the responsibility of receiving this data and effecting its transcription into the register file. However, for the register value to be accommodated, the register file must have an available time slot dedicated to data transfer. Consequently, in cases where the target core is currently incapable of assimilating the provided register state, it triggers the *holdLoad* signal. This signal is subsequently cleared as soon as a fitting time slot emerges on the target side. It's crucial to emphasize that the requirement for synchronous time slot availability on both the source and target cores is mitigated by the decoupling of time slots.

Additionally, the *load unit* incorporates a buffer designed to temporarily retain the loaded register value until such time as the *holdLoad* signal is cleared. This signal only returns to its default state when the *load unit* within the target core identifies a suitable time slot for the smooth transition of the register value.

An additional *valid* port signal has been added to extend the interface of the original general-purpose register file. This *valid* signal serves the purpose of indicating whether the specified read register is currently accessible and can be used for program execution. The *integer unit* monitors this signal closely and, if the required register value is not yet available, it temporarily stalls the pipeline. It remains in this stalled state until the *load unit* confirms the presence of a valid register value. The register value is considered valid as soon as the master provides it to the slave, and the slave successfully writes this value to its register file. Another way how a register value can be valid is when the executed program write the value to the register file.

This behavior of the register file is similar to a cache miss scenario, where the processor must wait until the cache retrieves data from a lower-level memory hierarchy. It is the master's responsibility to en-

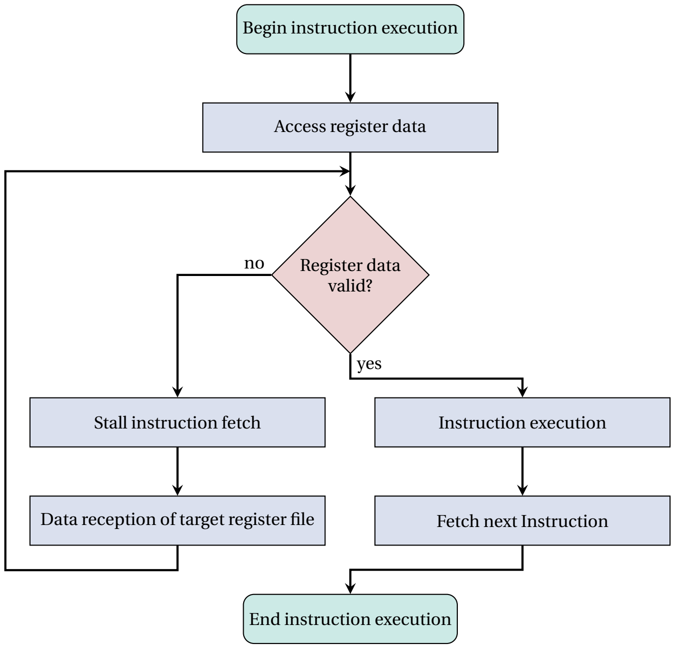


Figure 8.15: Flowchart illustrating the pipeline behavior fetching an register value during the interleaved state transfer.

sure that only unmodified, valid values are provided, with no modifications made since the creation of the checkpoint.

The provision of register data to the pipeline and its behavior for interleaved state transfer are illustrated in the flowchart in Figure 8.15. The pipeline exclusively executes valid register values to ensure correct behavior. Consequently, the slave pipeline halts until a valid register value is provided from the master's pipeline.

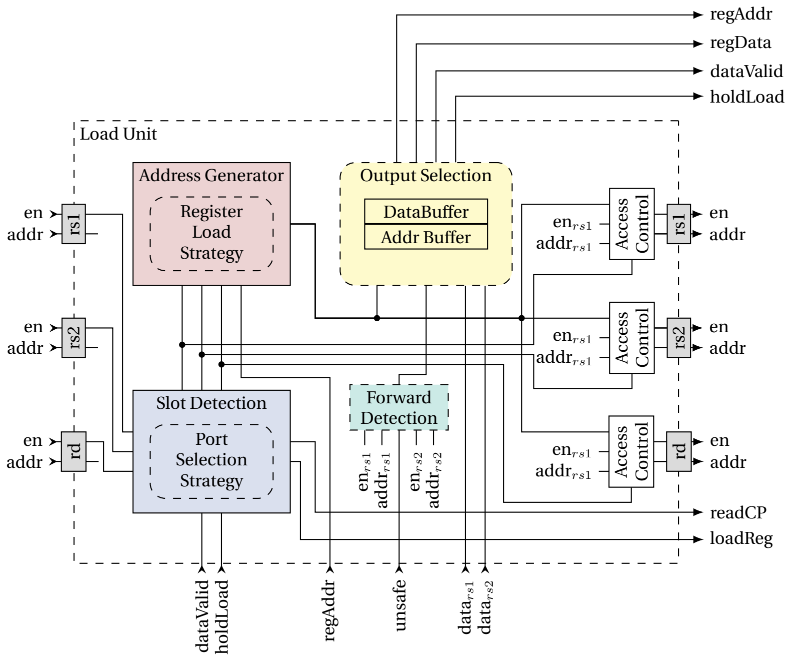


Figure 8.16: Architecture of the *load unit*. The slot detection finds suitable register access within of the register file access stream. The address generator provides the address of the next register to be transferred. The forward detection is optional and is used detects operands which can be directly forward from the source to the target core. (refer to [Kem23a])

8.4.2 The Load Unit Architecture

The *load unit* serves as a control unit for the interleaved state transfer between two ACCP processor cores. Positioned between the processor pipeline and the register cores, it manges the transfer of register data from the source core to the target core. In the source core, the *load* *unit* reads the register values from its associated *register file*, while in the target core, it writes the received register data to its respective *register file* register. The architecture of the *load unit* is illustrated in Figure 8.16.

Each *load unit* incorporates signals such as *regData* (representing register data), *regAddr* (denoting the address of the register), *dataValid* (indicating the validity of *regData*), and *holdLoad* (used for flow control). These signals have a essential role in transferring the source state to the target core and accepting the source state within the target core. Each of these signals have a source on the master core's *load unit* and their destination on the target core's *load unit*.

The *load unit* within the source core makes the register data from its associated *register file* available to the target core. The provide data of *regData* signal is valid by setting the *dataValid* signal. The *load unit* in the slave core, in turn, accepts this data and proceeds to write it into its own *register file*. For a transfer to occur, the register file in the target core requires an free time slot. Therefore, the target core signals its unavailability by raising the *holdLoad* signal when it cannot immediately receive the provided register state. This signal is only cleared when an appropriate time slot becomes accessible on the target side.

The identification of suitable time slots is the responsibility of the *slot detection* unit, which operates in accordance with the chosen register *port selection strategy*. This unit identifies time slot and configures the register file for writing and/or reading of register data. The *address generator* and the selected *register load strategy* collaborate to determine the subsequent register value to be replicated.

To overcome the constraint of necessitating concurrent available time slots on both cores, the time slots for the source and target cores are decoupled. This allows for a more flexible and asynchronous data transfer process between the two cores.

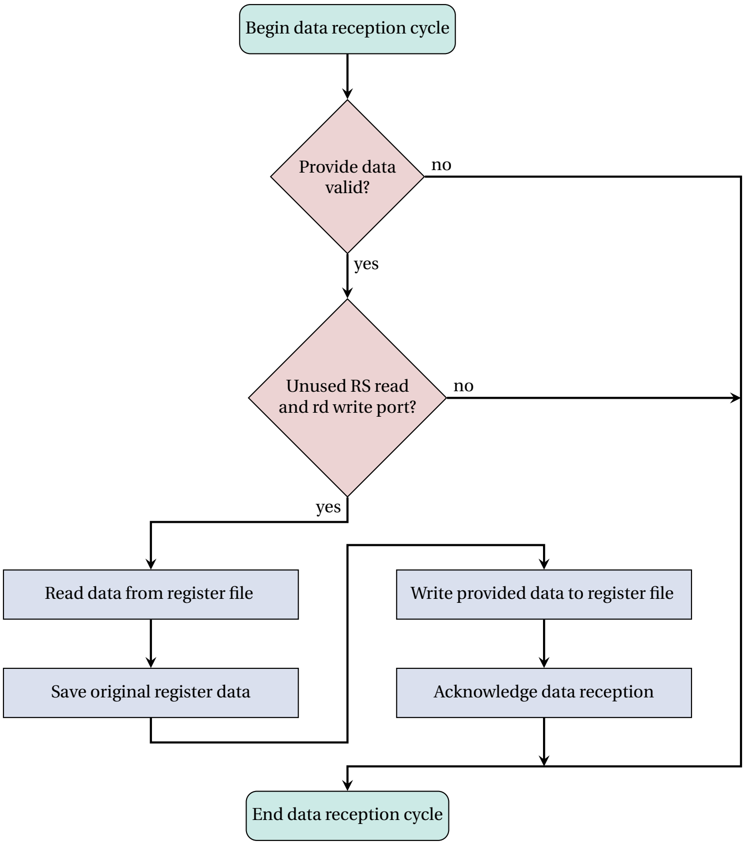

Begin data reception cycle
Provide data valid?
no
yes
Unused RS read and rd write port?
no
yes
Read data from register file
Save original register data
Write provided data to register file
Acknowledge data reception
End data reception cycle

Figure 8.17: Flowchart illustrating the data reception behavior behavior of the *load unit*.

The reception behavior of the *load unit* is illustrated in the flowchart presented in Figure 8.17. Data is exclusively written to the *register file* under two conditions: when it is valid and when there's a valid time slot for reading the current register value before writing the provided data. Upon acceptance of the data, an acknowledgment is issued.

Within the confines of the *output selection* unit, a buffer is in place with the purpose of transiently holding the loaded register value along with its corresponding address until the *holdLoad* signal is no longer active. As soon as the target core's *slot detection* unit identifies an appropriate time slot for receiving the register value, the *holdLoad* signal is deactivated, thereby enabling the transfer of the buffered register value.

Additionally, there is the possibility of utilizing an optional *forward detection* unit to identify operands that meet the criteria for direct forwarding from the target core to the source core. In cases where the target core accesses a register value from the checkpoint data (when *unsafe* is '0'), these operand values can be instantaneously forwarded to the target core. It is important to note that these forwarded operand values bypass the buffer within the *output selection* unit and are exclusively accepted by the target core when an appropriate time slot becomes available.

The data provisioning behavior of the *load unit* is depicted in the flowchart shown in Figure 8.18. Data is forwarded whenever a free read port of the *register file* is available, and the output buffer has space to store the register data. The output buffer holds the register data until acknowledgment of acceptance from the slave is received. Additionally, unchanged register data is forwarded to the slave whenever feasible. Unlike the read register data, the forwarded data, contrastingly, is not stored in the output buffer.

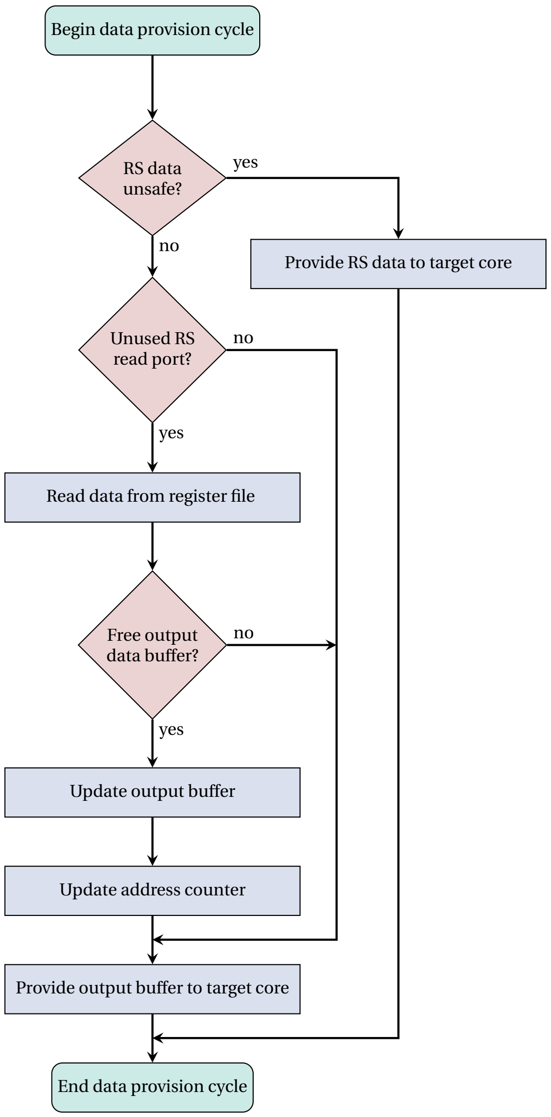


Figure 8.18: Flowchart illustrating the data provision behavior of the *load unit*.

