Fault-tolerant Execution of Parallel Applications on x86 Multi-core Processors with Hardware Transactional Memory

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Fakultät für Angewandte Informatik
der Universität Augsburg

eingereicht von
Florian Ludwig Haas, M. Sc.

FAULT-TOLERANT EXECUTION OF PARALLEL APPLICATIONS ON X86 MULTI-CORE PROCESSORS WITH HARDWARE TRANSACTIONAL MEMORY

Florian Ludwig Haas, M. Sc.

Erstgutachter:	Prof. Dr. Theo Ungerer
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Tag der mündlichen Prüfung:	22. Juli 2019

Zusammenfassung

Um der anhaltenden Nachfrage nach zunehmender Rechenleistung gerecht zu werden, versuchen die Hersteller das Verhältnis von Rechenleistung zu verbrauchter elektrischer Leistung zu erhöhen, was durch Verkleinerung der Strukturgrößen elektronischer Schaltkreise erreicht werden kann. Jedoch werden die Grenzen des technologisch Machbaren bald erreicht sein, und eine weitere Absenkung der Versorgungsspannung sowie die Erhöhung der Taktraten führen zu steigenden Fehlerraten aufgrund transienter Fehler, die aus der Miniaturisierung der Transistoren und deren zunehmender Anzahl auf einem Chip resultieren.

Solchen Fehlern kann entgegen gewirkt werden, indem Techniken von hochverfügbaren Serversystemen, sowie aus sicherheitskritischen eingebetteten Systemen auch in Standardsysteme integriert werden. Die typische Lockstep-Ausführung, bei der die Zustände zweier redundanter Prozessoren taktweise verglichen werden, ist in komplexen out-of-order Prozessoren kaum umsetzbar. Es wurden jedoch Ansätze vorgestellt, die eine lose Kopplung ermöglichen, sowohl integriert in Hardware, als auch rein Software-basiert. Software-Ansätze ermöglichen eine anwendungsspezifische redundante Ausführung zur Fehlererkennung auf einem COTS- (commercial-off-the-shelf, seriengefertigten) Prozessor ohne weitere Hardware-Anpassungen.

In den vergangenen Jahren erhielt die Transaktionsspeichertechnik zunehmende Aufmerksamkeit in der Forschung zu fehlertoleranten Systemen. Deren Eigenschaft der Isolation und der integrierte Mechanismus zur Erstellung von Sicherungspunkten zur Wahrung der Atomarität brachte mehrere Ansätze zur Verwendung von Transaktionsspeicher für Fehlertoleranz hervor. Mit der Verfügbarkeit erster Hardware-Implementierungen, zum Beispiel TSX in den teureren x86-Prozessoren der *Core*-Familie von Intel, wurden Software-Mechanismen möglich, die auf Hardware-Transaktionsspeicher basieren und dessen Rückrollfähigkeit für den Fehlerfall ausnutzen.

In dieser Dissertation wird eine fehlertolerierende Ausführung mit Transaktionsspeicher auf einem Intel-Prozessor untersucht, die auf lose gekoppelter redundanter Ausführung basiert. Ein Instrumentierungsverfahren, das als Optimierungsmodul für die LLVM-Compiler-Toolchain entwickelt wurde, und eine zu POSIX-Systemen kompatible Bibliothek stellen die Mechanismen für Fehlererkennung und Fehlerbehebung zur Verfügung. Funktionalität und Leistungsfähigkeit des Ansatzes wurden mit Benchmarks aus der SPEC2017-Benchmark-Suite getestet. Die Ergebnisse zeigen, dass eine fehlertolerante redundante Ausführung auf einem x86-Prozessor möglich ist, und dass spezifische Erweiterungen der Hardware die Leistungsfähigkeit noch weiter steigern können.

Die redundante Ausführung mehrfädiger Anwendungen erfordert eine besondere Betrachtung, da indeterministisches Verhalten aufgrund auseinanderlaufender Synchronisierung zwischen redundanten Paaren von Threads auftreten kann, zum Beispiel bei gegenseitigem Ausschluss. Es wird eine Schnittstelle zu den Pthread-Synchronisie-

rungsfunktionen und ein Mechanismus zur Fehlerbehebung beschrieben, um die redundante und fehlertolerierende Ausführung mehrfädiger Anwendungen zu ermöglichen. Dies wurde mithilfe von Benchmarks aus der PARSEC-Suite evaluiert, die die Machbarkeit redundanter Mehrfädigkeit auf COTS-Prozessoren zeigen und die nur geringe Auswirkung der zusätzlichen Schicht auf Leistungsfähigkeit und Skalierung bestätigen.

Abstract

To satisfy the enduring demand for increasing computational power, the processor manufacturers try to raise the performance per Watt of a chip, which can be achieved by minimizing the structure sizes of electronic circuits. However, the technological limits are about to be reached, and the further reduction of the supply voltages and rising frequencies will lead to increased error rates due to transient faults, which result from the miniaturization of transistors, and from the growing number of transistors on a chip.

To mitigate such errors, techniques from dependable server systems and safety-critical embedded systems become attractive in commodity systems as well. However, the typically used cycle-by-cycle lockstep execution of a redundant processor is hardly feasible on a complex out-of-order CPU. Approaches that enable a loose coupling have been proposed, integrated in hardware as well as software-only approaches. Software mechanisms allow to run specific applications redundantly to detect errors on a COTS (commercial-off-the-shelf) processor without hardware modifications.

In recent years, transactional memory gained interest in the research of fault-tolerant systems. Its property of isolation and the integrated checkpointing mechanism to guarantee atomicity spawned multiple approaches to utilize transactional memory for fault tolerance. With the availability of first hardware implementations, for example TSX in the more expensive processors of the Intel x86 *Core* family, software mechanisms that rely on hardware transactional for checkpointing became feasible.

This thesis investigates a fail-operational execution with transactional memory on a COTS Intel CPU, based on loosely-coupled redundant execution. An instrumentation mechanism, which was developed as an optimization pass for the LLVM compilation toolchain, and a support library for POSIX compatible systems provide the functionality for error detection and recovery. The feasibility and the effectiveness of the approach were evaluated with benchmarks of the SPEC2017 benchmark suite. Results show that a fault-tolerant redundant execution can be achieved on an x86 CPU, and that specific enhancements to the hardware could further improve the overall performance.

Multi-threaded applications require further consideration for redundant execution, since indeterminism can occur between redundant pairs of threads, due to diverging synchronization, for example on mutual exclusion. An interface to the Pthread synchronization functions is described, as well as an error recovery mechanism, to enable the redundant and fail-operational execution of multi-threaded applications. This was evaluated by means of benchmarks of the PARSEC suite to prove that redundant multi-threading is feasible on a COTS CPU. The impact of the additional layer on performance and speedup is shown to be minimal.

Danksagung

Großer Dank gilt meinem Doktorvater, Prof. Dr. Theo Ungerer, für die Unterstützung beim Verfassen der Dissertation durch kritische und konstruktive Ratschläge. Teil seines Lehrstuhls zu sein, sowie im Rahmen eines Drittmittelprojekts internationale Kontakte mit Forschern zu knüpfen, trugen wesentlich zur Themenfindung und Konzeptentwicklung dieser Arbeit bei. Ebenfalls bedanken möchte ich mich bei Herrn Prof. Dr. Bernhard Bauer für die Begutachtung dieser Arbeit.

Meinen Kollegen am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme an der Universität Augsburg danke ich für die unzähligen, die eigene Kreativität fördernden Diskussionen fachlicher und privater Natur. Dr. habil. Florian Kluge und Dr. Stefan Metzlaff sei ganz besonders gedankt, da durch sie der Kontakt zum Lehrstuhl während meines Studiums entstand, und durch deren Betreuung in Vorlesungen, Praktika und in der Masterarbeit mein Interesse in diesem Fachgebiet geweckt wurde, was letztlich zur Arbeit an dem dieser Dissertation zugrunde liegenden Forschungsprojekt führte. Bei Dr. Sebastian Weis bedanke ich mich für die jahrelange und freundschaftliche Zusammenarbeit bei Forschungsprojekten und in der Lehre, sowie für die zahlreichen Diskussionen, die einen wesentlichen Beitrag zu dieser Arbeit geleistet haben.

Meiner Familie und besonders meiner Ehefrau Sonja möchte ich für ihre Geduld und die andauernde Unterstützung danken, die mir das Verfassen dieser Arbeit erst ermöglicht haben.

Florian Haas
Augsburg im Juli 2019

Contents

1

Introduction

Contents

The main source of increasing computational power in recent years lies in the miniaturization of the structure sizes of electronic circuits [Hen+17, pp. 21–23]. However, the smaller the circuits become, the more prone they become for transient faults due to destructive charges of impacting particles [Shi+02]. Such faults may toggle a bit and thus may lead to a corrupted data word in memory or in a register. The importance of correct computation results depends on the application domain, an application crash on a personal computer is generally less critical than incorrect calculations in an embedded system that controls a machine or an autonomously driving car. Such safety-critical systems, and also server systems with high reliability demands, are often run in a lockstep configuration with cycle-by-cycle comparison of the redundant processors [Muk08, p. 212]. The double cost for the same computing performance of a lockstep system is not justifiable for commodity systems like personal computers, where the probability of soft errors is low, at least until today.

1.1 Motivation

The decreasing structure sizes for CMOS transistors, and the increasing amount of them on a single chip lead to rising error rates, which possibly cannot be detected and corrected in future processors without any additional redundancy. The susceptibility for soft errors depends not only on the transistor, but also on the supply voltage and the clock rate [Uem+16]. Following the current trend of decreasing supply voltages and rising clock rates, a higher soft error rate results already for a constant structure size.

Until now, electric circuits are designed in a way to ensure correct outcome for valid inputs. For main memory, Error Correcting Codes (ECC) are often used as a protection mechanism against such data corruption. Data in the main memory of servers is kept there for a long time, compared to the time a data word is stored in a register. Although, the processor cores are also susceptible for bit flips. A powerful countermeasure is lockstep execution, where two cores execute the same program with identical data. Correctness is ensured by continuously comparing the states of both cores at each cycle. This is a strict form of redundant execution without temporal displacement, which is typically used in safety-critical embedded systems and servers with high dependability requirements. However, a complete cycle-by-cycle lockstep execution of all cores in a high-performance multi-core processor is at least hard [Muk08, pp. 213–214], and also may not be required for all applications. Thus, a more flexible approach is required for selective and loosely-coupled fault-tolerance for different application domains.

This work describes a software mechanism for redundant execution, which leverages hardware transactional memory (HTM) for checkpointing. Transactional memory is an optimistic synchronization mechanism that allows to access and modify large data structures atomically [Her+93]. Hardware implementations have been proposed for multiple architectures, and are now available in server and desktop processors. X86 CPUs from Intel of the Core and Xeon series offer the *Transactional Synchronization Extensions* (TSX) since the Haswell architecture [Ham+14].

The idea for the topic of this thesis originated in the work done for the research project funded by the Intel Corporation on *Executing Parallel Safety-Critical Applications on COTS Processors by Exploiting Hardware Transactional Memory*. In the three years of the project, which started in 2013, the applicability of hardware transactional memory in safety-critical systems had been researched, especially in the context of fault-tolerance and real-time. TSX has become available in desktop processors at nearly the beginning of the project, and thus became the central part of the project. The main objective was to research the possibilities of how HTM can be leveraged to improve the real-time behavior, and to implement a fault tolerant execution scheme on existing hardware with TSX to support error recovery. As a consequence, the focus of the research for the approaches presented in this thesis lies on x86 processors with transactional memory as a central part of the implementation.

Transactional memory implementations comprise some kind of checkpointing mechanism to restore memory contents and the registers in case of a conflict. Such checkpointing mechanisms can be leveraged for fault-tolerance, both for hardware or software implementations of transactional memory. Lockstep-alike processor architectures have been proposed that utilize a custom transactional memory implementation for fault-tolerant execution [Yal+13]. With Intel TSX as one of the first widely available HTM systems, a software-based fault tolerance mechanism with hardware checkpointing support can be crafted for existing COTS (commercial off the shelf) hardware. The developed techniques described in this thesis are based on Intel TSX and thus require an x86 processor. However, the redundant multi-threading approach is independent of the underlying hardware, and thus is portable to other platforms. The software fault tolerance mechanism with HTM for checkpointing and recovery can be implemented for other systems as well, provided that hardware transactional memory is available on the target platform. Thus, the general idea of an hybrid hardware and software approach for fail-operational execution targets a broad range of systems with dependability requirements, from safety-critical embedded systems to high-performance computing.

1.2 Objectives and Contributions

The main objective of this work is to enable a fault-tolerant execution of multi-threaded applications on a COTS x86 multi-core processor, with redundancy-based error detection and a correction mechanism that leverages the checkpoints of TSX, the transactional memory implementation of Intel.

Contributions of this doctoral thesis:

- Thorough investigation of a fail-operational execution with transactional memory, based on loosely-coupled redundant execution.
- Development of an LLVM-based instrumentation tool to enhance programs with checksums.
- Implementation of an error detection mechanism for the redundant execution of a process on a POSIX compatible operating system.
- Proposal and evaluation of hardware extensions to further increase the applicability of the approach.
- Support for synchronization and error recovery in redundant processes with multiple threads, and evaluation of the proposed redundant thread synchronization extension.

1.3 Overview

The contents of this thesis is structured in six chapters, beginning with the introduction into fault tolerance in redundant systems in this chapter. Chapter 2 provides background information on fault tolerance and redundancy. Further, multi-threading for parallel applications in shared memory systems is discussed, followed by an explanation of hardware transactional memory, and Intel TSX in particular. At the end of the chapter, the LLVM framework is outlined with a tight explanation of its structure. Chapter 3 examines related work upon which the ideas of the following chapters are based, as well as alternative approaches implemented in hardware and software. In Chapter 4, the baseline mechanism for fault-tolerant execution of single-threaded applications is described, presenting the loosely-coupled redundancy approach with explanation of the program instrumentation, and the software fault-tolerance library that handles error detection and recovery. This chapter ends with the evaluation of the approach and the discussed hardware extensions. Chapter 5 defines the requirements for a redundant execution of multi-threaded applications, upon which an implementation of the basic synchronization mechanisms is explained. After the presentation of the error recovery mechanism for redundant multi-threaded programs, an evaluation shows the performance impact of redundant synchronization and error recovery. A summary of the presented work is given in Chapter 6.3, followed with an outlook on further applications of loosely-coupled redundancy, and custom designs for a fault-tolerant embedded multi-core processor.

2

Background

Contents

This chapter provides the required background on fault tolerance, multi-threading, and transactional memory to set the foundation for the proposed fault-tolerance approaches. Section 2.1 introduces the basics of fault tolerance, starting with the definitions of faults and errors. Then, an overview of error models and redundancy concepts is given, and mechanisms for error detection are introduced. Section 2.2 gives a brief overview of multi-threading and synchronization mechanisms, and Section 2.3 introduces transactional memory, with a focus on Intel TSX as an hardware implementation. Hardware support for software-based fault tolerance approaches is described in Section 2.4, including the basic idea of leveraging hardware transactional memory for recovery.

2.1 Fault Tolerance

The environmental influence on processors and memory, e. g. by radiation, has been discovered decades ago [May+79]. For example, alpha radiation has been found as the cause for faulty chips when radioactive contaminated material was used for chip packaging [Muk08, pp. 2–3]. Faults can also be caused by cosmic radiation, where neutrons and other particles lead to flipped values or transistor outputs [Sor09, pp. 3–4]. This results in potential soft errors in computational logic and memory, which require mechanisms to detect and correct erroneous data. Soft error rates keep increasing due to the technological trend of smaller structure sizes and a higher quantity of transistors per processor [Shi+02].

2.1.1 From Fault to Failure

An unexpected misbehavior of a processor often originates in mistakes that occurred during software development, so-called *software bugs*. Although, assuming correct software and correct input, computation results can still be wrong, or the control flow of the program can get corrupted. This observable incident, which results in system malfunction, is denoted as a *failure* [Sor09, p. 2]. The cause for such failure may lie in an incident deep in the logic circuit of the hardware.

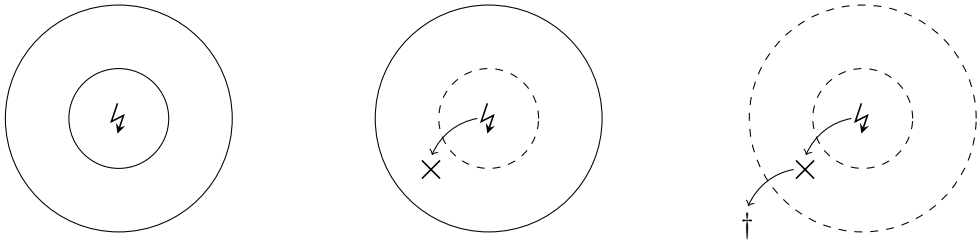
Fault

A defective circuit or a dropped charge due to environmental influence may lead to a *fault* [Muk08, p. 6]. In general, all unintentional changes can be considered as faults, and a fault can happen on any abstraction layer of a computer system. Mukherjee defines six abstractions layers: process technology, circuits, architecture, firmware, operating system, and user application [Muk08, p. 7]. For example, design faults occur at the lower layers like circuits or architecture, while software bugs appear in the upper layers, e. g. the operating system or the application [Muk08, p. 6].

The discharge of a transistor due to a neutron that arises from cosmic radiation leads to a fault at the lowest layer. However, such fault may be *masked* by the logic and thus does not become visible. For example, an `AND` gate would mask a toggled signal if the other signal is zero [Sor09, p. 2]. Since faults require the actual usage of the affected signal or value to manifest in the next layer as an error, the design of the system often causes faults to be without effect.

Error

A faulty signal can become manifest as an *error*, if it is not masked [Sor09, p. 2]. However, only if the manifestation of the fault can be observed, it is considered as an error [Muk08, p. 7]. For example, such a fault may occur in a physical register, resulting in a wrong



(a) A fault occurs in an inner scope

(b) The fault is detected as an error in the outer scope

(c) An unhandled error may result in a system failure

Figure 2.1: A fault (ϟ) becomes manifest in an error (×) in the scope where it can be detected. An undetected or unrecoverable error may lead to a failure (†) of the system. [Muk08, p. 8]

value for a specific register. This fault only becomes visible as an error if the register is read and its data is used for further computation. Otherwise, when this physical register is not mapped to an architecture register, that fault is masked again, this time by the micro-architecture of the processor [Sor09, p. 2]. Likewise, faults in instructions are masked when the affected part is not needed, e. g. in a `NOP` instruction [Sor09, p. 2]. In general, a fault becomes manifest in an error in the *scope* where it can be detected or recovered [Muk08, p. 7]. When an error is corrected in that scope, it does not propagate to the outer scope.

Figure 2.1 illustrates a fault that is detected as an error but not corrected, thus resulting in a potential system failure. A bit-flip in a DRAM cell, for example, represents a fault (see Figure 2.1a). The faulty word can be detected as an error if the main memory is protected with ECC, which also corrects the error inside the scope, as shown in Figure 2.1b. However, if the main memory does not offer error correction through ECC, the error passes the scope, and eventually may lead to a system failure (see Figure 2.1c).

Failure

If an error leads to a system malfunction, it is referred to as failure [Muk08, p. 8]. This can be the case if an error is not detected, or if it cannot corrected within the outermost scope. As a consequence, the system can no longer guarantee its correctness [Muk08, p. 8]. A failure is not a direct consequence of an error, since the error still may be masked on a layer above its origin [Muk08, p. 8]. To continue with the previous example, a corrupted word in the main memory can be masked, e. g. if it gets overwritten and is not read before.

2.1.2 Fault Duration

The impact of errors and the underlying faults on a system, and the mechanisms to detect and treat them, depend on the duration of the errors and faults. Since errors are the potential manifestations of the faults that occurred in an underlying layer, the following sections refer to faults, but also concern the resulting errors.

Permanent Faults

A fault is *permanent*, if it occurs once and then persists until repair [Sor09, p. 3]. This can lead to a faulty unit, which then generates erroneous output, potentially resulting in a repeating error. To recover from permanent faults, the affected unit either has to be repaired, or a spare has to be used.

When permanent faults in CMOS origin in fabrication defects or design bugs, they often are detected before delivering the produced chips [Sor09, pp. 4, 5]. However, permanent faults are not guaranteed to be detected by testing, as various processor design bugs have shown in the past, e. g. the Intel Pentium floating point division bug [Sor09, p. 5]. Another source of permanent faults is physical wear-out, where the chip circuits fail during operation, for example due to electromigration [Muk08, p. 15]. This effect, caused by electrons colliding with the metal atoms of the interconnect, is known for a long time and has been investigated by Black in 1969 [Bla69]. The collisions displace atoms of the conductor metal over time and lead to a hole, thus breaking the interconnect. Further flaws that can lead to permanent faults are metal stress voiding, gate oxide wear-outs, and hot carrier injection [Muk08, pp. 16–18]. The causes for these are mostly high frequencies and high temperatures, hence an often used remedy is dynamic voltage and frequency scaling [Sor09, p. 4].

The probability for permanent faults rises with the age of the chip, since after a specified lifetime, the wear-out of interconnects and transistors accelerates [Muk08, p. 8]. In combination with the initial defects and the permanent faults due to *infant mortality*, the fault rate resembles the well-known *bathtub curve* [Muk08, p. 8].

Intermittent Faults

A fault may be not permanent, but may occur repeatedly, caused by the same flaw. Such incident is named *intermittent fault* [Sor09, p. 3]. For example, changes in the resistance of a connection due to varying temperatures may result in an intermittent fault. This can escalate to a permanent fault at some point in time when the interconnection is eventually broken. Intermittent faults often occur due to wear-out before a component fails permanently [Sor09, p. 5].

Transient Faults

A *transient fault* does not persist and also does not repeat [Sor09, p. 3]. Its potential manifestation, a transient error, is also referred to as *single event upset*, since it occurs only once [Sor09, p. 3].