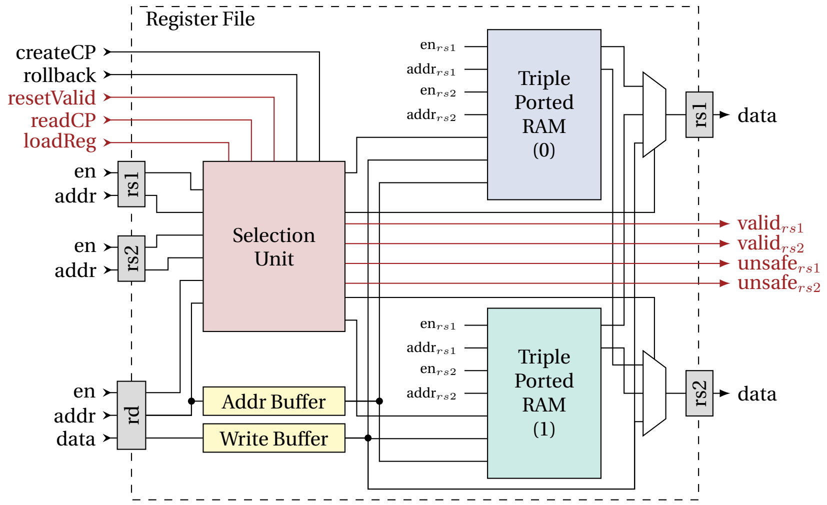


Figure 8.19: Checkpoint capable register file. The red highlighted signals are associated with the interleaved state transfer. Based on the input signals the *selection unit* chooses the correct memory for the access. The valid signals indicate whether the read values are legit. (refer to [Kem23a])

8.4.3 The Register File Architecture

The design of the register file is based on the architecture introduced for the ACCP (refer to Section 7.2). It uses as the base design two memory components for handling active and maintaining checkpoint data. The task of memory selection is controlled by the *selection unit*. Within this design, an existing *unsafe* register keeps a record of all registers that have undergone changes since the last checkpoint, and the *active* register points to the currently active memory location.

For read operations, the *active* register value dictates the target memory location, as it indicates the current active data location. When it comes to write accesses, both the *unsafe* and *active* registers collabo-

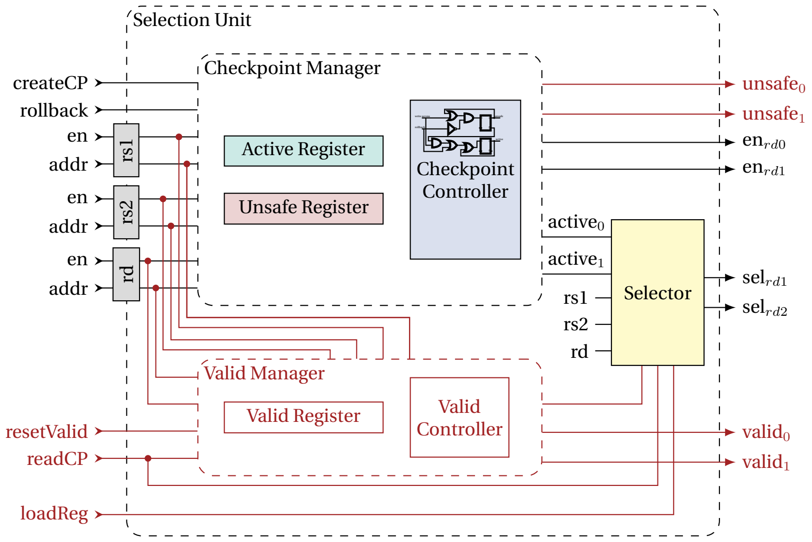


Figure 8.20: The architecture of *selection unit*. It manages the checkpoint separation from the active data and monitors the register validity. Red marked logic is associated to the interleaved register transfers. (refer to [Kem23a])

rate to determine the destination location, ensuring that checkpoint data remains untouched. The creation of a checkpoint occurs whenever the *createCP* signal is raised. Subsequently, the *unsafe* register monitors any changes. In the event of a request to roll back to the last checkpoint signaled by *rollback*, the changes tracked by the *unsafe* register are reversed.

In Figure 8.19, the fundamental structure of the register file is illustrated. It comprises three register ports: *rs1*, *rs2*, and *rd*, as well as the *selection unit* and the two memory units. Additionally, there are *addr* and *data write* buffers in place, necessary due to the pipeline architecture of the *selection unit*. This unit is responsible for managing *unsafe*, *active*, and *valid* and provides the memory data one cycle later.

For the interleaved state transfer, the *selection unit* is extended to include an additional *valid* register that monitors the validity of register values. This extended architecture is illustrated in Figure 8.20. Following a system reset, all *valid* registers are initially set to ’1’. At the beginning of the status replication, the *resetValid* signal is activated, resetting all *valid* register values to ’0’.

The *valid* register corresponding to the *addr* of *rd* is set to ’1’ whenever data is written. This can occur during the transfer of state values from the source to the target core, or when the program writes to the register. In the latter case, the data is transferred via program execution. If the state transfer process attempts to take over a register value that already possesses a legitimate register value (indicated by a *valid* register set to ’1’), the *selection unit* discard the write access. Therefore, the presence of a write related to the state transfer process is signaled by the *loadReg* signal.

The *valid* registers share similarities with the *unsafe* registers but are not entirely identical. The *unsafe* registers are reset every time a checkpoint is created, whereas the *valid* registers are cleared before any state is assumed. However, due to parallel code execution, a checkpoint can be created during the state transfer process.

For each of the *rs1* and *rs2* read ports, the *valid* signal reflects the state of the associated *valid* register. When a read port’s *valid* signal is ’0’, the processor pipeline halts until the state transfer process provides the required register value. As soon as the register value is written from the *rd* port, the *valid* signal of the register causing the stall transitions to ’1’. The design of the checkpointable register file abstracts access from the pipeline, but in the case of interleaved state transfer, it’s necessary to access the checkpoint even if active data is marked as *unsafe*. Consequently, the *readCP* signal indicates that the read transfer of the selected associated *rs* port is accessing checkpoint data.

Both cores create a checkpoint prior to the start of the state transfer. When the pipeline identifies a suitable time slot for an interleaved state transfer, it reads the checkpoint data from its register file. The target pipeline unloads the current state while accepting the state from the source core. For the state offloading, the pipeline reads the checkpoint data and simultaneously writes the takeover state. Both *readCP* and *loadReg* are activated whenever a register state is accepted by the target core. Reading checkpoint data always results in valid data, ensuring that the valid signal of the corresponding *rs* port is consistently active.

8.5 Strategies for Interleaved State Transfer

Regardless of the method employed to read from or write to the register file, it is necessary that the required register values are delivered in a timely manner. This requirement remains consistent whether accesses are made through additional register file ports or integrated within the sequence of register accesses initiated by program execution. Failure to provide a required register value to the target core in a timely manner leads to a pipeline stall, obstructing program execution. Progress cannot resume until the conflict is resolved, and the correct register value is accepted by the *load unit*. The moment the register value is successfully accepted, the pipeline can re-engage, ensuring the use of the accurate register value.

Consequently, the efficiency of the register selection process significantly influences the time required for transferring the processor state. As demonstrated earlier, the ideal scenario involves transferring all register values in advance, preempting the need for the target core to retrieve them. In this optimal situation, the additional time required for the complete state transfer would be reduced to zero. However, it's not always feasible to accurately predict the next operand that will be required.

The interleaved state transfer is explored through six distinct strategies, encompassing the selection of register values and read port allocation.

Strategy 1: This approach involves loading all register values from the source core using the *rs1* port statically. The slave, in turn, utilizes the *rs1* port for offloading and preserving the original register values. In the configuration with eight windows, this encompasses 136 register values loaded linearly, ranging from zero to 135, mirroring the order of the baseline implementation of the base ACCP architecture.

Strategy 2: Similar to Strategy 1, this strategy loads register values, but instead of the *rs1* port, it employs the *rs2* port.

Strategy 3: Utilizing the same linear register section order, this strategy dynamically selects one free *rs* port. The *load unit* observes both *rs* ports and utilizes the first available slot. Both the *load units* on the target and source cores employ this sectioning strategy.

Strategy 4: An extension of *Strategy 3*, this strategy incorporates operand forwarding. The *load unit* of the source core forwards the operands of the *rs1* port.

Strategy 5: This strategy adopts a different address generation scheme. Instead of the linear load from zero to 135, it prioritizes the current register window by using the source core's *CWP* value. Values from the current window are loaded with respect to previous findings. The read port selection employs the dynamic *rs* scheduler.

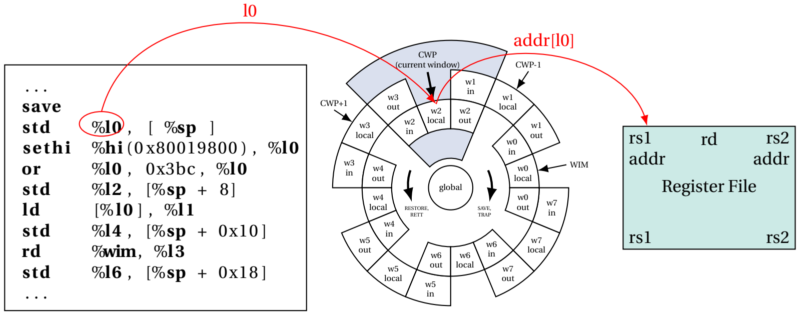
Strategy 6: Employing the same address scheme and register port selection methodology as *Strategy 5*, this strategy introduces source operand forwarding similar to *Strategy 4*.

The following should discuss the concept of operand forwarding employed by both *Strategy4* and *Strategy6.* Since both cores execute the same program, they generate identical register accesses. Leveraging this, the source core's register access information becomes instrumental in the register selection process. This process, termed operand forwarding in this thesis, involves forwarding an operand's value along with its associated register address from the source core to the slave core.

Operand forwarding capitalizes on the likelihood that the slave core either imminently requires the register value or has already stalled due to its absence. The *load unit* ensures that only unchanged register values are forwarded to the target core. Transmitting a value that has changed since the initiation of the state transfer process could lead to incorrect execution on the target core. To maintain data consistency, the target core safeguards against overwriting a register with an outdated state from the source core. The *load unit* plays a essential role in ensuring this data consistency by discarding incoming data if program execution on the target core has already written to the register.

Crucially, regardless of the use of operand forwarding, the load strategy and hardware implementation must guarantee the comprehensive copying of all register values from the source to the target register file. Failure to achieve this would result in the processor either stalling and aborting program execution or executing with flawed outcomes. This requirement remains independent of the utilization of operand forwarding.

The following discussion focuses on the address scheme based on the *Current Window Pointer (CWP)* of the PSR register. The active register window is determined by the current window pointer (*CWP*), stored in the status register. The actual accessed general-purpose registers depend on the *CWP* value. Alongside these variable registers, a set of



```
...
save
std     %l0 ,  [ %sp  ]
sethi   %hi(0x80019800), %l0
or      %l0 ,  0x3bc ,  %l0
std     %l2 ,  [%sp  +  8]
ld      [%l0 ] ,  %l1
std     %l4 ,  [%sp  +  0x10]
rd      %wim,  %l3
std     %l6 ,  [%sp  +  0x18]
...
```

Figure 8.21: The CWP points to the currently selected register window. The window highlighted in grey. All register accesses except the *global* registers are mapped to a register address dependent on the CWP. (refer to [Kem23a])

global registers, independent of *CWP*, is defined. Recognizing this distinction is essential for optimizing the register access analyzed earlier.

In Figure 8.21, the translation of an instruction operand into a register file address by the current window is demonstrated. The example illustrates how the *l0* operand is mapped to its address, with the *CWP* selecting the current window, as highlighted in the figure.

Registers not selected by the current register window cannot be directly accessed by software. The software must shift the current window before these registers become accessible. Consequently, these registers can be transferred with a lower priority.

Notably, *Strategy 5* and *Strategy 6* capitalize on this property. Additionally, these strategies leverage insights gained from the software-generated register accesses. Both the static and dynamic evaluations reveal an unevenly distributed register access pattern, indicating some registers are more likely to be accessed. An exact mapping of these results would entail a look-up based realization, which is hardware-

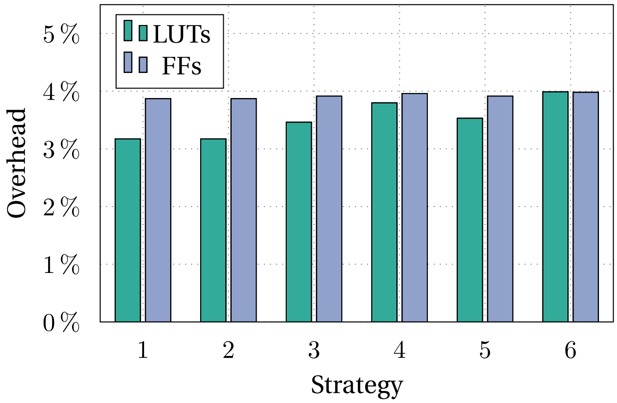


Figure 8.22: Hardware overhead of the processor cores of the different strategies compared to the baseline architecture. (refer to [Kem23a])

intensive. Therefore, the prioritization is initially given to the *global* registers and subsequently to the *in* registers. This strategic prioritization exploits the previously gained insights to enhance the performance of interleaved state transfer.

8.6 Experimental Evaluation

8.6.1 Hardware Resources

To conducted an evaluation of the hardware resource utilization the proposed hardware architecture is implementing it on an FPGA. The implementation targets AMD's Virtex UltraScale+ FPGA VCU118 using Vivado 2022.2. The hardware overhead introduced by the *load unit* and the *register file* when implementing the interleaved state replication strategies is shown in Figure 8.22. The overhead is compared to

Table 8.1: Hardware resources compression of the *load unit* and *register file* for the different transfer strategies. (refer to [Kem23a])

	Load Unit		Register File	
	LUTs	FFs	LUTs	FFs
Baseline	0	0	354	350
Strategy 1	119	33	651	492
Strategy 2	119	33	651	492
Strategy 3	157	35	651	492
Strategy 4	179	36	673	493
Strategy 5	166	35	651	492
Strategy 6	204	37	673	493

a baseline implementation without the interleaved state replication. The required additional hardware is relatively small, with overhead not exceeding 4% for both Lookup Tables (LUTs) and FFs.

The actual needed hardware resources are presented in Table 8.1. The table indicates the needed hardware resource consumption of the *load unit* and the *register file*. This provides a comprehensive understanding of the resource requirements. The more advanced Strategy 6, which includes operand forwarding and dynamic register file selection, requires slightly more logic compared to the simpler *Strategy 1*.

8.6.2 Transfer Times

Each strategy of the presented strategies is evaluated by running the algorithm and randomly initiating a state replication. To measure the effectiveness of each strategy, the duration of time that the target core had to stall during the replication process due to unavailable register values is measured. A total of 320 independent state transfers for each algorithm and each transfer strategy is performed. This amounts to a comprehensive evaluation of 1600 randomly executed task replications for each of the six strategies.

Figure 8.23 illustrates the evaluation results of the measured number of stall cycles during the state transfer for each individual algorithm. In particular, it provides the information on the minimum, maximum, and average number of cycles required for each algorithm. The solid line represents the baseline implementation of the ACCP architecture. This stalls the processor pipeline throughout the entire register file replication process. On the other hand, the dashed line represents the required transfer time of the currently active registers within the current register window, considering the *baseline* implementation. By analyzing these results, the impact of the state transfer process on the execution of different algorithms is observable.

The minimum number of cycles indicates the best-case scenario, where the transfer is completed quickly without causing significant stalls. The maximum number of cycles represents the worst-case scenario, where the transfer takes longer and introduces substantial delays. Finally, the average number of cycles provides a general overview of the typical state replication time.

The six transfer Strategies are summarized as following:

- **Baseline** Hold Processor until Load Finished
- **Strategy 1**: Linear load from *rs1*
- **Strategy 2**: Linear load from *rs2*
- **Strategy 3**: Linear load from dynamic *rs* selection
- **Strategy 4**: Linear load from dynamic *rs* selection with source
- **Strategy 5**: *CWP* related load from a dynamic *rs* selection
- **Strategy 6**: *CWP* related load from a dynamic *rs* selection with source operand forwarding.

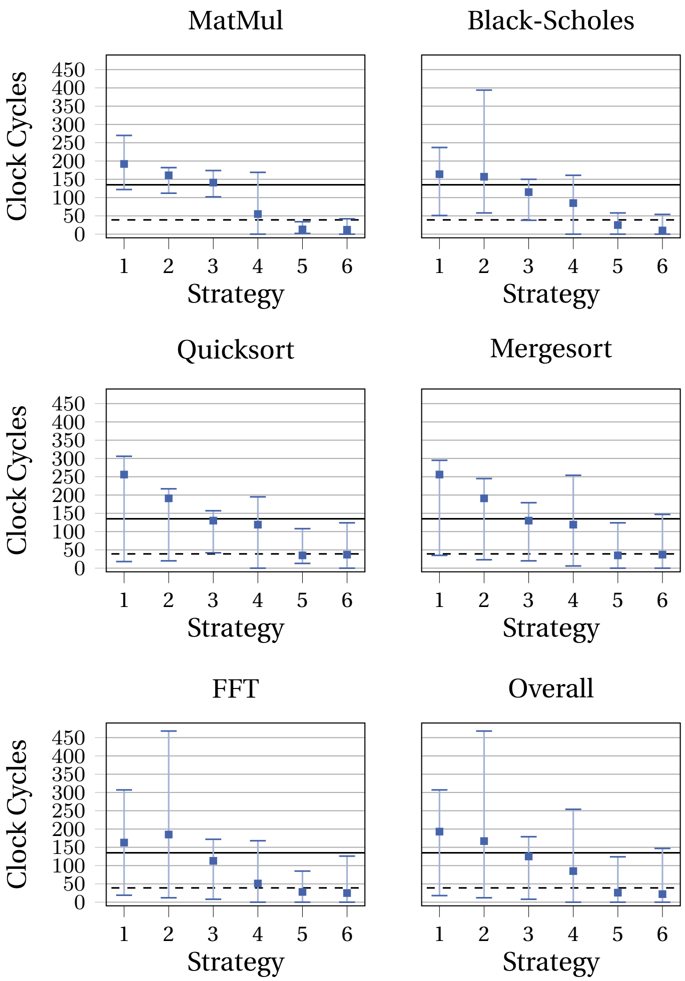


Figure 8.23: Evaluation of the maximum, minimum, and average number of stall clock cycles a transfer takes. The solid line indicates the baseline when both processor cores are held during full state transfer. The dashed line indicates the number of clock cycles required just to transfer the active registers using the baseline strategy. The overall plot summarizes the five benchmarks. (refer to [Kem23a])

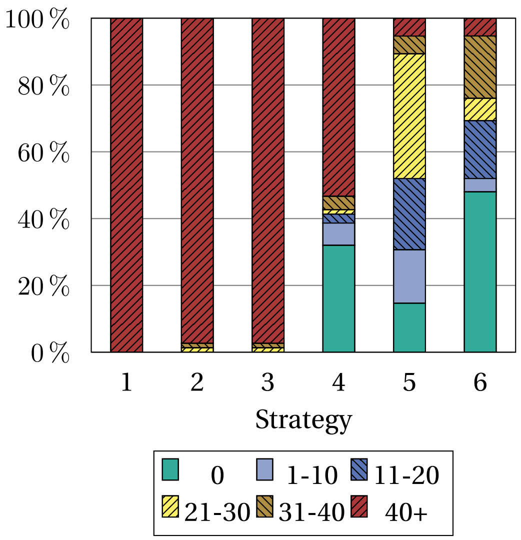


Figure 8.24: Distribution of clock cycles the state transfers requires for a specific strategy. (refer to [Kem23a])

The results of the evaluation indicate that a fixed read port selection strategy results in considerably higher stall times compared to the baseline implementation. Even with a dynamic read port selection, the average transfer time remains similar to the baseline. However, the inclusion of operand forwarding improves the average stall times, and in the best case scenario, the state transfer process does not interfere with program execution. On the other hand, in the worst case, it can lead to higher stall times than the baseline.

The best results is achieved with *Strategy 6*, which incorporates dynamic read port selection, operand forwarding, and register value prioritization.

Additionally, a deeper analysis of the distribution of required stall cycles for each strategy is performed. Instead of focusing only on maximum, minimum, and average times, the evaluation categorizes the expected stall cycles into six different classes. Of particular interest are the number of transfers with zero stalls or fewer stalls than the currently 32 active registers. As shown in Figure 8.24, the strategies that include operand forwarding (*Strategy 4* and *Strategy 6*) demonstrate a significant increase in the number of state transfers with zero stalls. This highlights the effectiveness of operand forwarding in reducing stall times and improving overall performance.

8.7 Summary

This chapter investigates a concept aimed at minimizing the duration of a complete register file state replication process. The initial focus involves an in-depth analysis of register access and various alternatives within the register file, encompassing diverse hardware implementations and the statistical behavior of software. Leveraging the existing hardware architecture, register accesses are strategically interleaved with regular program execution. The evaluation of different transfer strategies aims to compare replication behavior and the associated hardware resource requirements.

In the evaluation, each strategy show an improvement in the best-case scenario for the replication process. However, in the average and worst-case scenarios, the more advanced strategies demonstrate more significant enhancements. Strategies that take into account the software state and leverage typical register accesses demonstrate optimal performance. Especially, the forwarding of register data access from the source to the target contributes significantly to reducing the required stall time to zero.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

The advancement of semiconductor technology has driven single-core performance to a saturation point, where only marginal improvements are achievable. Consequently, the semiconductor industry has addressed these limitations by adopting multi-core architectures. However, Amdahl's Law emphasizes the limitations associated with parallelizing workloads. To harness the increasing transistor counts and processors effectively, multi-core processors are now employed not only for a single functionality but for a diverse range of tasks. This diversification, especially in embedded systems, has led to the concept of Mixed Criticality System (MCS), integrating functionalities with varying non-functional criticality requirements.

This work delves into an novel hardware-based Adaptive Fault Tolerance (AFT) methodology. It introduces the dynamic clustering approach of Adaptive Redundancy (AR), allowing processor cores to be redundantly grouped based on the executed functionality's functional safety demands.

This thesis demonstrates the application of Adaptive Redundancy (AR) to a multi-core system, presenting the Adaptive Lockstep Processor (ALP) architecture, which introduces a fine-grained lockstep design for both Dual-Core Lockstep (DCLS) (realizing DMR) and Triple-Core Lockstep (TCLS) (realizing TMR) configurations. The ALP efficiently manages state replication from a *master* to a *slave core*, incorporating fault detection mechanisms and a hardware-based fault mitigating technique. In the case of DCLS, transient faults are addressed through the re-execution of flawed program code, while TCLS utilizes a majority voting approach to mask SEU. Integration into a tile-based NoRMA processor design is successful, justified by a proof of concept involving the deployment of a complex dataflow RTE to compare software and hardware redundancy.

Additionally, the study explores a cores-grained redundancy architecture, the Adaptive Cache Checkpointing (ACCP), which leverages the cache to reduce the frequency of comparison and error checking. This shift from an instruction-based comparison, where each instruction result is assessed, to a comparison of processor core states, provides a loosely coupled and flexible approach. The comparison relies on a state signature derived by calculating a checksum over all system state changes.