One cause for transient faults is the influence of cosmic radiation on a chip through neutrons and other particles, which result from spallation in the atmosphere, as it has been predicted in 1979 [Zie+79] and reported in a study of errors in the main memory of multiple large computers [Nor96]. Alpha particles stem from decay of radioactive nuclei, which can occur in the packaging material of chips [Muk08, p. 20]. The sensitivity of the chip can be reduced by additional shielding, but this cannot fully prevent the influence of radiation. Thus, the chip requires protection against alpha particles through error detection and correction mechanisms. When particles of cosmic radiation collide with the atmosphere, new particles like neutrons are produced, leading to further collisions and more particles in a cascading manner [Muk08, p. 23]. The neutron flux depends on the location and the altitude, and reaches its maximum at approximately 15 km above sea level. Thus, the control units in airplanes are exposed to a much higher neutron flux compared to computers on the surface of the earth. An additional problem is the impractical shielding of chips against neutrons, which would require meters of concrete [Muk08, pp. 24–26]. Alpha particles interact directly with the electrons in the silicon by generating pairs of electrons and holes in transistors, which then potentially lead to an incorrect signal. Neutrons however generate these electron-hole pairs indirectly through collision in the semiconductor, producing particles that can cause ionization tracks, which then lead to a fault [Muk08, pp. 26–27].

The critical charge entering a circuit that is required to produce a fault is denoted as Q_{crit} [Sor09, p. 3]. If an SRAM cell or a transistor gets hit by a particle and its charge exceeds Q_{crit}, the state of that element can flip. The *soft error rate* (SER) of an element can be determined by combining the critical charge with measurements obtained with the help of particle accelerators. Alternatively, simulation models allow to calculate the probabilities of bit flips. [Muk08, pp. 29–30, 46–49]

Since transient faults are observable for short time only, they may become manifest as an error unnoticed. The detection of this error may happen later in time where a correction is no longer possible, e. g. when reading the data again, since the erroneous data has already been persisted. This effect is referred to as *detected unrecoverable error* [Muk08, p. 32], and its cause is a *silent data corruption* [Muk08, p. 32]. Although a detected unrecoverable error may be masked by an upper layer, the chance of corrupted data exists. This requires countermeasures to detect such errors within sufficient time to still be capable of recovery.

2.1.3 Effects of Technology Scaling on Fault Rates

Decreasing CMOS structure sizes are required to reduce the overall power consumption of the chip for further performance improvements, although this results in increasing soft error rates. In 2002, Shivakumar et al. analyzed that the SER is increasing for smaller structures due to the reduction of the critical charge Q_{crit}, which depends on the structure size [Shi+02]. They concluded that the technology trends lead to rapidly increasing fault rates. Luckily, new transistor technologies like FinFET and Tri-Gate FET are more resilient against soft errors and thus feature a lower SER than what has been predicted for future CMOS technology [Sei+12; Uem+16].

Beside the transistor size, also the supply voltage and the clock rate influence the susceptibility for soft errors [Uem+16; Shi+02]. For a given structure size, higher clock rates or lower voltages lead to a higher SER.

Even when the fault rate for a single transistor decreases due to smaller structure sizes, processors contain an still increasing amount of transistors, which in total leads to an increasing SER for the overall processor [Sor09, p. 6]. As a consequence, soft errors remain an important problem in modern processors, and thus demand for mitigation methods on multiple layers of the overall system.

2.1.4 Error Models

Before the design and the implementation of error detection mechanisms and strategies for recovery, a model of the potential errors that can occur in the system has to be created. This model is an abstraction of the underlying physical phenomena, used to describe the potential impact of faults and the resulting behavior of the specific chip [Sor09, p. 7]. Three categories exist for the classification of error models: the type of error, its duration, and the number of simultaneous errors [Sor09, p. 7].

Different error models have been proposed to describe the system reaction to specific types of errors. The most popular model, the *stuck-at* model describes an error due to a faulty bit, which is stuck at 0 or 1 [Sor09, p. 7]. However, this does not regard short-circuits or cross-talk, where one bit takes the value of a neighbor bit. This situation can be described by the *bridging error* model [Sor09, p. 8]. Also higher-level models exist, like the *fail-stop* model, in which a component stops when an error occurs [Sor09, p. 8].

The error duration is categorized according the fault duration classification described in Section 2.1.2: transient, intermittent, and permanent. A system, which is designed based on an error model that expects only transient errors, will probably fail if a permanent error occurs.

The third category regards the number of errors that can occur simultaneously. Although multiple errors at the same time are often disregarded, they pose a threat to a critical system [Sor09, pp. 8–9]. For example DMR configurations assume that an error occurs only in one component, but not in both, and neither in one component and the

comparator. A system that experiences such a situation but is designed after an error model without multiple simultaneous errors, will suffer a failure.

Multiple simultaneous errors can also occur if a fault happened in the past and remains undetected as a *latent error* [Sor09, p. 9]. When another fault becomes manifest as an error and is detected, the delayed detection of the latent error can lead to multiple simultaneous errors. In case such circumstance is not covered by the error model, a system failure may result.

2.1.5 Redundancy

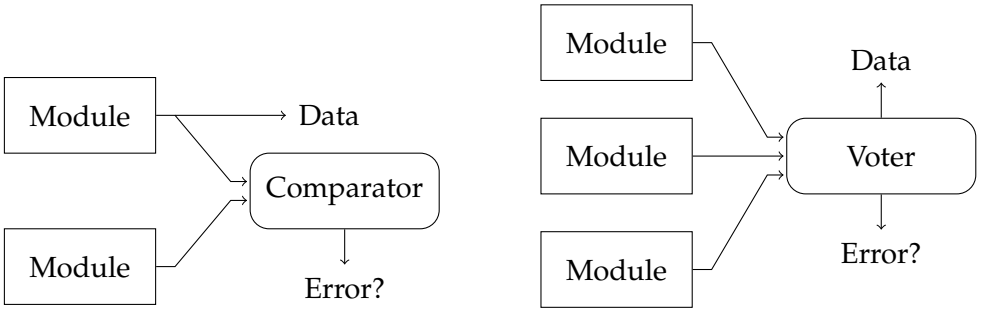
Error detection requires some kind of redundancy, since without additional information or repeated computation, no evidence for the correctness of the data can be given, and thus error detection would be impossible [Sor09, p. 19].

The general idea of redundancy is to provide an additional copy of a module to generate the same data twice or more often. Assuming that only errors described by the error model occur, one ore more results from the redundant modules are then guaranteed to be free of errors. For example, if only one transient error can occur simultaneously, this error affects one module but not the redundant copy. In general, three types of redundancy are distinguished: spatial, temporal, and information redundancy [Sor09, p. 19].

Spatial Redundancy

The replication of a module adds *spatial redundancy* to a system, and by computing the same data twice, the comparison of both copies enables error detection [Sor09, p. 19].

Multiple possibilities exist to provide spatial redundancy on the different abstraction layers of a computer system. On a lower layer, individual units of the processor can



(a) Dual-Modular Redundancy (b) Triple-Modular Redundancy

Figure 2.2: Dual Modular Redundancy allows only to detect errors [Sor09, p. 20], while the voter in a Triple Modular Redundant system selects the majority of the outputs [Sor09, p. 21].

be replicated, like registers or execution units [Sor09, p. 20]. Also a whole processor can be duplicated, which then resembles a *lockstep processor* [Sor09, p. 21]. Redundant modules require to provide identical output in the error-free case, but they are not constrained to be implemented identical. This allows different designs of the modules, for example different processors, to mitigate bugs in the design through the so-called *design diversity* [Sor09, p. 20]. Spatial redundancy and design diversity are not limited to hardware, software-based implementations are also possible. An algorithm for example can be implemented twice in different programming languages and executed on two cores in parallel, with a comparison of both outputs. Spatial redundancy is further not required to be static, it is also possible to dynamically extend the redundancy of a module [Kor+07, p. 3].

An example for static hardware redundancy is a duplex system [Kor+07, pp. 27–28], which is also referred to as *dual-modular redundancy* (DMR). It consists of two modules that generate identical outputs in the error-free case, which then are verified by a comparator (see Figure 2.2a). In case of a mismatching output, an error is detected. However, without further mechanisms, this does not allow to detect which module is erroneous.

Additional error detection capability is enabled with *triple modular redundancy* (TMR), where three modules are compared [Sor09, p. 21]. The comparator acts as a voter, and selects the output that is generated at least twice. More enhanced redundant configurations are possible, like NMR, featuring N entities to compare, which enables to detect and correct more errors [Sor09, p. 22]. It is also possible to cascade spatial redundancy, for example by combining three DMR configurations [Kor+07, pp. 29–30] or three TMR systems with design-diverse processors, as it is commonly used in aerospace [Sor09, p. 21]. However, TMR and higher redundancy entails high cost for hardware and energy [Sor09, p. 22].

Temporal Redundancy

Computations can be performed redundantly in parallel, as described above, or consecutively. In this case, not the location of the redundant computation is different, but the time, which leads to *temporal redundancy* [Sor09, p. 22]. This is an effective mitigation of transient errors, since a transient fault will most likely not occur again on the repeated computation [Kor+07, p. 4]. The hardware cost for temporal redundancy is low, compared to spatial redundancy. However, the consecutive computation leads to twice the energy consumption and to additional latency [Kor+07, p. 4]. This negative performance impact can be reduced through pipelining, which can hide the latency of the redundant computation [Sor09, p. 22].

Information Redundancy

Redundancy is not only applied on computation to ensure correct output, but also on data to transmit and store it with the possibility to detect and correct errors through *information redundancy* [Kor+07, p. 55]. A prominent example is RAID (redundant array of independent disks), where the data is stored redundantly on more than one hard disk. This allows to tolerate the failure of one or more disks, depending on the configuration.

Depending on the error model, a full duplication of the data may not be required, since applied coding theory enables to increase fault tolerance through code words that contain redundant information. Through *error-detecting codes*, the data is extended with additional information to enable the detection of errors [Sor09, p. 22]. Parity bits for example signal an even or odd number of ones in a binary data representation, which enables to detect single flipped bits. However, error correction is not possible with parity, since the location of the flipped bit cannot be derived [Sor09, p. 23]. This can be confirmed by the *Hamming distance* that describes the differences between two code words with identical length [Kor+07, pp. 56–57]. The Hamming distance of a code is determined by the minimum distance between all possible code words. For a parity-based code, the Hamming distance is 2, which guarantees the detection of one flipped bit, but allows no correction [Kor+07, pp. 57–58]. In general, a Hamming distance of $k+1$ is required to detect k-bit errors, while for the correction of all k-bit errors, the Hamming distance needs to be at least $2k+1$ [Kor+07, p. 57]. Thus, a larger Hamming distance for a code enables the detection of more errors in a single code word and potentially allows to correct errors [Sor09, p. 23]. These *error-correcting codes* (ECC) require additional redundancy and thus allow the correction of one ore more errors. Often, error codes provide SECDED capabilities: single error correction and double error detection [Sor09, p. 24].

Cyclic Redundancy Check (CRC) codes are generated by polynomial division with a constant generator polynomial [Muk08, p. 178]. Since the resulting data is also a valid code word, this error detection code is classified as cyclic [Kor+07, p. 67]. CRC is commonly used to detect errors when transmitting data, for example in networks. Furthermore, CRC allows to detect transient errors, but additional redundancy is required for error correction [Muk08, p. 179]. To create a CRC checksum, the input data is divided by a generator polynomial, resulting in a remainder that is appended to the input data for transmission [Muk08, p. 179]. The calculation of the remainder is efficiently implementable in hardware, since polynomial division with a binary generator polynomial can be implemented with XOR gates and flip-flops [Muk08, p. 180]. On the receiving side, the transmitted data is divided by the generator polynomial again, where a remainder that is not zero signals an error. The error detection capabilities depend on the generator polynomial, which is mathematically designed to fit the required needs for detection of one or more single-bit errors and multi-bit errors up to a specific length. For an input data word with k bits a generator polynomial of degree $n-k$ is used to obtain an

encoded word of n bits. The resulting (n, k) cyclic code allows to detect multi-bit errors of $n - k$ consecutive bits [Kor+07, p. 68]. Commonly used generator polynomials are CRC-16 and CRC-32, of whose the latter is used for example in network communication. CRC-32 is a $(32 + k, k)$ cyclic code with $n = 32 + k$ to detect up to $n - k = 32$ consecutive erroneous bits in a block of up to $n = 2^{32} - 1$ bits [Kor+07, p. 73].