Comparing this work with existing research, as detailed in Table 9.1, a selection of hardware architectures that are similar and comparable to this approaches are considered. This selection encompasses Commercial off-the-Shelf (COTS) architectures and academic solutions, all detailed in Chapter 3.

Among these, Arm's Dual-Core Lockstep (DCLS) and Triple-Core Lockstep (TCLS) stand out as non-academic architectures, sharing similar-

Table 9.1: Comparison of related work with the proposed architectures

	Reliability Method	**Fault Recovery**	**Split-Lock**	**Recovery Cycles**
This Work	**DMR**	**HW**	✔	**1**
	TMR	**HW**	✔	**0**
	ACCP	**HW**	✔	**1**
Rogenmoser et al. [118]	DCLS	SW	✔	-
		HW	✔	24
	TCLS	SW	✔	363
		HW	✔	24
Arm DCLS [105]	DCLS	SW	✔*	App. dep.
Arm TCLS [107]	TCLS	SW	✔*	2351
Shukla et al. [114]	DCLS	HW	✔	1
CEVERO [112]	DCLS	HW	✗	40
SHAKTI-F [91]	DCLS	HW	✗	3
DuckCore [90]	ECC	HW	✗	3

* requiers reboot

ities with proposed approaches of the presented work. Like the proposed approach, these architectures support reconfiguration between a performance *split mode* and a redundant *lock mode*. The nomenclature aligns with the naming scheme for the Adaptive Lockstep Processor (ALP). Notably, the transition between these modes requires a reboot and is orchestrated by the executing software. In configurations such as DMR or TMR lockstep clusters, fault detection is handled by a dedicated hardware unit. However, the responsibility for fault handling lies with the application software.

The concepts and processor architecture presented by Rogenmoser et al. [118] are based on a RISC-V processor and have similarities to the

Table 9.2: Reconfiguration time comparison of related work and the proposed architectures

	Reliability Method	ISA	Entry Cycles	Exit Cycles Master	Exit Cycles Slave
This Work	**DMR**	**SPARC 136 GPR**	**2-10**	**0**	**2-10**
	TMR		**2-10**	**0**	**2-10**
	ACCP		**137-145**	**0**	**137-145**
	ACCP (IST)*		**2-157**	**0**	**2-157**
Rogenmoser et al. [118]	DMR (SW)	RISC-V 32 GPR	534	22	147
	TMR (SW)		410	23	165
	DMR-R**		397	22	184
	TMR-R**		310	23	182

* Interleaved State Transfer (IST)
** Rapid Recovery (hardware-based)

Adaptive Lockstep Processor (ALP) outlined in this thesis. The features of the presented DCLS and TCLS architecture are comparable to the introduced and discussed Adaptive Lockstep Processor (ALP) architecture. Notably, both architectures can delegate fault recovery and reconfiguration processes to a dedicated hardware unit.

In Table 9.2, the reconfiguration times of the RISC-V processor, the ALP, and the ACCP architecture are compared. Regarding state transfer, it is essential to note that the RISC-V processor employs 32 General-Propose Registers (GPRs), whereas the ALP and the ACCP architecture utilize a register file with a total number of 136 GPRs. Across all scenarios, both the ALP and ACCP architectures exhibit lower reconfiguration times. This holds true for entering or exiting *locked* or *reliable mode*, with both architectures demonstrating fast and smooth reconfiguration processes.

The interleaved state transfer methodology is presented as an efficient means to significantly reduce the best case and average replication time. Notably, the worst-case transfer times measured in this study outperform related work approaches, even when replicating 4.25 times the registers. The presented architectures exhibit the lowest reaction times in fault handling and recovery compared to state-of-the-art approaches.

All investigated architectures and concepts are successfully implemented and realized as VHDL designs. FPGA prototypes validate the applicability of these designs, contributing to the enhancement of fault tolerance for MCS.

9.2 Future Work

This work primarily focuses into the applicability of AR as a realization of AFT, with a specific emphasis on multi-core architectures designed for Mixed Criticality System (MCS). The proposed ALP and ACCP architectures rely on a shared memory structure with a Uniform Memory Access (UMA). However, these architectures do not directly support processor architectures featuring NuMA or NoRMA. This investigation and research focused on a tile-based NoRMA architecture, limiting redundancy to a single tile due to the complex management of dynamic heap and stack memory. Future work may explore hardware support for hardware-based multi-tile redundancy, a concept that the ACCP architecture is designed to enable. However, additional research is needed to address the hardware-based state transfer of relevant instruction, stack and heap data.

Moreover, the concept of AR can be extended to AI accelerators. Our work on CNN workloads has revealed varying fault behaviors in individual convolutional layers, emphasizing the inherent redundancy within artificial neural networks. This inherent redundancy exhibits certain layers are more susceptible to faults than others, making AR a valuable approach to exploit these vulnerability levels and safeguard critical layers with hardware redundancy. An additional proposal, as presented in [Kem23c], introduces an on-demand time and spatial redundancy concept for AI accelerators with systolic array structures, positioning dependability improvements as a hardware/software co-design problem utilizing AR. Future research may focus on refining and expanding the concept of AR, mapping it to various problems involving mixed-critical calculations and challenges.

Abbreviation

ABS	Anti-lock Braking System
ACCP	Adaptive Cache Checkpointing
ACC	Adaptive Cruise Control
ADAS	Advanced Driver Assistance Systems
AD	Active Data
AFT	Adaptive Fault Tolerance
AHB	Advanced High-performance Bus
AI	Artificial Intelligence
ALP	Adaptive Lockstep Processor
ALU	Arithmetic Logic Unit
AL	Adaptive Lockstep
AMBA	Advanced Microcontroller Bus Architecture
AR	Adaptive Redundancy
ASIL	Automotive Safety Integrity Level
AXI	Advanced eXtensible Interface
BCC	Bare-C Cross Compilation System
BER	Backward Error Recovery
BSS	Block Start Symbol
BS	Black-Scholes
CALL	Call and Link
CCF	Common Cause Failures
CCP	Cache Checkpointing
CFC	Control Flow Checker
CFE	Control Flow Error

CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
CMOS	Complementary Metal-Oxide-Semiconductor
CNN	Convolutional Neural Network
COTS	Commercial off-the-Shelf
CPU	Central Processing Unit
CP	Checkpoint
CRC	Cyclic Redundancy Codes
CSR	Control and Status Register
CSR	Control and Status Register
CTI	Control-Transfer Instruction
CWP	Current Window Pointer
DAG	Directed Acyclic Graphs
DCLS	Dual-Core Lockstep
DCS	Dataflow and Control Signature
DCTI	Delayed Control-Transfer Instruction
DDR	Double Data Rate
DFE	Data-flow Error
DMA	Direct Memory Access
DMR	Dual Modular Redundancy
DRAM	Dynamic Random-Access Memory
DR	Dissable Reconfigration
DSU	Debug Support Unit
DUE	Detected Unrecoverable Error
ECC	Error-Correcting Code
ECU	Electronic Control Unit
EHP	Electron-Hole Pairs
ELF	Executable and Linkable Format
FDTI	Fault Detection Time Interval
FEC	Forward Error Correction
FER	Forward Error Recovery

FFT	Fast Fourier Transform
FF	Flip-Flop
FHTI	Fault Handling Time Interval
FIFO	First in First out
FIT	Failure in Time
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FP	Fram Pointer
FRTI	Fault Reaction Time Interval
FSM	Finite State Machine
FTTI	Fault Tolerant Time Interval
GDB	GNU Debugger
GEMM	General Matrix Multiply
GPR	General-Propose Register
GS	Guaranteed Service
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
HLS	High-Level Synthesis
HPC	High-Performance Computing
HTM	Hardware Transactional Memory
HW	Hardware
ICC	Integer Condition Code
ILP	Instruction-level Parallelism
IP	Intellectual Property
IRQMP	Multi-Processor Interrupt Controller
ISA	Instruction Set Architecture
IU	Integer Unit
JMPL	Jump and Link
LIFO	Last in First out
LRU	Least Recently Used
LUT	Lookup Table

MatMul	Matrix Multiplication
MCS	Mixed Criticality System
MPI	Message Passing Interface
MPPA	Massively Parallel Processor Array
MS	Mergesort
MTBF	Mean Time between Failure
MTTF	Mean Time to Failure
MTTR	Mean Time to Repair
NA	Network Adapter
NBTI	Negative Bias Temperature Instability
NI	Network Interface
NI	Network Interface
NMR	N-Modular Redundancy
NoC	Network-on-Chip
NOP	No Operation
NoRMA	No Remote Memory Access
NPC	Next Program Counter
NuMA	Non-Uniform Memory Access
NVM	Non-Volatile Memory
OCP	Open Core Protocol
PC	Program Counter
PSR	Processor State Register
QM	Quality Management
QoS	Quality of Service
QS	Quicksort
RAM	Random-Access Memory
RAW	Read-After-Write
RBCP	Rollback Program Counter
RETT	Return from Trap
RISC	Reduced Instruction Set Computer
RMU	Redundancy Management Unit

RTE	Runtime Environment
RTL	Register-Transfer Level
SAE	Scociety of Automotive Engineers
SDC	Silent Data Corruption
SEE	Single Event Effect
SEL	Single Event Latch
SET	Single Event Transient
SEU	Single Event Upset
SFT	Static Fault Tolerance
SHS	State History Signature
SIL	Safety Integrity Level
SoR	Sphere of Replication
SPARC	Scalable Processor Architecture
SP	Stack Pointer
SRAM	Static Random-Access Memory
SW	Software
TAWS	Terrain Avoidance and Warning System
TBR	Trap Base Register
TBR	Trap Base Register
TCLS	Triple-Core Lockstep
TID	Total Ionizing Dose
TLM	Tile Local Memory
TMR	Triple Modular Redundancy
TM	Transactional Memory
UART	Universal Asynchronous Receiver Transmitte
UMA	Uniform Memory Access
VC	Virtual Channel
VHDL	VHSIC (Very High Speed Integrated Circuits) Hardware Description Language
VLIW	Very Long Instruction Word
WCET	Worst-Case Execution Time

WCTT	Worst-Case Transmission Time
WIM	Window Invalid Mask
Y	Multiply/Divide Register

List of Figures

List of Tables

Listings

Bibliography

[1] Gordon E Moore. “Cramming more components onto integrated circuits”. In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[2] Anantha P Chandrakasan, Samuel Sheng, and Robert W Brodersen. “Low-power CMOS digital design”. In: *IEICE Transactions on Electronics* 75.4 (1992), pp. 371–382.

[3] Gary K Yeap. Practical low power digital VLSI design. Springer Science & Business Media, 2012.

[4] John L Hennessy and David A Patterson. Computer architecture: a quantitative approach. Morgan Kaufmann, 2011.

[5] Karl Rupp. Microprocessor Trend Data. (Online; accessed 27-November-2023). 2023. url: `https://github.com/karlrupp/microprocessor-trend-data`.

[6] Shekhar Borkar. “Thousand core chips: a technology perspective”. In: *Proceedings of the 44th annual design automation conference*. 2007, pp. 746–749.

[7] Gene M Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[8] Alan Burns and Robert Davis. “Mixed criticality systems-a review”. In: *Department of Computer Science, University of York, Tech. Rep* (2013).

[9] On-Road Automated Driving (ORAD) Committee. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. SAE International, 2021.

[10] Stanley Bak, Deepti K Chivukula, Olugbemiga Adekunle, Mu Sun, Marco Caccamo, and Lui Sha. “The system-level simplex architecture for improved real-time embedded system safety”. In: *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2009, pp. 99–107.

[11] Tobias Dörr, Timo Sandmann, Florian Schade, Falco K Bapp, and Jürgen Becker. “Leveraging the partial reconfiguration capability of FPGAs for processor-based fail-operational systems”. In: *Applied Reconfigurable Computing: 15th International Symposium, ARC 2019, Darmstadt, Germany, April 9–11, 2019, Proceedings 15*. Springer. 2019, pp. 96–111.

[12] Florian Oszwald, Philipp Obergfell, Matthias Traub, and Juergen Becker. “Reliable fail-operational automotive e/e-architectures by dynamic redundancy and reconfiguration”. In: *2019 32nd IEEE International System-on-Chip Conference (SOCC)*. IEEE. 2019, pp. 203–208.

[13] Philipp Weiss, Ali Younessi, and Sebastian Steinhorst. “Reliability Analysis of Gracefully Degrading Automotive Systems”. In: *arXiv preprint arXiv:2305.07401* (2023).

[14] Michael Zimmer, David Broman, Chris Shaver, and Edward A Lee. “FlexPRET: A processor platform for mixed-criticality systems”. In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2014.

[15] Zheng Li, Xiayu Hua, Chunhui Guo, and Shangping Ren. “Empirical study of energy minimization issues for mixed-criticality systems with reliability constraint”. In: *Proc. LPDC*. 2014.

[16] Behnaz Ranjbar, Bardia Safaei, Alireza Ejlali, and Akash Kumar. “FANTOM: Fault tolerant task-drop aware scheduling for mixed-criticality systems”. In: *IEEE access* 8 (2020), pp. 187232–187248.

[17] Siva Satyendra Sahoo, Behnaz Ranjbar, and Akash Kumar. “Reliability-aware resource management in multi- / many-core systems: A perspective paper”. In: *Journal of Low Power Electronics and Applications* 11.1 (2021), p. 7.

[18] Frontgrade Gaisler. GRLIB IP Core User’s Manual. 2023.

[19] SPARC International. The SPARC Architecture Manual. Version 8.

[20] D Haris and S Haris. Digital Design and Computer Architecture ARM Edition. 2016.

[21] Shubu Mukherjee. Architecture design for soft errors. Morgan Kaufmann, 2011.

[22] 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* (2004).

[23] Wei Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M.R. Stan. “HotSpot: a compact thermal modeling methodology for early-stage VLSI design”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2006).

[24] M. Pedram and S. Nazarian. “Thermal Modeling, Analysis, and Management in VLSI Circuits: Principles and Methods”. In: *Proceedings of the IEEE* (2006).

[25] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. “Temperature-Aware Microarchitecture: Modeling and Implementation”. In: *ACM Trans. Archit. Code Optim.* 1.1 (2004). doi: 10.1145/980152.980157.

[26] R. Baumann. “Soft errors in advanced computer systems”. In: *IEEE Design & Test of Computers* 22.3 (2005).

[27] Jörg Henkel and Nikil Dutt. Dependable Embedded Systems. Springer, Jan. 2021. doi: 10.1007/978-3-030-52017-5.

[28] P.E. Dodd, M.R. Shaneyfelt, J.A. Felix, and J.R. Schwank. “Production and propagation of single-event transients in high-speed digital logic ICs”. In: *IEEE Transactions on Nuclear Science* 51.6 (2004), pp. 3278–3284. doi: 10.1109/TNS.2004.839172.

[29] Masanori Hashimoto, Kazutoshi Kobayashi, Jun Furuta, Shin-Ichiro Abe, and Yukinobu Watanabe. “Characterizing SRAM and FF soft error rates with measurement and simulation”. In: *Integration* 69 (2019), pp. 161–179. doi: https://doi.org/10.1016/j.vlsi.2019.03.005. url: https://www.sciencedirect.com/science/article/pii/S0167926018305613.

[30] Eishi Ibe. “Terrestrial Radiation Effects in ULSI Devices and Electronic Systems”. In: *Terrestrial Radiation Effects in ULSI Devices and Electronic Systems* (Jan. 2015), pp. 1–268. doi: 10.1002/9781118479308.

[31] Véronique Ferlet-Cavrois, Lloyd W. Massengill, and Pascale Gouker. “Single Event Transients in Digital CMOS—A Review”. In: *IEEE Transactions on Nuclear Science* 60.3 (2013), pp. 1767–1790. doi: 10.1109/TNS.2013.2255624.

[32] Israel KOREN and Zahava KOREN. “Defect tolerance in VLSI circuits: techniques and yield analysis”. In: *Proceedings of the IEEE* 86.9 (1998), pp. 1819–1838.

[33] Laung-Terng WANG, Cheng-Wen WU, and Xiaoqing WEN. VLSI test principles and architectures: design for testability. Elsevier, 2006.

[34] T.R. OLDHAM and F.B. MCLEAN. “Total ionizing dose effects in MOS oxides and devices”. In: *IEEE Transactions on Nuclear Science* (2003).

[35] H. J. BARNABY. “Total-Ionizing-Dose Effects in Modern CMOS Technologies”. In: *IEEE Transactions on Nuclear Science* 53.6 (2006), pp. 3103–3121. DOI: `10.1109/TNS.2006.885952`.

[36] Mridul AGARWAL, Bipul C. PAUL, Ming ZHANG, and Subhasish MITRA. “Circuit Failure Prediction and Its Application to Transistor Aging”. In: *25th IEEE VLSI Test Symposium (VTS’07)*. 2007.

[37] Jens LIENIG and Hans BRUEMMER. Fundamentals of electronic systems design. Springer, 2017.

[38] William Wesley PETERSON and Edward J WELDON. Error-correcting codes. MIT press, 1972.

[39] Maha KOOLI and Giorgio DI NATALE. “A survey on simulation-based fault injection tools for complex systems”. In: *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE. 2014.

[40] Carles HERNANDEZ and Jaume ABELLA. “Live: Timely error detection in light-lockstep safety critical systems”. In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2014.

[41] Steven K Reinhardt and Shubhendu S Mukherjee. "Transient fault detection via simultaneous multithreading". In: *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No. RS00201)*. 2000.

[42] NXP Semiconductors. MPC5777M Reference Manual. English. 2016.

[43] Infineon Technologies AG. AURIX 32-bit microcontrollers for automotive and industrial applicationsHighly integrated and performance optimized. English. 2020.

[44] IBM. PowerPC 750GX Lockstep Facility, Application note. English. 2008.

[45] Freescale Semiconductor. Qorivva MPC5643L Microcontroller Data Sheet. English. 2013.

[46] Radu Teodorescu, Jun Nakano, and Josep Torrellas. "SWICH: A prototype for efficient cache-level checkpointing and rollback". In: *IEEE Micro* 26.5 (2006), pp. 28–40.

[47] O. Gonzalez, H. Shrikumar, J.A. Stankovic, and K. Ramamritham. "Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling". In: *Proceedings Real-Time Systems Symposium*. 1997.

[48] K.H. Kim and T.F. Lawrence. "Adaptive fault tolerance: issues and approaches". In: *[1990] Proceedings. Second IEEE Workshop on Future Trends of Distributed Computing Systems*. 1990.

[49] KH (Kane) Kim and Thomas F Lawrence. "Adaptive fault-tolerance in complex real-time distributed computer system applications". In: *Computer Communications* 15.4 (1992). Software aspects of future trends in distributed systems, pp. 243–251. doi: https://doi.org/10.1016/0140-3664(92)90107-P. url: https://www.sciencedirect.com/science/article/pii/014036649290107P.

[50] J. GOLDBERG, I. GREENBERG, and T.F. LAWRENCE. “Adaptive fault tolerance”. In: *Proceedings 1993 IEEE Workshop on Advances in Parallel and Distributed Systems*. 1993.

[51] INTERNATIONAL STANDARD. IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems. English. 2010.

[52] INTERNATIONAL STANDARD. ISO 26262-1: Road vehicles — Functional safety — Vocabulary. English. 2018.

[53] INTERNATIONAL STANDARD. ISO 26262-3: Road vehicles — Functional safety — Concept phase. English. 2018.

[54] INTERNATIONAL STANDARD. ISO 26262-5: Road vehicles — Functional safety — Product development at the hardware level. English. 2018.

[55] Iraj MOGHADDASI, Saeid GORGIN, and Jeong-A LEE. “Dependable DNN Accelerator for Safety-critical Systems: A Review on the Aging Perspective”. In: *IEEE Access* (2023).

[56] Reinhard WILHELM et al. “The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools”. In: *ACM Trans. Embed. Comput. Syst.* 7.3 (2008).

[57] Sepideh SAFARI, Mohsen ANSARI, Heba KHDR, Pourya GOHARI-NAZARI, Sina YARI-KARIN, Amir YEGANEH-KHAKSAR, Shaahin HESSABI, Alireza EJLALI, and Jörg HENKEL. “A survey of fault-tolerance techniques for embedded systems from the perspective of power, energy, and thermal issues”. In: *IEEE Access* 10 (2022), pp. 12229–12251.

[58] Dimitris GIZOPOULOS, Mihalis PSARAKIS, Sarita V ADVE, Pradeep RAMACHANDRAN, Siva Kumar Sastry HARI, Daniel SORIN, Albert MEIXNER, Arijit BISWAS, and Xavier VERA. “Architectures for online error detection and recovery in multicore processors”. In: *2011 Design, Automation & Test in Europe*. IEEE. 2011, pp. 1–6.

[59] Shuguang FENG, Shantanu GUPTA, Amin ANSARI, and Scott MAHLKE. “Shoestring: probabilistic soft error reliability on the cheap”. In: *ACM SIGARCH Computer Architecture News* 38.1 (2010), pp. 385–396.

[60] Marco OTTAVI, Salvatore PONTARELLI, Dimitris GIZOPOULOS, Cristiana BOLCHINI, Maria K MICHAEL, Lorena ANGHEL, Mehdi TAHOORI, Antonis PASCHALIS, Pedro REVIRIEGO, Oliver BRINGMANN, et al. “Dependable multicore architectures at nanoscale: The view from europe”. In: *IEEE Design & Test* 32.2 (2014), pp. 17–28.

[61] Salvatore PONTARELLI, Juan A MAESTRO, and Pedro REVIRIEGO. “Dependability Solutions”. In: *Dependable Multicore Architectures at Nanoscale* (2018), pp. 155–188.

[62] George A REIS, Jonathan CHANG, Neil VACHHARAJANI, Ram RANGAN, and David I AUGUST. “SWIFT: Software implemented fault tolerance”. In: *International symposium on Code generation and optimization*. IEEE. 2005, pp. 243–254.

[63] J. CHANG, G.A. REIS, and D.I. AUGUST. “Automatic Instruction-Level Software-Only Recovery”. In: *International Conference on Dependable Systems and Networks (DSN’06)*. 2006, pp. 83–92. DOI: `10.1109/DSN.2006.15`.

[64] Christoph KÜHBACHER, Theo UNGERER, and Sebastian ALTMEYER. “Redundant dataflow applications on clustered many-core architectures”. In: *SAC ’22: proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, virtual event, April 25 - 29, 2022*. 2022, pp. 226–235. DOI: `10.1145/3477314.3507272`.

[65] Falco K Bapp. “Adaptives Monitoring für Mehrkernprozessoren in eingebetteten sicherheitskritischen Systemen”. PhD thesis. Dissertation, Karlsruhe, Karlsruher Institut für Technologie (KIT), 2017, 2018.

[66] Hamidreza Ahmadian, Roman Obermaisser, and Jon Perez. Distributed Real-Time Architecture for Mixed-Criticality Systems. CRC Press, 2018.

[67] Zonghua Gu and Qingling Zhao. “A state-of-the-art survey on real-time issues in embedded systems virtualization”. In: (2012).

[68] Salvador Trujillo, Alfons Crespo, Alejandro Alonso, and Jon Pérez. “MultiPARTES: Multi-core partitioning and virtualization for easing the certification of mixed-criticality systems”. In: *Microprocessors and Microsystems* 38.8 (2014), pp. 921–932.

[69] Irune Agirre, Mikel Azkarate-Askasua, Asier Larrucea, Jon Perez, Tullio Vardanega, and Francisco J Cazorla. “A safety concept for a railway mixed-criticality embedded system based on multicore partitioning”. In: *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*. IEEE. 2015, pp. 1780–1787.