2.1.6 Error Detection with Lockstep Execution

An established mechanism that is used for decades in mainframe system is *lockstepping*, where the processors are duplicated to provide spatial redundancy in a DMR setting (see Section 2.1.5) [Sor09, pp. 29–30]. High reliability with error detection and the resulting high availability is not only required in mainframes for business applications, but also in mission-critical systems. Especially aircrafts suffer from high neutron flux due to high altitude, which results in higher error rates due to transient faults (see Section 2.1.2) and thus demand proper fault-tolerant mechanisms.

Lockstep processors execute the same instruction twice at the same time, and a *cycle-by-cycle* synchronization is required to ensure an identical state at each cycle in the error-free case [Muk08, p. 212]. In this *tightly-coupled* redundant execution, the comparator checks the outputs and the state of both processors continuously, and a divergence between the cores signals an error. A recovery mechanism is not part of the lockstep execution and has to be provided additionally. If allowed by the system requirements, a reset of the whole processor can be feasible, or otherwise, a checkpointing mechanism is needed to enable a rollback to an error-free state. Figure 2.3 depicts a lockstep system consisting of two processors with cycle-by-cycle execution. The outputs and the state of both processors is compared repeatedly after every cycle.

To ensure that divergences between the redundant processors occur only due to errors, the processors have to be completely deterministic to guarantee an identical state and output for same input [Muk08, p. 213]. Furthermore, a reset of the processors must lead to an identical state, for example in the caches and branch predictors, to ensure that both processors continue their execution identical. These requirements for determinism are not needed for commodity systems without fault tolerance, and thus lockstep processors cannot profit from the commonly used optimizations, which would entail indeterministic timing, e. g. out-of-order execution [Muk08, p. 214]. Additionally,

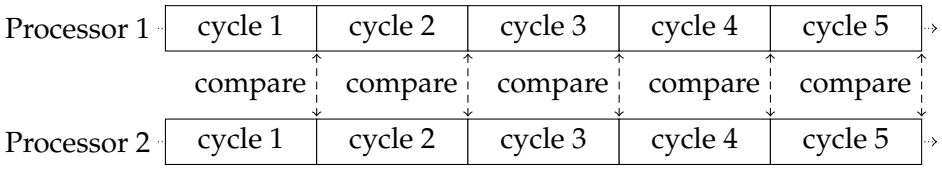


Figure 2.3: Tightly coupled lockstep execution

lockstep execution demands over twice the hardware, since not only the double amount of processors is needed, but also an additional comparator unit. This results in high cost of the overall system, when the price performance and also the performance-per-watt are compared.

The benefit of lockstep execution is the high fault coverage, where most errors can be detected due to spatial redundancy [Muk08, p. 213]. This includes errors due to permanent and intermittent faults, and also transient faults, since the chance of a fault to affect both processors is unlikely. Further, the redundancy is managed in hardware and thus is transparent to the software.

Lockstep processors can also be configured as triple-modular redundant system (TMR), where a voter replaces the comparator [Sor09, p. 30]. This entails the triple cost for hardware, but avoids a reset when at least two processors are in the same state and produce the same output. Spatial redundancy is also possible in multi-core processors, where usually the cores are coupled for lockstep execution instead of the whole processors [Sor09, p. 30].

2.1.7 Loosely-coupled Redundant Execution

With simultaneous multi-threading (SMT), thread-parallelism has been introduced in processor cores. The support for multiple threads can be leveraged for redundant execution, as proposed by Reinhardt et al. [Rei+00], with simultaneous redundant threads (SRT). In contrary to lockstep processors, where redundant processors are tightly coupled and compared on a cycle-by-cycle basis, this approach resembles a *loosely-coupled* redundant execution [Muk08, p. 212]. The benefit of relaxing the tight coupling allows to disregard the individual micro-architectural states, and comparison can be performed on a coarser level like committed instructions [Muk08, p. 222]. Cache misses and branch mispredictions, which are critical in tightly-coupled lockstep execution, can be tolerated, since the resulting timing divergence does not impair the comparator. With SRT, a full duplication of the processor hardware is not necessary, since error detection in individual parts can also be provided through spatial or temporal redundancy [Rei+00]. However, the potentially diverging execution of the redundant threads entails challenges in the distribution of the inputs and the comparison of the outputs.

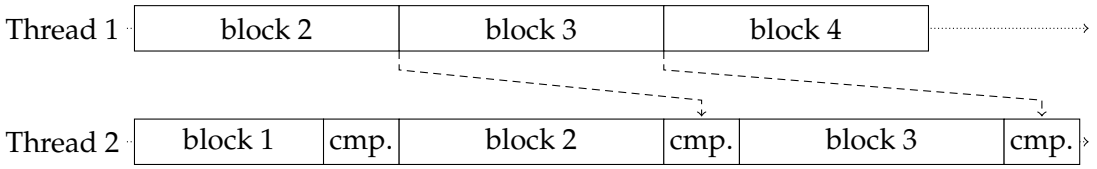


Figure 2.4: Loosely coupled comparison

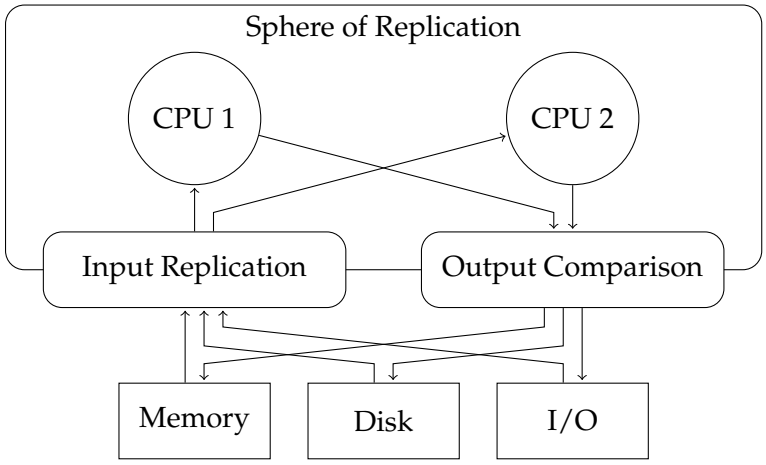


Figure 2.5: Sphere of Replication [Muk08, p. 209]

The principle of SRT can be generalized as *redundant multi-threading* (RMT), since it can also be implemented purely in software [Muk08, p. 301]. The granularity of software redundancy can be even more coarse-grained than for hardware SRT, since it is sufficient to compare only the output data that leaves the redundant threads, for example by means of a system call [Muk08, p. 302]. At the boundaries of this *sphere of replication*, the output data is compared and checked for errors, and the input data is distributed to the redundant threads. RMT is usually implemented by duplicating the thread inside the process context, but separation into two individual processes is also feasible.

Figure 2.4 describes the idea of RMT: two threads execute a sequence of blocks, which consist of one or more instructions, delimited by the boundaries for output comparison and input replication. Due to the loose coupling, the redundant threads are not required to be executed synchronously. The output comparison can be done in the second thread only, for example, which effectively leads to one *leading thread* and one *trailing thread*, since the latter executes its blocks after the other thread. A benefit of this shifted execution is a combination of spatial and temporal redundancy, which further reduces the chance of an error affecting both redundant threads simultaneously.

2.1.8 Sphere of Replication

With the *sphere of replication* (SoR), the domain is defined in which errors can be detected through redundancy [Muk08, pp. 208–210]. For example, this can be a pair of processors coupled as a lockstep pair, where all other parts of the system that are not redundant are outside of the sphere of replication. The concept of the sphere of replication was introduced bei Reinhardt et al. with their proposal for RMT (see Section 2.1.7).

The redundant parts require to receive the same input data to be able to produce identical output. This is enabled with the *input replication*, where the instructions and

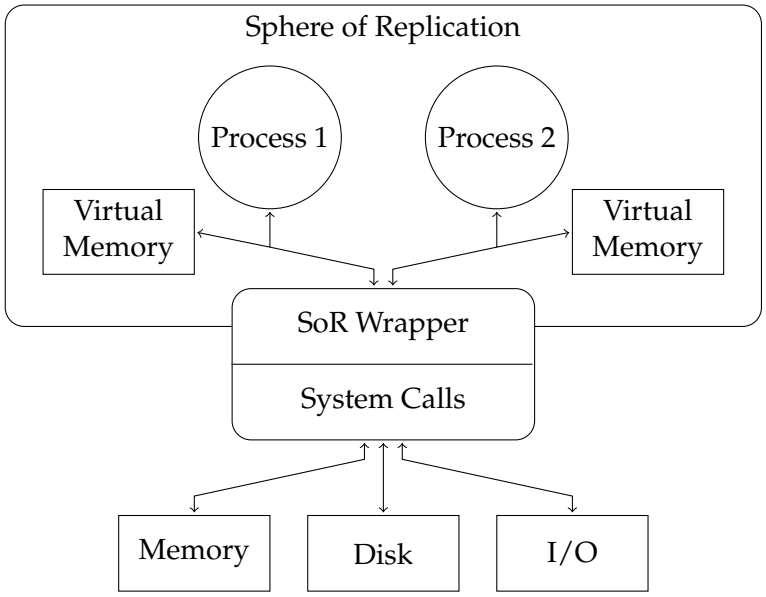


Figure 2.6: Sphere of Replication for redundant processes, based on the SoR for an RMT system [Muk08, p. 303].

the operands are distributed to all parts inside the sphere of replication. A correct duplication and distribution of the input data is crucial to prevent false errors due to diverging paths in the redundant parts [Muk08, p. 209]. Further, everything that leaves the sphere of replication must be guaranteed to be error-free, which is ensured by the *output comparison* unit. There, the outputs are gathered from the individual redundant parts, compared to detect errors, and then forwarded to the appropriate components outside of the sphere of replication.

Figure 2.5 gives an overview of the sphere of replication for a redundant set of processors. Both receive their input from the input replication unit that distributes the requested data from memory, disks, or other devices. Before the output data is forwarded to the appropriate device, the data from both processors is compared to detect errors.

Input replication is easier for cycle-by-cycle lockstep processors, which are only spatial redundant, since the redundant processors run synchronously. Because the redundant threads may be at different positions in the execution, loosely-coupled configurations like RMT require more elaborate input replication mechanisms. This requires to cache the input data to provide the exact same data to the trailing thread if the leading thread writes to the corresponding memory location in between [Muk08, pp. 232–236]. Special consideration is needed for interrupts and exceptions, since a different handling of those in both redundant threads leads to diverging execution paths. Output comparison for loosely-coupled redundancy requires to cache data from stores to prevent writing to

memory before the data is compared with the output of the trailing thread to detect errors [Muk08, pp. 230–231].