[70] Irune Agirre, Mikel Azkarate-askasua, Asier Larrucea, Jon Perez, Tullio Vardanega, and Francisco J Cazorla. “Automotive safety concept definition for mixed-criticality integration on a COTS multicore”. In: *Computer Safety, Reliability, and Security: SAFECOMP 2016 Workshops, ASSURE, DECSoS, SASSUR, and TIPS, Trondheim, Norway, September 20, 2016, Proceedings 35*. Springer. 2016, pp. 273–285.

[71] Jon PEREZ, David GONZALEZ, Carlos Fernando NICOLAS, Ton TRAPMAN, and Jose Miguel GARATE. “A safety certification strategy for IEC-61508 compliant industrial mixed-criticality systems based on multicore partitioning”. In: *2014 17th Euromicro Conference on Digital System Design*. IEEE. 2014, pp. 394–400.

[72] Benôıt Dupont de DINECHIN et al. “A clustered manycore processor architecture for embedded and accelerated applications”. In: *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. 2013, pp. 1–6. DOI: `10.1109/HPEC.2013.6670342`.

[73] Thierry GOUBIER, Renaud SIRDEY, Stéphane LOUISE, and Vincent DAVID. “ΣC: A programming model and language for embedded manycores”. In: *Algorithms and Architectures for Parallel Processing: 11th International Conference, ICA3PP, Melbourne, Australia, October 24-26, 2011, Proceedings, Part I 11*. Springer. 2011, pp. 385–394.

[74] Tiago AO ALVES, Sandip KUNDU, Leandro AJ MARZULO, and Felipe MG FRANÇA. “Online error detection and recovery in dataflow execution”. In: *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*. IEEE. 2014, pp. 9–104.

[75] Samir JAFAR, Thierry GAUTIER, Axel KRINGS, and Jean-Louis ROCH. “A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing”. In: *European Conference on Parallel Processing*. Springer. 2005, pp. 675–684.

[76] Federico REGHENZANI, Zhishan GUO, and William FORNACIARI. “Software fault tolerance in real-time systems: identifying the future research questions”. In: *ACM Computing Surveys* (2023).

[77] Jonah CAPLAN, Zaid AL-BAYATI, Haibo ZENG, and Brett H. MEYER. “Mapping and Scheduling Mixed-Criticality Systems with On-

Demand Redundancy”. In: *IEEE Transactions on Computers* (2018).

[78] Mohammad SALEHI, Alireza EJLALI, and Bashir M. AL-HASHIMI. “Two-Phase Low-Energy N-Modular Redundancy for Hard Real-Time Multi-Core Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.5 (2016), pp. 1497–1510. DOI: 10.1109/TPDS.2015.2444402.

[79] Rami MELHEM, Daniel MOSSE, and Elmootazbellah ELNOZAHY. “The interplay of power management and fault recovery in real-time systems”. In: *IEEE Transactions on Computers* 53.2 (2004), pp. 217–231.

[80] Daniel MOSSE, Rami MELHEM, and Sunondo GHOSH. “Analysis of a fault-tolerant multiprocessor scheduling algorithm”. In: *Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing*. IEEE. 1994, pp. 16–25.

[81] Sandra RAMOS-THUEL and Jay K STROSNIDER. “Scheduling fault recovery operations for time-critical applications”. In: *Dependable Computing for Critical Applications 4*. Springer. 1995, pp. 411–432.

[82] Jon Perez CERROLAZA, Roman OBERMAISSER, Jaume ABELLA, Francisco J CAZORLA, Kim GRÜTTNER, Irune AGIRRE, Hamidreza AHMADIAN, and Imanol ALLENDE. “Multi-core devices for safety-critical systems: A survey”. In: *ACM Computing Surveys (CSUR)* 53.4 (2020), pp. 1–38.

[83] M KRSTIC, M ANDJELKOVIC, O SCHRAPE, A BREITENREITER, J CHEN, A BALASHOV, and A SIMEVSKI. “Cross-Layer Digital Design Flow for Space Applications”. In: *2021 IEEE 32nd International Conference on Microelectronics (MIEL)*. IEEE. 2021, pp. 45–54.

[84] Jörg HENKEL, Lars BAUER, Nikil DUTT, Puneet GUPTA, Sani NASSIF, Muhammad SHAFIQUE, Mehdi TAHOORI, and Norbert WEHN. "Reliable on-chip systems in the nano-era: Lessons learnt and future trends". In: *Proceedings of the 50th Annual Design Automation Conference*. 2013, pp. 1–10.

[85] Hai YU. "Low-cost highly-efficient fault tolerant processor design for mitigating the reliability issues in nanometric technologies". PhD thesis. Université de Grenoble, 2011.

[86] Ikhwan LEE, Mehmet BASOGLU, Michael SULLIVAN, Doe Hyun YOON, Larry KAPLAN, and Mattan EREZ. "Survey of error and fault detection mechanisms". In: *University of Texas at Austin, Tech. Rep* 11 (2011), p. 12.

[87] XILINX. Xilinx TMRTool User Guide TMRTool Software, UG156. 2017.

[88] XILINX. Soft Error Mitigation Controller v4.1 LogiCORE IP Product Guide, PG036. 2018.

[89] Alexander WALSEMANN, Michael KARAGOUNIS, Alexander STANITZKI, and Dietmar TUTSCH. "STRV — a radiation hard RISC-V microprocessor for high-energy physics applications". In: *Journal of Instrumentation* 18.02 (2023), p. C02032.

[90] Jiemin LI, Shancong ZHANG, and Chong BAO. "DuckCore: a fault-tolerant processor core architecture Based on the RISC-V ISA". In: *Electronics* 11.1 (2021), p. 122.

[91] Sukrat GUPTA, Neel GALA, GS MADHUSUDAN, and V KAMAKOTI. "SHAKTI-F: A fault tolerant microprocessor architecture". In: *2015 IEEE 24th Asian Test Symposium (ATS)*. IEEE. 2015, pp. 163–168.

[92] Salvatore PONTARELLI, Juan A MAESTRO, and Pedro REVIRIEGO. "Dependability Solutions". In: *Dependable Multicore Architectures at Nanoscale* (2018), pp. 155–188.

[93] Todd M Austin. "DIVA: A reliable substrate for deep submicron microarchitecture design". In: *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture.* IEEE. 1999, pp. 196–207.

[94] Abdelmajid Bouajila, Thomas Sommer, Johannes Zeppenfeld, Walter Stechele, and Andreas Herkersdorf. "A Fault-Tolerant Processor Architecture". In: *22th International Conference on Architecture of Computing Systems 2009.* 2009, pp. 1–6.

[95] Albert Meixner, Michael E Bauer, and Daniel Sorin. "Argus: Low-cost, comprehensive error detection in simple cores". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007).* IEEE. 2007, pp. 210–222.

[96] Stephan Wong, Thijs van As, and Geoffrey Brown. "ρ-VEX: A reconfigurable and extensible softcore VLIW processor". In: *2008 International Conference on Field-Programmable Technology.* 2008.

[97] Anderson L Sartor, Stephan Wong, and Antonio CS Beck. "Adaptive ILP control to increase fault tolerance for VLIW processors". In: *2016 IEEE 27th International Conference on Applicationspecific Systems, Architectures and Processors (ASAP).* IEEE. 2016, pp. 9–16.

[98] Fakhar Anjam and Stephan Wong. "Configurable Fault-Tolerance for a Configurable VLIW Processor". In: *Reconfigurable Computing: Architectures, Tools and Applications.* Springer Berlin Heidelberg, 2013, pp. 167–178.

[99] Anderson L Sartor, Arthur F Lorenzon, Luigi Carro, Fernanda Kastensmidt, Stephan Wong, and Antonio CS Beck. "A novel phase-based low overhead fault tolerance approach for VLIW processors”. In: *2015 IEEE Computer Society Annual Symposium on VLSI*. IEEE. 2015, pp. 485–490.

[100] Anderson L Sartor, Arthur F Lorenzon, Luigi Carro, Fernanda Kastensmidt, Stephan Wong, and Antonio CS Beck. “Exploiting idle hardware to provide low overhead fault tolerance for vliw processors”. In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13.2 (2017), pp. 1–21.

[101] Rafail Psiakis, Angeliki Kritikakou, and Olivier Sentieys. “NEDA: NOP exploitation with dependency awareness for reliable vliw processors”. In: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2017, pp. 391–396.

[102] Rafail Psiakis. “Performance optimization mechanisms for fault-resilient VLIW processors”. PhD thesis. Rennes 1, 2018.

[103] Rafail Psiakis, Angeliki Kritikakou, and Olivier Sentieys. “Run-time instruction replication for permanent and soft error mitigation in vliw processors”. In: *2017 15th IEEE International New Circuits and Systems Conference (NEWCAS)*. IEEE. 2017, pp. 321–324.

[104] Infineon Technologies AG. AURIX TC27x D-Step User’s Manual V2.2 2014-12. English. 2014.

[105] Xabier Iturbe, Balaji Venu, and Emre Ozer. “Soft error vulnerability assessment of the real-time safety-related ARM Cortex-R5 CPU”. In: *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE. 2016, pp. 91–96.

[106] Arm. Cortex-R5 Core Technical Reference Manual. English. 2011.

[107] Xabier ITURBE, Balaji VENU, Emre OZER, and Shidhartha DAS. "A triple core lock-step (TCLS) ARM® Cortex®-R5 processor for safety-critical and ultra-reliable applications". In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE. 2016, pp. 246–249.

[108] ARM. Cortex-A65AE Core Technical Reference Manual. English. 2022.

[109] ARM. Cortex-A76AE Core Technical Reference Manual. English. 2020.

[110] ARM. Cortex-A78AE Core Technical Reference Manual. English. 2022.

[111] ARM. DynamIQ Shared Unit-AE Technical Reference Manual. English. 2020.

[112] Davide ROSSI, Francesco CONTI, Andrea MARONGIU, Antonio PULLINI, Igor LOI, Michael GAUTSCHI, Giuseppe TAGLIAVINI, Alessandro CAPOTONDI, Philippe FLATRESSE, and Luca BENINI. "PULP: A parallel ultra low power platform for next generation IoT applications". In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE Computer Society. 2015, pp. 1–39.

[113] Igor SILVA, Otávio do ESPÍRITO SANTO, Diego do NASCIMENTO, and Samuel XAVIER-DE-SOUZA. "Cevero: A soft-error hardened soc for aerospace applications". In: *Anais Estendidos do X Simpósio Brasileiro de Engenharia de Sistemas Computacionais*. SBC. 2020, pp. 121–126.

[114] Satyam SHUKLA and Kailash Chandra RAY. "A low-overhead reconfigurable RISC-V quad-core processor architecture for fault-tolerant applications". In: *IEEE Access* 10 (2022), pp. 44136–44146.

[115] Andrew WATERMAN, Yunsup LEE, David PATTERSON, Krste ASANOVIC, Volume I User level ISA, Andrew WATERMAN, Yunsup LEE, and David PATTERSON. "The RISC-V instruction set manual". In: *Volume I: User-Level ISA', version* 2 (2014).

[116] Markus ULBRICHT, Li LU, Junchao CHEN, and Milos KRSTIC. "The TETRISC SoC—A resilient quad-core system based on the ResiliCell approach". In: *Microelectronics Reliability* 148 (2023), p. 115173.

[117] Junchao CHEN, Li LU, Marko ANDJELKOVIC, Markus ULBRICHT, and Milos KRSTIC. "Adaptive Lock-Step System for Resilient Multiprocessing Architectures". In: *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*. 2023, pp. 1–6. DOI: `10.1109/NorCAS58970.2023.10305460`.

[118] Michael ROGENMOSER, Yvan TORTORELLA, Davide ROSSI, Francesco CONTI, and Luca BENINI. "Hybrid Modular Redundancy: Exploring Modular Redundancy Approaches in RISC-V Multi-Core Computing Clusters for Reliable Processing in Space". In: *arXiv preprint arXiv:2303.08706* (2023).

[119] Michael ROGENMOSER, Nils WISTOFF, Pirmin VOGEL, Frank GÜRKAYNAK, and Luca BENINI. "On-demand redundancy grouping: Selectable soft-error tolerance for a multicore cluster". In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2022, pp. 398–401.

[120] Dirk KOCH, Christian HAUBELT, and Jürgen TEICH. "Efficient hardware checkpointing: concepts, overhead analysis, and implementation". In: *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*. 2007, pp. 188–196.

[121] Alban Bourge, Olivier Muller, and Frédéric Rousseau. “Automatic high-level hardware checkpoint selection for reconfigurable systems”. In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2015, pp. 155–158.

[122] Tuo Li, Jude Angelo Ambrose, and Sri Parameswaran. “RECORD: Reducing register traffic for checkpointing in embedded processors”. In: *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2016, pp. 582–587.

[123] “Cache-aided rollback error recovery (CARER) algorithm for shared-memory multiprocessor systems”. In: *[1990] Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*. IEEE. 1990, pp. 82–88.

[124] Daniel J Sorin, Milo MK Martin, Mark D Hill, and David A Wood. “SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery”. In: *ACM SIGARCH Computer Architecture News* 30.2 (2002), pp. 123–134.

[125] Daniel Sánchez, Juan L Aragón, and Jose M Garcia. “A log-based redundant architecture for reliable parallel computation”. In: *2010 International Conference on High Performance Computing*. IEEE. 2010, pp. 1–10.

[126] Luke Yen, Jayaram Bobba, Michael R Marty, Kevin E Moore, Haris Volos, Mark D Hill, Michael M Swift, and David A Wood. “LogTM-SE: Decoupling hardware transactional memory from caches”. In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE. 2007, pp. 261–272.

[127] Gulay Yalcin, Osman Unsal, and Adrian Cristal. “FaulTM: error detection and recovery using hardware transactional memory". In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2013, pp. 220–225.

[128] Rico AMSLINGER, Christian PIATKA, Florian HAAS, Sebastian WEIS, Theo UNGERER, and Sebastian ALTMEYER. "Hardware multiversioning for fail-operational multithreaded applications". In: *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 9-11 Sept. 2020, Porto, Portugal.* Ed. by Miguel AREIAS, Jorge BARBOSA, and Inês DUTRA. 2020. DOI: `10.1109/sbac-pad49847.2020.00014`.

[129] Rico AMSLINGER, Sebastian WEIS, Christian PIATKA, Florian HAAS, and Theo UNGERER. "Redundant execution on heterogeneous multi-cores utilizing transactional memory". In: *Lecture Notes in Computer Science* 10793 (2018), pp. 155–167. DOI: `10.1007/978-3-319-77610-1\_12`.

[130] Rico AMSLINGER. "Loosely-coupled fail-operational execution on embedded heterogeneous multi-cores". doctoralthesis. Universität Augsburg, 2021, p. 193.

[131] Albert MEIXNER and Daniel J SORIN. "Error detection using dynamic dataflow verification". In: *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. IEEE. 2007, pp. 104–118.

[132] Roshan G RAGEL and Sri PARAMESWARAN. "A hybrid hardware–software technique to improve reliability in embedded processors". In: *ACM Transactions on Embedded Computing Systems (TECS)* 10.3 (2011), pp. 1–16.

[133] José Rodrigo AZAMBUJA, Mauricio ALTIERI, Jürgen BECKER, and Fernanda Lima KASTENSMIDT. "HETA: Hybrid error-detection technique using assertions". In: *IEEE Transactions on Nuclear Science* 60.4 (2013), pp. 2805–2812.

[134] Augusto Hoppe, Jürgen Becker, and Fernanda Lima Kastensmidt. “Fine grained control flow checking with dedicated FPGA monitors”. In: *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*. IEEE. 2020, pp. 219–224.

[135] Augusto Hoppe, Jürgen Becker, and Fernanda Lima Kastensmidt. “High-speed Hardware Accelerator for Trace Decoding in Real-Time Program Monitoring”. In: *2021 IEEE 12th Latin America Symposium on Circuits and System (LASCAS)*. IEEE. 2021, pp. 1–4.

[136] Augusto Wankler Hoppe. “Real-Time Trace Decoding and Monitoring for Safety and Security in Embedded Systems”. PhD thesis. Dissertation, Karlsruhe, Karlsruher Institut für Technologie (KIT), 2020, 2021.

[137] Arm. CoreSight Components - Technical Reference Manual, English. 2009.

[138] Tuo Li, Roshan Ragel, and Sri Parameswaran. “Reli: Hardware/software checkpoint and recovery scheme for embedded processors”. In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2012, pp. 875–880.

[139] Florian Haas, Sebastian Weis, Theo Ungerer, Gilles Pokam, and Youfeng Wu. “Fault-tolerant execution on cots multi-core processors with hardware transactional memory support”. In: *Architecture of Computing Systems-ARCS 2017: 30th International Conference, Vienna, Austria, April 3–6, 2017, Proceedings 30*. Springer. 2017, pp. 16–30.

[140] Florian Haas, Sebastian Weis, Stefan Metzlaff, and Theo Ungerer. “Exploiting Intel TSX for fault-tolerant execution in safety-critical systems”. In: *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE. 2014, pp. 197–202.

[141] Richard M Yoo, Christopher J Hughes, Konrad Lai, and Ravi Rajwar. “Performance evaluation of Intel® transactional synchronization extensions for high-performance computing”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* 2013, pp. 1–11.

[142] Ralph Nathan and Daniel J Sorin. “Nostradamus: Low-cost hardware-only error detection for processor cores”. In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE. 2014, pp. 1–6.

[143] Christoph Kühbacher. “Analyzable dataflow executions with adaptive redundancy”. doctoralthesis. Universität Augsburg, 2022, p. 239.

[144] Richard M. Stallman and the GCC Developer Community. Using the GNU Compiler Collection. English. 2023.

[145] LLVM project. LLVM Documentation. English. 2018.

[146] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification. English. 1995.

[147] Yoav Raz. “The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Mangers Using Atomic Commitment”. In: *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings.* Ed. by Li-Yan Yuan. Morgan Kaufmann, 1992, pp. 292–312.

[148] E. W. Dijkstra. “Solution of a Problem in Concurrent Programming Control”. In: *Commun. ACM* 8.9 (Sept. 1965), p. 569. doi: `10.1145/365559.365617`. url: `https://doi.org/10.1145/365559.365617`.

[149] Gadi TAUBENFELD. “The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms”. In: *International Symposium on Distributed Computing*. Springer. 2004, pp. 56–70.

[150] Hagit ATTIYA and Jennifer WELCH. Distributed computing: fundamentals, simulations, and advanced topics. Vol. 19. John Wiley & Sons, 2004.

[151] Advanced Micro Devices (AMD). VCU118 Evaluation Board User Guid, UG1224. 2023.

[152] Advanced Micro Devices (AMD). Vivado 2022.2. 2023.

[153] P. DUHAMEL and M. VETTERLI. “Fast fourier transforms: A tutorial review and a state of the art”. In: *Signal Processing* 19.4 (1990), pp. 259–299. DOI: `https://doi.org/10.1016/0165-1684(90)90158-U`. URL: `https://www.sciencedirect.com/science/article/pii/016516849090158U`.

[154] O. BOTTEMA and B. ROTH. Theoretical Kinematics. Dover Books on Physics. Dover Publications, 2012.

[155] Kumar CHELLAPILLA, Sidd PURI, and Patrice SIMARD. “High Performance Convolutional Neural Networks for Document Processing”. In: (Oct. 2006).

[156] C. A. R. HOARE. “Quicksort”. In: *The Computer Journal* 5.1 (Jan. 1962), pp. 10–16. DOI: `10.1093/comjnl/5.1.10`.

[157] Donald Ervin KNUTH. “Sorting and searching”. In: *The art of computer programming* 3 (1998).

[158] Fischer BLACK and Myron SCHOLES. “The Pricing of Options and Corporate Liabilities”. In: *Journal of Political Economy* 81.3 (1973), pp. 637–654. (Visited on 10/10/2023).

[159] Christian BIENIA, Sanjeev KUMAR, Jaswinder Pal SINGH, and Kai LI. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’08. Toronto, Ontario, Canada: Association for Computing Machinery, 2008, pp. 72–81. DOI: `10.1145/1454115.1454128`.

[160] Frontgrade GAISLER. BCC User’s Manual. 2023.

[161] L. BENINI and G. DE MICHELI. “Networks on chips: a new SoC paradigm”. In: *Computer* 35.1 (2002), pp. 70–78. DOI: `10.1109/2.976921`.

[162] Jörg HENKEL et al. “Invasive manycore architectures”. In: *17th Asia and South Pacific Design Automation Conference*. 2012, pp. 193–200. DOI: `10.1109/ASPDAC.2012.6164944`.

[163] Benoit Dupont de DINECHIN. “Kalray MPPA®: Massively parallel processor array: Revisiting DSP acceleration with the Kalray MPPA Manycore processor”. In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. 2015, pp. 1–27. DOI: `10.1109/HOTCHIPS.2015.7477332`.

[164] Jan HEISSWOLF. “A scalable and adaptive network on chip for many-core architectures”. In: (2014).

[165] Nidhi ANANTHARAJAIAH, Felix KNOPF, and Juergen BECKER. “Ant Colony Optimization Based NoCs for Flexible Spatial Isolation in Mixed Criticality Systems”. In: *2021 IEEE 34th International System-on-Chip Conference (SOCC)*. IEEE. 2021, pp. 248–253.

[166] Nidhi ANANTHARAJAIAH, Zhe ZHANG, and Juergen BECKER. “Multilayered NoCs with Adaptive Routing for Mixed Criticality Systems”. In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications: 17th International Symposium, ARC*

2021, Virtual Event, June 29–30, 2021, Proceedings 17. Springer. 2021, pp. 125–139.

[167] Babak Aghaei, Midia Reshadi, Mohammad Masdari, Seyed Hadi Sajadi, Mehdi Hosseinzadeh, and Aso Darwesh. “Network adapter architectures in network on chip: comprehensive literature review”. In: *Cluster Computing* 23 (2020), pp. 321–346.

[168] Marcelo Ruaro, Felipe B Lazzarotto, César A Marcon, and Fernando G Moraes. “DMNI: A specialized network interface for NoC-based MPSoCs”. In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2016, pp. 1202–1205.

[169] Jian Liang, Sriram Swaminathan, and Russell Tessier. “aSOC: A scalable, single-chip communications architecture”. In: *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*. IEEE. 2000, pp. 37–46.

[170] Mohammad Fattah, Masoud Daneshtalab, Pasi Liljeberg, and Juha Plosila. “Transport layer aware design of network interface in many-core systems”. In: *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE. 2012, pp. 1–7.

[171] Jens Sparsø, Evangelia Kasapaki, and Martin Schoeberl. “An area-efficient network interface for a TDM-based network-on-chip”. In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2013, pp. 1044–1047.

[172] Seung Eun Lee, Jun Ho Bahn, Yoon Seok Yang, and Nader Bagherzadeh. “A generic network interface architecture for a networked processor array (NePA)”. In: *Architecture of Computing Systems–ARCS 2008: 21st International Conference, Dres-*

den, Germany, February 25-28, 2008. Proceedings 21. Springer. 2008, pp. 247–260.

[173] Andrei RADULESCU, John DIELISSEN, Santiago González PESTANA, Om Prakash GANGWAL, Edwin RIJPKEMA, Paul WIELAGE, and Kees GOOSSENS. "An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration". In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 24.1 (2004), pp. 4–17.

[174] Masoud DANESHTALAB, Masoumeh EBRAHIMI, Juha PLOSILA, and Hannu TENHUNEN. "CARS: Congestion-aware request scheduler for network interfaces in NoC-based manycore systems". In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2013, pp. 1048–1051.

[175] Praveen BHOJWANI and Rabi N MAHAPATRA. "Core network interface architecture and latency constrained on-chip communication". In: *7th International Symposium on Quality Electronic Design (ISQED'06)*. IEEE. 2006, 6–pp.

[176] L. BENINI and G. DE MICHELI. Networks on chips: technology and tools (Systems on Silicon). Morgan Kaufmann, 2006.

[177] W.J. DALLY. "Virtual-channel flow control". In: *IEEE Transactions on Parallel and Distributed Systems* 3.2 (1992), pp. 194–205. DOI: `10.1109/71.127260`.