For software-based redundancy approaches with redundant processes, a sphere of replication can be designed as depicted in Figure 2.6. Each process has its own virtual memory with heap and stack, which is inside the sphere of replication since it is redundant, too. Communication with the outside is managed with a SoR wrapper that handles system calls and provides the input data to both processes. The wrapper further has the task to check the output data that leaves the sphere of replication through system calls.

2.2 Multi-Threading in Shared-Memory Systems

With the dawn of multi-core processors in the early 2000s as a way to make use of the increasing amount of transistors per chip, and to bypass the power wall (see Chapter 1), parallel programming techniques became important also in the small scale, like individual servers or personal computers [Hen+17, p. 368]. The main objective is to spread the workload of a program across multiple threads to maximize the utilization of the available hardware. Two widely used categories exist, which are based on the type of decomposition: data parallelism and task parallelism [Hen+17, p. 11]. For multi-processor and multi-core systems, task parallelism is the preferred method to decompose a given workload.

Multiple programming paradigms for task-parallel computing exist, which can be separated into *memory-coupled* and *message-passing*, depending on whether the communication is explicit, as for message-passing, or implicit through synchronized accesses on the data in memory-coupled *shared-memory* systems [Hen+17, p. 373]. Multi-core processors form such a shared-memory system, since the same memory is accessible from all cores. Thus, parallel applications for multi-core processors are usually programmed *multi-threaded*, where all threads of the application process access the same memory. The POSIX standard contains the definition of Pthreads, to which the common thread implementations in UNIX and GNU/Linuxbased systems adhere [Ste+08, p. 356].

2.2.1 Synchronization with Mutual Exclusion

To avoid conflicts and to guarantee consistent data in concurrent programs, appropriate synchronization and mutual exclusion is required for accessing data that is shared between multiple threads. Mutual exclusion is often used to guarantee atomic access on data structures. A correct implementation of the program assumed, this ensures a correct execution of the parallel program, and avoids race conditions [Her+08, p. 200]. However, all accesses to such protected data are serialized, causing potential bottlenecks. Figure 2.7 shows two threads, which lock the same mutex to enter a critical section. Since the data structure or the algorithm used in the threads requires to atomically execute the critical

section, the other thread has to wait until the mutex is unlocked. The resulting execution is a serialization of the critical sections, independent if conflicts exist in between these critical sections, or not. It may become observable that the speed-up of the parallel program is lower than expected, which leads to a low overall performance. This is due to the pessimistic locking, which would not be necessary under all circumstances. An example is a linked list with one producer and one consumer, where mutual exclusion would only be required when head and tail of the list collide.

To ease the programming, whole data structures are often protected by a single mutex only. In the example of the linked list, the single mutex is locked for every access to the list, which prevents parallel reads and writes on different ends of the list. This can be improved with *fine-grained* locking, where programmability can be traded against performance [Her+08, pp. 201–202]. Rather large locked regions, which are more easy to program, are transformed into multiple small regions that are protected by one or more individual mutexes. For the linked list, each node has its individual mutex to protect the pointer to the next node. Before modifying the linked list, all mutexes of all involved nodes need to be locked first. The need to lock multiple mutexes for a single critical region to guarantee the correct exclusion of other threads increases the complexity and thus leads to a higher chance of race conditions and deadlocks.

2.2.2 Lock-free Data Structures

An alternative to avoid these problems is a *lock-free* implementation of algorithms and data structures, which do not require mutual exclusion, and rely only on atomic memory operations like *compare and swap* [Her+93]. With knowledge of the ordering of memory accesses and correctly designed methods for the operations on the data structure, no locks or other synchronization mechanisms are required to guarantee consistency [Her+93]. Lock-free implementations of various data structures exist, also for the mentioned linked list [Val95]. However, the guaranteed freedom of deadlocks through lock-free implementations often entails lower performance, or strongly increased complexity, or, probably in most cases, both. Additionally, not everything can be implemented lock-free, and particularly special hardware support of the memory system is required, e. g. compare-

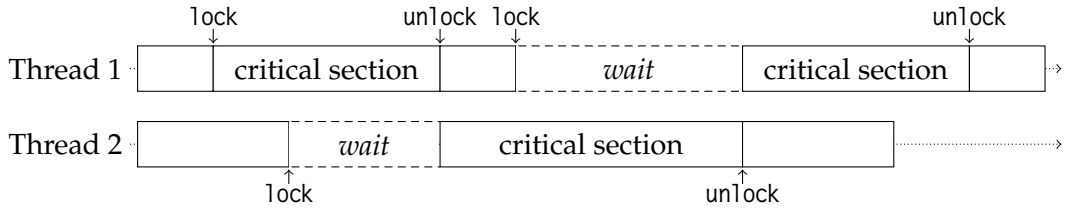


Figure 2.7: Two threads executing a critical section protected by a mutex.

and-swap that takes two pointers simultaneously and atomically, which is required for double linked lists [Her+93].

2.3 Hardware Transactional Memory

Transactional memory (TM) offers the capability to access and modify large data structures atomically [Har+10, p. 6]. As an optimistic synchronization mechanism, transactional reads and writes on shared data are executed immediately and without mutual exclusion. With the assumption that reads on the shared data occur more often than writes, conflicts where reads and writes or multiple writes happen at the same time are rare. Such conflicts can be detected by tracking all read and write accesses during a transaction. Conflicts are resolved by forcing one or more concurrent transactions to abort, which restores the state of the corresponding threads at the beginning of the transaction [Har+10, p. 8].

In Figure 2.8, the same two threads of Figure 2.7 enter a critical section optimistically, i. e. without locking a mutex. The critical section is executed transactionally, so that the transactional memory system detects conflicts and resolves them by aborting one of the transactions. In the common case, where no conflicts occur, both threads are executed in parallel without any serialization. This results in a shorter overall execution time and thus leads to an higher utilization of the available processing power.

In 1977, Lomet described *atomic actions* for multi-process programs to provide process isolation and synchronization [Lom77]. This is regarded as the first publication in which the idea of database transactions has been adopted to parallel programs [Har+10, p. 6]. *Transactional memory* was introduced by Herlihy and Moss in 1993, with the proposal of an hardware implementation that extends the cache-coherence protocol [Her+93]. The intended main purpose is to enable lock-free implementations of data structures that would either be complex and intricately to program, or not implementable at all. Their often-cited example is a double-linked list, where the `head` and `tail` pointer require to be updated simultaneously in certain situations [Her+93]. The required atomicity of the whole list update is provided by the transactional memory, and is easily usable by

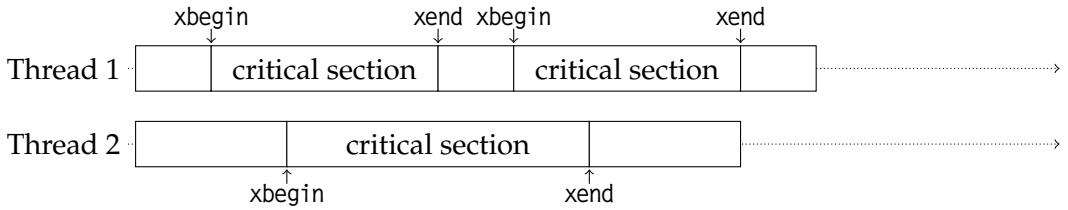


Figure 2.8: Optimistic synchronization of a critical section in two threads with transactional memory.

programmers. Since then, various software implementations (STM) have been proposed, but hardware implementations in commercial processors did not become available until recent years.

Hardware implementations of transactional memory (HTM) have been developed for the Sun Microsystems Rock SPARC multi-core processor [Dic+09], which has never been released. The IBM supercomputer processor BlueGene/Q [Wan+12], and the IBM POWER8 architecture [Cai+13] both support transactional memory. AMD developed the *Advanced Synchronization Facility* [Chr+10] to implement transactional memory, although it is not yet available in any AMD microprocessor. With the Haswell micro-architecture, Intel released the *Transactional Synchronization Extensions* (TSX) [Ham+14]. Hence, many architectures and processors for server systems and personal computers support transactional execution of sequences of instructions. The particular implementations differ on both the instruction-set level and on the micro-architectural level, since different mechanisms can be implemented to create checkpoints and to track changes within transactions.

2.3.1 Version Management

For the optimistic execution of transactions, multiple versions of the data that is read and written within a transaction have to be managed to ensure atomicity and to enable the rollback in case of an abort. A transaction may overwrite data at a specific address, but this must not be visible to the rest of the system before the transaction committed successfully. If a conflict occurs, the overwritten data must be restored [Har+10, pp. 21–22].

A transactional memory system can feature either an *eager* or a *lazy versioning* mechanism. The former is often implemented in STMs, where data is updated directly, but the old data is kept in a log for potential restoration [Har+10, p. 21]. In case of a transaction abort, the data is restored with the backup kept in the log. Hardware TMs usually keep changes during a transaction in a private cache and the data becomes visible only at the transaction commit, which is referred to as lazy versioning [Har+10, pp. 21–22]. If an abort occurs in such an HTM, the buffered data changes are simply discarded. However, the data in hardware transactions can be in memory or in registers, but the transaction cache for lazy versioning is only applicable for data stored in memory, not for register values. Thus, actual hardware implementations implement a combined approach, where the registers are backed up eagerly, which can be implemented through shadow registers, or by appropriate renaming of the physical registers [Har+10, p. 153].

2.3.2 Conflict Detection and Resolution

A transaction is executed isolated, and the memory modifications that were done within need to be *committed* atomically at the end of the transaction. However, a transaction is

allowed to commit only if there is no conflict with any other thread. A conflict is, for example, if a memory location is read within one transaction and written to in another, or if both transactions write to the same memory location. Such events have to be detected before the modified data of a transaction becomes visible to other threads on the transaction commit.

Depending on the version management, STMs need to check for conflicts immediately before modifying the affected memory location, featuring a *pessimistic concurrency control* [Har+10, p. 20]. To fully exploit the potential performance, HTMs implement *optimistic concurrency control*, where conflicts have to be detected before transaction commit, but not immediately when the corresponding data is being modified [Har+10, p. 20].

A TM systems need to track the data that is read and written inside a transaction to detect conflicts, and to backup old versions of data, if demanded by the applied version management. Addresses of memory locations where data is read are put into the *read-set*, and the addresses of writes are collected in the *write-set* [Har+10, p. 17]. Transactions in HTMs in particular are limited in their size, which depends on the capacity of the read- and write-sets. That capacity is influenced by the structure used to store the sets, which can be for example a full-associative cache, or a set-associative. Another characteristic that influences the capacity of the sets is the granularity, which for HTMs is commonly identical to the cache line size [Har+10, p. 22].

Optimistic concurrency control can feature *lazy conflict detection*, where the read- and write-sets are compared at the end of the transaction in the commit phase [Har+10, p. 22]. With this method, which is commonly used in STMs, only conflicts with other transactions can be detected, since non-transactional memory accesses are not recorded in a read- or write-set. Alternatively, *eager conflict detection* enables the TM system to detect conflicts earlier [Har+10, p. 22], which is the preferred method in HTMs since it allows to combine conflict detection with cache coherence. The read- and write-sets in HTMs are usually hold in the caches, and the granularity for conflict detection matches the cache line size, thus allowing an effective conflict detection by snooping on the cache coherence messages.

2.3.3 Intel Transactional Synchronization Extensions

A hardware transactional memory implementation that is available in commercial off-the-shelf (COTS) processors for servers and personal computers is the *Restricted Transactional Memory* (RTM), which is part of Intel's Transactional Synchronization Extensions (TSX) [Ham+14]. TSX is available since the "Haswell" generation of the Core architecture, and provides an extension to the instruction set architecture to interface with the transactional memory system.

The write-set is hold in the L1 cache, and transactional reads are tracked in the L1 and L2 cache, with a granularity of full cache-lines [Ham+14]. Data of uncommitted

transactions is hold back in the cache and is not visible to others, resembling a lazy versioning. The conflict detection is implemented through the cache coherence protocol and thus is eager [Ham+14].

The optimistic synchronization of TSX improves the performance of parallel programs, compared to both lock-based and fine-grained locking, as the benchmarks results of the performance evaluation of Yoo et al. shows [Yoo+13].

Defining Transactions

A transaction is defined by surrounding the desired code fragment with the RTM intrinsics `_xbegin` and `_xend` [Int13a, pp. 24–26]. These intrinsics encapsulate x86 instructions with the same name, but provide a return value instead of jumping to a relative target, which better fits the C/C++ programming style.

The intrinsic `_xbegin` starts a transaction and implicitly creates a checkpoint of the processor state for a potential rollback in case of a transaction abort [Int13d, pp. 14–16]. All following memory accesses are tracked by the read- and write-set, and memory modifications are hold back in the L1 cache, hidden from other cores. With `_xend`, a transaction commit is initiated, which ends the transaction if no conflict is detected [Int13d, pp. 17–18]. When the transaction commits, the modified cache lines become visible to the other cores.

Conflict Detection and Resolution

Memory accesses inside of other transactions and also non-transactional memory operations can lead to conflicts, which are eagerly detected. False conflicts may be detected due to the coarse granularity of the size of a full cache line. In case of a conflict, the transaction gets aborted, and the processor status of the beginning of the transaction is restored. Modified cache lines of the transaction simply are dropped, and the registers are rolled back to the checkpoint. Conflicts immediately lead to an abort of one or more of the affected transactions, an intervention to control the conflict resolution, e. g. through priorities, is not possible.

Other reasons beside real conflicts for a transaction abort exist, like an overflowing cache where the read- or write-set can no longer be hold in the cache. Exceptions and interrupts also lead to an abort, as well as other events in the processor, which are not further defined by Intel.

Fallback Mechanism

The behavior of RTM in case of a transaction abort is independent of the cause of the abort, the execution always continues after the `_xbegin` intrinsic. The return code of `_xbegin` can be used to determine if a transaction has started, and thus the following code

Listing 2.1 Example of a RTM transaction wrapper for a critical section

```
1  int i = MAX_TRIES;                                  ▷ max. number of TX starts
2
3  for (; i>0; --i) {                                  ▷ repeat up to MAX_TRIES times
4      if (_xbegin() == _XBEGIN_STARTED) {             ▷ start TX, check if aborted
5          if (is_locked(mutex)) {                     ▷ put mutex into read-set
6                                                      ▷  and check if locked
7              _xabort(0xFF);                          ▷ abort transaction
8          }
9          break;                                      ▷ successfully in transaction
10     }
11 }
12 if (i == 0) {                                       ▷ TX was never successful
13     lock(mutex);                                    ▷ lock fallback mutex
14 }
15
16 // ...                                              ▷ critical section
17
18 if (i > 0) {                                        ▷ if inside transaction
19     _xend();                                        ▷ end transaction
20 } else {
21     unlock(mutex);                                  ▷ otherwise unlock mutex
22 }
```

is executed transactional, or if the transaction had been aborted. If the return code is not equal to a defined constant, the transaction has been aborted, and a specified range of bits in the return code allows to read the reason for the abort.

Due to the various reasons for a transaction to abort, a transaction cannot be guaranteed to commit successfully after a given number of tries. A fallback mechanism is required to avoid an infinite loop of starts and aborts of the same transaction. Based on the abort reason, a transaction can be started a fixed number of times, and after that, or depending on the abort reason, a *fallback path* can be executed. There, a traditional mutex can be locked before executing the critical section non-transactional.

Example

Listing 2.1 shows a basic implementation of a RTM wrapper around a critical section, oriented at the example given in [Int13a, p. 11]. Since it is not guaranteed that a transaction eventually commits, a maximum number of tries is specified by `MAX_TRIES` in Line 1.

In a loop (lines 3 – 11), a transaction is started in Line 4 with the intrinsic `_xbegin`, which returns the constant `_XBEGIN_STARTED` when the transaction was not aborted, and the `break` ends the loop (Line 9). If the transaction aborts, the execution still continues at the instruction after the `_xbegin`, but due to the different return status, the `if` block is not taken. In the given example, the reason for the transaction abort is ignored and it is simply retried to start the transaction in the next iteration of the loop.

It is important to include the fallback mutex into the read-set to be able to detect a conflict if one thread executes the fallback path and another thread executes a transaction. Checking the mutex inside of the transaction includes it into the read-set, and allows to explicitly abort the transaction if it is locked (Lines 5 and 7). In case the mutex gets locked later, while the transaction is still running, this will be detected as conflict, and the transaction will abort.

After the for-loop, if the counter is zero, the transaction was not able to be executed, and the fallback mutex has to be locked (lines 12 – 14). After this, it is guaranteed that the following critical section in Line 16 is either executed transactionally or protected by a mutex.

To end the transaction, the intrinsic `_xend` is called, but only if the transaction has been started (Line 19). Otherwise, the mutex has to be unlocked, as listed in Line 21.

2.4 Software Fault Tolerance with Hardware Support for Error Detection and Recovery

Alternatively to the previously described hardware approaches for fault tolerance, like lockstep and RMT, error detection and recovery can be provided on the higher software level as well. This removes the need for custom hardware and allows to execute applications with dependability requirements on common COTS hardware. However, hardware extensions that exist on the target platform can be leveraged to provide parts of the required functionality, and to speed up error detection and recovery.

The combination of transactional memory, which originally was intended to support efficient programming of parallel applications, with software-managed redundant execution offers a promising mechanism to provide fault-tolerance for individual applications on existing hardware. The utilization of hardware transactional memory in embedded systems has been suggested to provide efficient concurrency control and to implement error recovery mechanisms [Fet+11]. The demand for high performance, combined with the necessity to save energy in mobile and embedded devices, draws the attention to multi-cores also in the embedded domain.

2.4.1 Transactional Memory in Dependable Systems

The optimistic concurrency control of HTMs is beneficial, as it scales better than lock-based mutual exclusion for an increasing number of cores. Apparently, HTMs are designed to provide a good average performance, but no guarantees for a successful transactional execution are given. For example Intel TSX, as described in Section 2.3.3, requires a fallback handler, where mutual exclusion for the critical section is ensured by traditional locks. To satisfy the real-time requirements of embedded systems, the TM system needs to control the execution of transactions by means of a contention manager to respect deadlines and priorities [Met+13].

Transactional memory can also support fault tolerance mechanisms, implemented either in hardware or software. Fetzer and Felber encourage the use of the checkpointing and rollback mechanism to efficiently rollback and recover when errors are detected [Fet+11]. Since the aforementioned real-time requirements need only to be satisfied for embedded systems, existing TMs can be leveraged for example in server systems to provide hardware support for software-managed fault tolerance. This allows to build a dependable system out of COTS processors, where redundancy is managed in software for individual applications to facilitate fault tolerance.

2.4.2 Transaction Checkpoints for Error Recovery

HTMs like Intel TSX implement lazy versioning, where the modifications that are made inside of a transaction are kept in a private cache. Further, the processor state is backed up, containing the contents of the architectural registers. In case of a transaction abort, the processor state is restored and all memory modifications from within the transaction are discarded. That checkpointing mechanism can be leveraged for error recovery, since it contains the complete state of the processor, and no writes to the memory can leave the transaction without commit.

However, to provide checkpoints for an application, it is required to execute the given program fully covered with transactions, since a transaction can only be rolled back before it is committed. The length of the transactions, in which the program is wrapped, depends on the TM system and its limits on the read- and write-sets, and on calls that leave the sphere of replication, e. g. system calls.

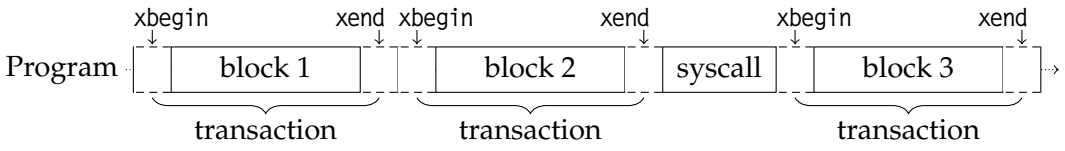


Figure 2.9: Transactional wrapping of a program to provide checkpoints [Fet+11].

An example is shown in Figure 2.9, where a program is divided into subsequent blocks, which each are wrapped into a transaction. At every `_xbegin`, a transaction is started, and thus a checkpoint is created. The system call is not wrapped into a transaction, since this call is outside of the sphere of replication.

2.5 The LLVM Compilation Framework

The fault tolerance approach described in the subsequent chapter relies on program instrumentation, where a given program is enhanced with the error detection functionality by inserting additional instructions. One feasible method for this is the modification of the program during compilation. The LLVM compilation framework supports such modifications in an elegant manner, and thus is the preferred tool in the presented approach. The primary objective of LLVM, the Low-Level Virtual Machine, is the "transparent, life-long program analysis and transformation for arbitrary programs" [Lat+04]. Based on a low-level representation of the program code, transformations are possible at different stages and levels during the compilation process.

The process of compilation is depicted in Figure 2.10: A language-specific front-end compiles the input source files into the abstract LLVM intermediate representation (IR). The typical front-end is `clang` for C programs, and `clang++` for C++. Multiple steps of optimization can be applied upon the IR, the typical `-O` parameter for example selects different optimizations to be performed. The resulting optimized IR can then be compiled into machine-dependent assembly with the `llc` command. The final linking is platform-specific and is thus done with GCC on GNU/Linux systems. Alternatively to assembling and linking the program into target-specific machine-code, the IR can also be executed directly by the IR interpreter `lli`, which uses just-in-time compilation.

The whole compilation and optimization process can be also executed through a single call of `clang`, similar to GCC, where the tools for the appropriate tasks are called

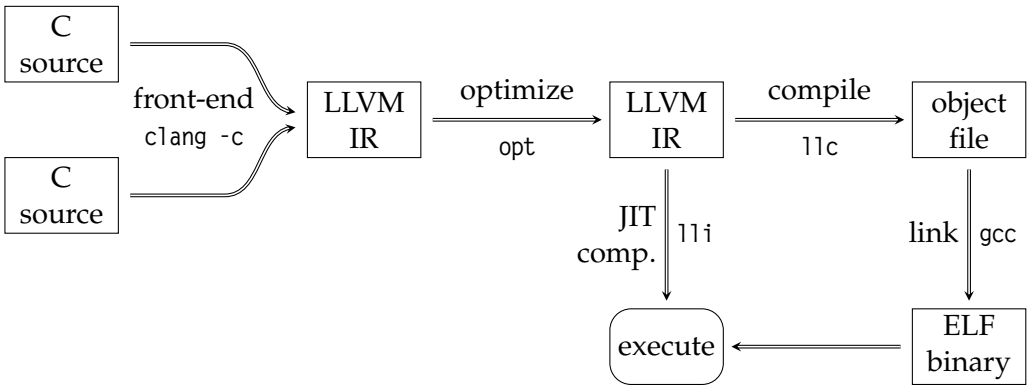


Figure 2.10: The process from source files to execution in the LLVM toolchain.

internally. Options to LLVM and the optimizations passes, for example, can be passed to the front-end, and then are available for the designated tool.