[178] Christoph KÜHBACHER, Christian MELLWIG, Florian HAAS, and Theo UNGERER. "A functional programming model for embedded dataflow applications". In: *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), 15-19 Jul 2019, Milwaukee, WI, USA*. 2019. DOI: `10.1109/compsac.2019.10281`.

[179] Simon Reder, Leonard Masing, Harald Bucher, Timon ter Braak, Timo Stripf, and Jürgen Becker. "A WCET-aware parallel programming model for predictability enhanced multicore architectures". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2018, pp. 943–948. doi: `10.23919/DATE.2018.8342145`.

[180] Simon Reder. "Compileroptimierung und parallele Code-Generierung für zeitkritische eingebettete Multiprozessorsysteme". PhD thesis. Dissertation, Karlsruhe, Karlsruher Institut für Technologie (KIT), 2020, 2020.

[181] Christian Ferdinand and Reinhold Heckmann. "ait: Worst-case execution time prediction by static program analysis". In: *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27 August 2004 Toulouse, France*. Springer. 2004, pp. 377–383.

[182] Subhabrata Ganguli and Kailash Krishnaswamy. Vision based navigation and guidance system. US Patent 7,881,497. Feb. 2011.

[183] David Mueller and Umut Durak. "Enhanced Functions for a Parallel Multicore Ground Proximity Warning System". In: *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*. 2018, pp. 1–7. doi: `10.1109/DASC.2018.8569446`.

[184] AbsInt Angewandte Informatik GmbH. `https://www.absint.com`. Accessed: 20.10.20223.

[185] Lynn Elliot Cannon. "A Cellular Computer to Implement the Kalman Filter Algorithm". PhD thesis. USA, 1969.

[186] William W. Plummer. "TCP Checksum Function Design". In: *SIGCOMM Comput. Commun. Rev.* 19.2 (Apr. 1989), pp. 95–101. doi: `10.1145/378444.378454`. url: `https://doi.org/10.1145/378444.378454`.

[187] W. W. PETERSON and D. T. BROWN. “Cyclic Codes for Error Detection”. In: *Proceedings of the IRE* 49.1 (1961), pp. 228–235. DOI: `10.1109/JRPROC.1961.287814`.

[188] J. FLETCHER. “An Arithmetic Checksum for Serial Transmissions”. In: *IEEE Transactions on Communications* 30.1 (1982), pp. 247–252. DOI: `10.1109/TCOM.1982.1095369`.

[189] Elena DUBROVA and Shohreh Sharif MANSOURI. “A BDD-Based Approach to Constructing LFSRs for Parallel CRC Encoding”. In: *2012 IEEE 42nd International Symposium on Multiple-Valued Logic*. 2012, pp. 128–133. DOI: `10.1109/ISMVL.2012.20`.

[190] Theresa C. MAXINO and Philip J. KOOPMAN. “The Effectiveness of Checksums for Embedded Control Networks”. In: *IEEE Transactions on Dependable and Secure Computing* 6.1 (2009), pp. 59–72. DOI: `10.1109/TDSC.2007.70216`.

[191] Robert BAUMANN. “Soft errors in advanced computer systems”. In: *IEEE Design & Test of Computers* (2005).

[192] Fan WANG and Vishwani D AGRAWAL. “Single event upset: An embedded tutorial”. In: *21st International Conference on VLSI Design (VLSID 2008)*. IEEE. 2008.

[193] Mentor Simens AG. Questa advanced simulator 10.6b. English. 2017.

[194] Advanced RISC Machines Limited (ARM). Arm Cortex-A76AE CoreRevision Technical Reference Manual. English. 2020.

[195] A LINDOSO, L ENTRENA, M GARCIA-VALDERAS, and L PARRA. “A hybrid fault-tolerant LEON3 soft core processor implemented in low-end SRAM FPGA”. In: *IEEE Transactions on Nuclear Science* (2016).

[196] Adam JACOBS, Grzegorz CIESLEWSKI, Alan D GEORGE, Ann GORDON-ROSS, and Herman LAM. "Reconfigurable fault tolerance: A comprehensive framework for reliable and adaptive FPGA-based space computing". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* (2012).

[197] Israel KOREN and C Mani KRISHNA. Fault-tolerant systems. Morgan Kaufmann, 2020.

[198] W.J. DALLY and H. AOKI. "Deadlock-free adaptive routing in multicomputer networks using virtual channels". In: *IEEE Transactions on Parallel and Distributed Systems* 4.4 (1993), pp. 466–475. DOI: 10.1109/71.219761.

[199] Yifeng GUO, Dakai ZHU, and Hakan AYDIN. "Reliability-aware power management for parallel real-time applications with precedence constraints". In: *2011 International Green Computing Conference and Workshops*. IEEE. 2011, pp. 1–8.

Publications

[Kem23a] Fabian Kempf, Julian Hoefer, Tim Hotfilter, and Juergen Becker. "A Low-Stall Methodology for an Interleaved Processor State Replication". In: *2023 IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. 2023, pp. 1–8.

[Kem23b] Fabian Kempf and Juergen Becker. "Leveraging Adaptive Redundancy in Multi-Core Processors for Realizing Adaptive Fault Tolerance in Mixed-Criticality Systems". In: *2023 12th Mediterranean Conference on Embedded Computing (MECO)*. 2023, pp. 1–5. doi: 10.1109/MECO58584.2023.10154986.

[Kem23c] Fabian Kempf et al. "The ZuSE-KI-Mobil AI Accelerator SoC: Overview and a Functional Safety Perspective". In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2023, pp. 1–6. doi: 10.23919/DATE56975.2023.10137257.

[Kem22a] Fabian Kempf, Julian Hoefer, Fabian Kress, Tim Hotfilter, Tanja Harbaum, and Juergen Becker. "Runtime Adaptive Cache Checkpointing for RISC Multi-Core Processors". In: *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. 2022, pp. 1–6. doi: 10.1109/SOCC56010.2022.9908110.

[Kem22b] Fabian KEMPF, Christoph KÜHBACHER, Christian MELL-WIG, Sebastian ALTMEYER, Theo UNGERER, and Juergen BECKER. “A holistic hardware-software approach for fault-aware embedded systems”. In: *2022 25th Euromicro Conference on Digital System Design (DSD)*. 2022, pp. 704–711. DOI: `10.1109/DSD57027.2022.00099`.

[Kem21] Fabian KEMPF, Thomas HARTMANN, Steffen BAEHR, and Juergen BECKER. “An Adaptive Lockstep Architecture for Mixed-Criticality Systems”. In: *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2021, pp. 7–12. DOI: `10.1109/ISVLSI51109.2021.00013`.

[Kem19] Fabian KEMPF, Nidhi ANANTHARAJAIAH, Leonard MASING, and Juergen BECKER. “A Network on Chip Adapter for Real-Time and Safety-Critical Applications”. In: *2019 32nd IEEE International System-on-Chip Conference (SOCC)*. 2019, pp. 39–44. DOI: `10.1109/SOCC46988.2019.1570558594`.

[Hoe24] Julian HOEFER, Michael GAUSS, Manuela ADAMS, Fabian KRESS, Fabian KEMPF, Christian KARLE, Tanja HARBAUM, Andreas BARTH, and Becker JUERGEN. “A Challenge-Based Blended Learning Approach for an Introductory Digital Circuits and Systems Course”. In: *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2024. (accepted).

[Kre23] Fabian KRESS, Johannes PFAU, Fabian KEMPF, Patrick SCHMIDT, Zhuofan HE, Tanja HARBAUM, and Juergen BECKER. “Automated Replacement of State-Holding Flip-Flops to Enable Non-Volatile Checkpointing”. In: *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*. 2023, pp. 1–6.

[Tee23] Gerd H. TEEPE et al. “ZUSE KI-mobil: Platform for energy efficient AI processors in mobile applications”. In:

MikroSystemTechnik Congress 2023; Congress. 2023, pp. 1–6.

[Hot23a] Tim Hotfilter, Julian Hoefer, Philipp Merz, Fabian Kress, Fabian Kempf, Tanja Harbaum, and Jürgen Becker. "Leveraging Mixed-Precision CNN Inference for Increased Robustness and Energy Efficiency". In: *2023 36th IEEE International System-on-Chip Conference (SOCC).* 2023, pp. 1–6.

[Hot23b] Tim Hotfilter, Julian Hoefer, Fabian Kress, Fabian Kempf, Leonhard Kraft, Tanja Harbaum, and Juergen Becker. "A Hardware-Centric Approach to Increase and Prune Regular Activation Sparsity in CNNs". In: *2023 IEEE 5th International Conference on Artificial Intelligence Circuits and Systems (AICAS).* 2023, pp. 1–5. doi: `10.1109/AICAS57966.2023.10168566`.

[Hoe23] Julian Hoefer, Fabian Kempf, Tim Hotfilter, Fabian Kress, Tanja Harbaum, and Jürgen Becker. "SiFI-AI: A Fast and Flexible RTL Fault Simulation Framework Tailored for AI Models and Accelerators". In: *Proceedings of the Great Lakes Symposium on VLSI 2023.* GLSVLSI '23. Knoxville, TN, USA: Association for Computing Machinery, 2023, pp. 287–292. doi: `10.1145/3583781.3590226`.

[Hot22a] Tim Hotfilter, Fabian Kress, Fabian Kempf, Juergen Becker, and Imen Baili. "Data Movement Reduction for DNN Accelerators: Enabling Dynamic Quantization Through an eFPGA". In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI).* 2022, pp. 371–372. doi: `10.1109/ISVLSI54635.2022.00082`.

[Hot22b] Tim Hotfilter, Fabian Kress, Fabian Kempf, Juergen Becker, Juan Miguel De Haro, Daniel Jiménez-González, Miquel Moretó, Carlos Álvarez, Jesús Labarta, and Imen Baili. "Towards Reconfigurable Accelerators in HPC: Designing a Multipurpose eFPGA Tile for Heterogeneous SoCs". In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2022, pp. 628–631. doi: `10.23919/DATE54114.2022.9774716`.

[Hot21] Tim Hotfilter, Julian Hoefer, Fabian Kress, Fabian Kempf, and Juergen Becker. "FLECSim-SoC: A Flexible End-to-End Co-Design Simulation Framework for System on Chips". In: *2021 IEEE 34th International System-on-Chip Conference (SOCC)*. 2021, pp. 83–88. doi: `10.1109/SOCC52499.2021.9739212`.

[Hot20] Tim Hotfilter, Fabian Kempf, Juergen Becker, Dominik Reinhardt, and Imen Baili. "Embedded Image Processing the European Way: A new platform for the future automotive market". In: *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*. 2020, pp. 1–6. doi: `10.1109/WF-IoT48130.2020.9221396`.

[Red19] Simon Reder et al. "Worst-Case Execution-Time-Aware Parallelization of Model-Based Avionics Applications". In: *Journal of Aerospace Information Systems* 16.11 (2019), pp. 521–533. doi: `10.2514/1.I010749`.

[Ana19] Nidhi Anantharajaiah, Fabian Kempf, Leonard Masing, Fabian Marc Lesniak, and Juergen Becker. "Dynamic and Scalable Runtime Block-Based Multicast Routing for Networks on Chips". In: *Proceedings of the 12th International Workshop on Network on Chip Architectures*. NoCArc. Columbus, Ohio: Association for Computing Machinery, 2019. doi: `10.1145/3356045.3360718`.

[Bae18a] Steffen Baehr, Fabian Kempf, and Juergen Becker. “Data Readout Triggering for Phase 2 of the Belle II Particle Detector Experiment Based on Neural Networks”. In: *2018 31st IEEE International System-on-Chip Conference (SOCC)*. 2018, pp. 174–179. doi: `10.1109/SOCC.2018.8618563`.

[Bae18b] Steffen Baehr, Fabian Kempf, and Juergen Becker. “Data Reduction and Readout Triggering in Particle Physics Experiments Using Neural Networks on FPGAs”. In: *2018 IEEE 18th International Conference on Nanotechnology (IEEE-NANO)*. 2018, pp. 1–4. doi: `10.1109/NANO.2018.8626239`.