FORTRAN programs can be compiled by LLVM using the *flang* front-end [Sca19]. An alternative compiler is *F18*, of which the integration with LLVM is currently under development [NVI19]. Further languages can easily be supported by LLVM if a front-end exists.

2.5.1 LLVM Intermediate Representation

The essential concept of LLVM is the Intermediate Representation (IR), a low-level, assembly-like representation of a program. It is independent of the source language, but allows to represent high-level information of the input program, which is valuable for optimization. The IR has a *static single assignment* (SSA) characteristic, which allows a *value* to be assigned only once. An instruction that takes a value as argument is a *user* of this value, and the dependency is defined by the *use*. The values are virtual registers, whose number is unlimited, in contrary to actual instruction set architectures, and a value name is unique, thus it cannot be referenced by another value. The concrete association of values to machine registers and stack variables is done by the back-end during the compilation into machine-specific binary code. Further, a value can only be assigned once, and changing a value requires the assignment of a new value. Because of the lack of a stack, data can only be stored in registers or in the memory, and transfers in between are handled by the `load` and `store` operations. Each value has a type, which cannot be changed, except through an explicit cast in the assignment to another value, which thus entails a type system independent of the source language.

A function consists of basic blocks, which itself are sequences of instructions. For each basic block, the successor and all predecessors are known statically, thus providing a control flow graph of the function. Programming in the IR by hand appears complicated, but the limited number of opcodes, the strong typing, and the restrictions of the SSA allow high-level optimizations, due to the reduced complexity of the dependency graph, which would not be possible on real machine code.

Example

Listing 2.2 shows the main function of a C program, where a variable `x` is initialized with the number 10. A function `a()` is called, and the return value is added to `x`. After the number 20 is added to `x`, it is returned at the end of the function.

The result of the compilation of the function in Listing 2.2 into the LLVM IR is shown in Listing 2.3. After the definition of the main function, memory is allocated for the variable `x`, and the allocated memory location is assigned to the value `%1` (see Line 2). The size of the memory location is specified by the type, which is a 32 bit integer. The initialization of `x` is done by the store in Line 3, where the number 10 with the type of a

Listing 2.2 Example C Program

```
1 int main() {
2     int x = 10;
3     x += a();
4     x += 20;
5     return x;
6 }
```

Listing 2.3 LLVM Intermediate Representation

```
1 define i32 @main() #0 {
2     %1 = alloca i32, align 4                          ▷ x
3     store i32 10, i32* %1, align 4                    ▷ x = 10
4     %2 = call i32 @a()                                ▷ a()
5     %3 = load i32, i32* %1, align 4                   ▷ 10
6     %4 = add nsw i32 %3, %2                           ▷ 10 + a()
7     %5 = add nsw i32 %4, 20                           ▷ 10 + a() + 20
8     store i32 %5, i32* %1, align 4                    ▷ x = 10 + a() + 20
9     ret i32 %5
10 }
```

32 bit integer is written to the memory location specified by the value `%1`. A function call is represented by a `call` instruction with a specified return type and a list of parameters, which is empty in this example. The return value of a call is assigned to value `%2` in Line 4. To increment the variable `x` by the number that has been returned by the function `a()`, a load from the memory location of `x` is performed, and a new value is created as the result of the addition of `x` in value `%3` and the result of `a()` in value `%2` (see Line 6). To this value, the number 20 is added (Line 4 in Listing 2.2 and Line 7 in Listing 2.3), resulting in the value `%5`, which then is stored into the memory location of the variable `x`, where the value `%1` points to. Finally, the function returns the value `%5`, which holds the result of the last addition (Line 9). In contrary to the C example in Listing 2.2, the variable `x` is allocated in memory, since a stack does not exist in the IR. However, local allocations in functions are translated into stack variables by the back-end, depending on the architecture of the target machine.

2.5.2 Program Instrumentation with an LLVM Optimization Pass

With an optimization pass, the IR of a program cannot only be transformed to optimize the code to increase the performance on execution, but also modifications are possible that enhance the functionality of the given program. Optimization passes can be loaded by the optimization tool at runtime, and thus they can be compiled separately of the whole LLVM toolchain by including the appropriate header files. The custom module can be called on different levels, depending on the inheritance of the module base class. This allows to implement modules that work on individual basic blocks, on functions, or on whole compilation units.

An alternative to the instrumentation on the IR during the program compilation is binary instrumentation. PEBIL [Lau+10] is a tool for binary instrumentation that allows to modify and insert x86 assembly instructions in a binary. This tool had been used in preliminary work to insert signature generation instructions [Haa+17], but the move into the compilation phase and the implementation of the LLVM optimization pass allowed to utilize further optimization passes before the program is compiled into machine code.

2.6 Summary

Lockstep processors provide an established mechanism to detect transient errors due to spatial redundancy, and stored data can be protected from errors by information redundancy methods like checksums. Redundancy approaches for processor or cores exist in hardware and in software, and hybrid approaches are also feasible.

Hardware implementations of transactional memory enable optimistic synchronization, thereby increasing the processor utilization of multi-threaded programs. The checkpointing mechanism of TM, which is required to rollback in case of transaction con-

flicts, can be leveraged in a software-based fault-tolerance approach to provide an error recovery facility. This allows a redundant and fault-tolerant execution of applications on a COTS processor.

A remaining challenge is loosely-coupled redundancy for multi-threaded programs, since the synchronization mechanisms and the resulting mutual exclusion leads to an indeterministic interleaving of the executed threads. Such diverging execution in the redundant duplicates is not feasible on hardware lockstep processors, and software mechanisms need to provide methods for scalable synchronization.

3

Related Work

Contents

Fault tolerance mechanisms have been implemented on different hardware, with varying methods for error detection and correction. This chapter provides a brief overview of related work to the approach presented in the next chapter. First, Section 3.1 describes hardware-based redundancy approaches for SMT and multi-core processors. In Section 3.2, methods that are based only on software to provide a redundant execution to detect errors are discussed. Section 3.3 introduces alternative approaches that leverage transactional memory. The redundant execution of multi-threaded programs is discussed in Section 3.4. At the end of this chapter in Section 3.6, the presented approaches are summarized and compared to the proposed technique of this theses.

3.1 Hardware Redundancy Approaches

Multiple redundancy-based approaches for fault tolerance have been proposed in the last decades, with varying flexibility compared to traditional cycle-by-cycle lockstep execution, where two processors are tightly coupled. These traditional lockstep processors resemble a dual-modular redundancy configuration (DMR), as described in Section 2.1.5. Figure 3.1 shows a lockstep system consisting of two processors and duplicated memory. The whole system is inside the sphere of replication, i. e. it is fully duplicated, and a comparator ensures the identical execution in both processors, which typically requires a cycle-by-cycle execution. In case of a detected error, the whole lockstep processor stops, resembling a fail-stop system.

The loosening of the tight coupling in the traditional lockstep systems is a general objective in the research of hardware redundancy. Most of the micro-architectural enhancements of the past decades are not available in cycle-by-cycle lockstep systems, due to the increasing indeterminism and the vast complexity, which renders the full comparison of two redundant processors practically impossible.

3.1.1 NonStop Advanced Architecture

Bernick et al. list multiple of these development trends, for example non-deterministic behavior due to asynchronous events, power management and variable clock frequencies, increasing soft-error rates, and chip multi-processors [Ber+05]. They introduce the NonStop Advanced Architecture (NSAA), which is an improvement of the high-availability NonStop system of HP [Ber+05]. The tight coupling of redundant processors is loosened, which allows the cores to run the identical applications with different clock rates. The NSAA consists of two or three four-way SMT Itanium processors to provide a DMR or TMR setup for four logical processors. Thus, a logical processor consists of single processing elements of two or three SMT processors, which are compared by the logical synchronization unit before an output to I/O devices or other processors occurs. With a TMR configuration, the logical synchronization unit acts as a voter and selects the correct output data, while DMR leads to an immediate stop of the logical processor.

3.1.2 Simultaneous and Redundantly Threaded Processor

AR-SMT [Rot99] is based on time-redundant Simultaneous Multi-Threading (SMT), since the author arguments that micro-architectural approaches for fault tolerance are required to provide a cost-efficient solution with good performance against transient errors, which are increasing in microprocessors due to rising clock-rates. A similar approach is the Simultaneous and Redundantly Threaded Processor (SRT), which is also derived from the Simultaneous Multi-Threading mechanism, and executes a thread redundantly and simultaneously as two independent hardware threads [Rei+00]. SMT

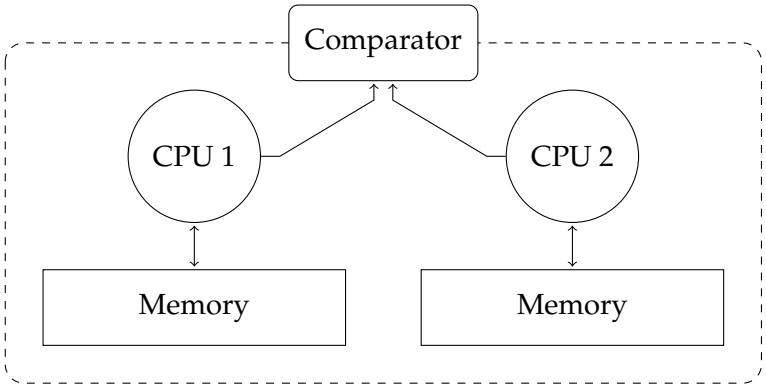


Figure 3.1: A comparator ensures the identical execution on both CPUs of a lockstep processor.

leverages the unused resources of a CPU to issue another thread to the free execution units to increase the overall processor utilization. Executing two redundant threads on an SMT processor increases resource conflicts, as both threads require the same resources. Dynamic scheduling of the threads loosens the tight coupling and thus reduces the resource conflicts, but necessitates a sphere of replication for input replication and output comparison. The performance of this approach is enhanced by inserting a slack between one thread and the redundant counterpart to provoke a leading and a trailing thread, of whose the latter benefits from resolved cache misses and calculated branch outcomes. In contrast to AR-SMT, the SRT approach does not include the main memory in the sphere of replication, which impacts the cache capacity of the redundant threads. Further, AR-SMT does not feature input replication and output comparison, and requires the comparison of every redundantly executed instruction for error detection. A recovery mechanism for SRT, which is capable of detecting transient errors, has been proposed with SRTR [Vij+02]. This approach requires to check the instruction in the trailing thread before the leading thread commits to avoid the output of erroneous data, since instructions can only be re-executed by means of a pipeline rollback. Thus, the leading instructions need to be verified between completion and commit. A register value queue is used to store instruction results that are needed for comparison, and a dependency analysis reduced the number of required checks. Instead of branch outcomes, the trailing thread in SRTR can only use the potentially corrected branch predictions of the leading thread, due to the narrow slack.

3.1.3 Dynamic Core Coupling

An approach for chip multi-processors, where pairs of cores execute the same instructions redundantly, instead of redundant threads on an SMT processor, has been proposed with Dynamic Core Coupling (DCC) [LaF+07]. Pairs of cores can by dynamically cou-

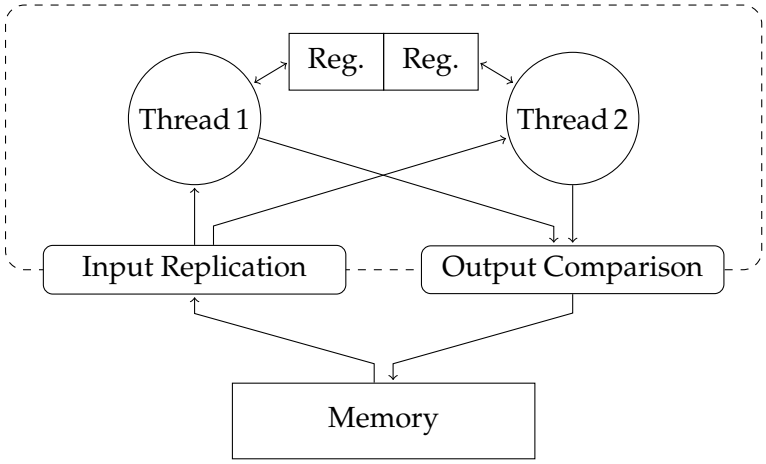


Figure 3.2: Redundant threads use a split register set, which is inside the sphere of replication [Muk08, p. 209].

pled into a DMR scheme for mutual verification, which enables to re-pair cores in case of permanent errors, and allows an on-demand triple-modular redundant (TMR) configuration. Checkpoints of the architectural state of both redundant cores are requested by the system bus to be taken by the cores, which then are compared to detect errors. In case of a differing redundant state, the cores can be rolled back to a previous checkpoint, and the execution can continue from this point in an error-free state. The cores are coupled over the system bus, and a modified private cache together with the customized cache coherence protocol is used to control the memory access and cache coherency actions of the redundant cores. The coupling through the system bus works well for low numbers of cores in the CMP, but it is not a scalable solution for large numbers of cores due to the increasing complexity of the interconnect.

3.2 Software-based Redundancy

An alternative to fault tolerance implementations directly in the processor are software-only redundancy mechanisms, which are independent of the underlying hardware.

3.2.1 Error Detection by Duplicated Instructions

An early software-only method for error detection is EDDI, *Error Detection by Duplicated Instructions* [Oh+02], where the instructions of a program are duplicated during compilation. These duplicated instructions work on a different set of registers and with different memory addresses, resulting in a time-redundant execution in a single hardware thread, in contrary to the redundant threads of the SRT hardware approach. At synchronization points, which for example are store instructions, inserted compare instructions enable to

detect a divergence between both instruction streams. The overhead of the additionally inserted redundant instructions and the compare instructions is reduced by the instruction level parallelism of super-scalar architectures, which is not fully utilized by the original program.

In Figure 3.2, two redundant threads are shown, which work on a divided register set. An input replication mechanism provides the data that is read to the redundant instruction streams. The output comparison is implemented with compare instructions, which are inserted before stores to check for errors before writing into the memory. The sphere of replication covers the instruction stream and a half of the register set, but the main memory is outside of the sphere of replication and thus requires checking the data to write for errors.

3.2.2 Software Implemented Fault Tolerance

Based on EDDI, the SWIFT approach, *Software Implemented Fault Tolerance* [Rei+05], additionally incorporates enhanced control flow checking. SWIFT assumes a single event upset error model, and ECC protected main memory. Like EDDI, transient errors are detected, but an external mechanism for checkpointing and recovery is required. SWIFT targets the Intel Itanium platform that has 128 general purpose registers available in the instruction-set architecture, compared to the 16 registers of the 64 bit x86 architecture. As a consequence, the division of the register set for the redundant instruction streams limits the usability of the approach on the common x86 architecture, due to the halved number of effectively available registers.

3.2.3 Software-based Redundant Multi-threading

SRMT, *Software-based Redundant Multi-threading* is a software error detection approach for x86 multi-core processors [Wan+07]. A program is executed redundantly with two threads and repeated comparison in between to detect errors. The compiler-based mechanism replicates the program in a trailing thread, where the computations of both threads are compared. The sphere of replication separates the operations of the program into repeatable operations and non-repeatable operations, which are I/O operations, shared memory accesses, and system calls. Values that enter the sphere of replication are duplicated for both threads, and the redundant values are compared before leaving the sphere of replication. A hardware queue is simulated to ease the transmission of values to the trailing thread when a value is duplicated, and when a value is to be compared in the trailing thread. To ensure a correct fail-stop behavior, stores to the main memory must await an acknowledgment of the trailing thread that signalizes a matching and error-free value. By means of compiler optimizations, the store operations, which stop the leading thread until the comparison has happened, are reordered to minimize the number of required comparisons. Operations and functions may be either

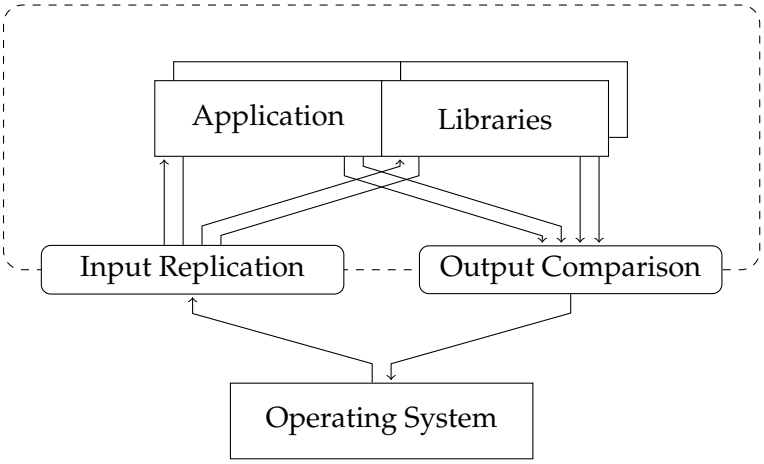


Figure 3.3: Sphere of Replication in PLR [Shy+09]

repeatable, non-repeatable but without fail-stop requirement, and non-repeatable fail-stop operations. The first are free of side effects, and can be executed independently in both redundant threads. The others may change a global state in memory or in a file, and thus require a single execution in the leading thread only. Depending on the fail-stop requirement, non-repeatable operations may need to receive the acknowledgment of the trailing thread beforehand. External functions are handled as non-repeatable operations, and the result is forwarded to the trailing thread, similar to the handling of external functions in this work. Since the communication channel to send values to the trailing thread has been identified as a bottleneck, the authors of SRMT assume a hardware FIFO queue for core-to-core communication, which would be also useful for other applications, as they argument.

3.2.4 Process-Level Redundancy

Instead of redundant thread, PLR, *Process-Level Redundancy*, duplicates the whole process of an application for redundant execution [Shy+09]. This software-focused approach provides a sphere of replication as a software layer between the application and the underlying operating system. A system call emulation unit resembles the boundary of the sphere of replication and implements the error detection and recovery mechanism. This results in a more coarse granularity compared to SRMT, where the identical redundant computations are compared on stores, and not only on system calls. However, error detection is sufficient before data leaves the sphere of replication, and since the full process is inside the sphere of replication in PLR, more frequent comparisons are not required. This renders an additional instrumentation of the application unnecessary, which allows to execute arbitrary programs redundantly with the capability to detect errors, provided by an underlying software library. To enable a redundant execution that

is transparent to the application, additional mechanisms are implemented, for example to forward signals to all redundant processes. System calls are executed only by one process, while the other processes emulate the execution of the system call. Transient errors can be corrected when at least two replicas are created, which allows to kill the erroneous process. For accesses on shared memory, a special mechanism is required to prevent indeterministic behavior of the redundant processes. PLR leverages a virtualization technique to trap on loads and stores in shared memory region, which then are emulated to ensure the identical execution in all redundant processes. However, multi-threaded programs cannot be supported, since the synchronization mechanisms are inside the sphere of replication, which leads to uncontrollable indeterminism in applications with multiple threads.

The principle of PLR is shown in Figure 3.3. Only the application and its used libraries are part of the sphere of replication, the operating system is on the outside. Since system calls are the single point where data crosses the boundary of the sphere of replication, the input replication and output comparison is implemented in a system call emulation layer. The benefit of this approach is the independence from both the underlying hardware and the user-space application and libraries, since no instrumentation is required.

3.3 Transactional Memory for Fault Tolerance

Transactional memory offers the capability to provide checkpoints for fault tolerance approaches, which are required for error correction with dual-modular redundancy and backward-error correction, in contrary to a voting mechanism in a triple-modular redundancy setup. The utilization of transactional memory in dependable embedded systems has been discussed by Fetzer and Felber [Fet+11], concluding that the atomicity of transactions is beneficial for error isolation and mitigation. However, modifications of the transactional memory implementations were assumed to be required to enable a fully functional fault-tolerant embedded system, especially with the additional real-time requirements of such systems in mind.

3.3.1 FaulTM

FaulTM [Yal+10] is based on a customized hardware transactional memory system and redundant threads. Vulnerable code sections are to be defined in the source code, which then are executed redundantly and transactional. The HTM is implemented with lazy conflict detection, i. e. potential conflicts between transactions are explicitly checked in the commit stage, and not eagerly by means of the cache coherence protocol, as it is done in TSX. Due to the lazy conflict detection, both redundant threads can be executed in parallel without provoking a transactional conflict. The versioning is also lazy, implicating a dedicated write-set to temporarily store the memory modifications

that occur during a transaction. The read- and write-set comparison of the conflict detection is used to detect errors by checking the write-sets of the redundant threads for equality. A validation stage that is executed before committing a transaction compares the writ-sets and aborts the transaction if an error is detected. However, a customization of the transactional memory system is required to support error detection, and existing TMs like Intel TSX feature eager conflict detection due to the implementation in the cache coherency.

3.3.2 Log-based Redundant Architecture

The Log-based Redundant Architecture (LBRA) executes redundant threads and compares the computation results by means of verification signatures [Sán+14]. The underlying hardware transactional memory system LogTM-SE is a cache-independent implementation that keeps the read- and write-sets in a log, and thus features an eager version management [Yen+07]. Conflict detection is also eager, i. e. conflicts are detected when they occur, and not in the commit stage. To decouple the conflict detection from the cache as well, signatures of the memory accesses are created, which then are checked to detect conflicts on cache coherence requests. Since LBRA allows decoupling of the redundant threads, input replication and output comparison are required. Memory accesses from the trailing thread are served from the log of the leading thread, and writes are effectively ignored. Special pointers to the logs of the leading and the trailing threads are provided by additional hardware, as well as checkpoint registers. The decoupling of the threads is achieved with a circular log that allows the leading thread to advance multiple transactions without verification. The trailing thread then verifies the transactions by reading from the circular log. The amount of transactions the leading thread can advance is limited by the size of the circular log, which is implemented in hardware.

3.3.3 Hardware-Assisted Fault Tolerance

Hardware-Assisted Fault Tolerance (HAFT) is a similar approach to the one proposed in this thesis, as it also targets transient errors in COTS processors [Kuv+16]. This approach relies on program instrumentation with an LLVM optimization pass, like in this work. Also, Intel TSX is used in the same manner to provide a checkpointing mechanism for rollback on a detected error. However, the authors favor a different kind of redundancy, and they duplicate the instructions of the input program to manufacture an instruction-level redundant execution within a single process. Transaction boundaries are inserted into the program where a transaction is committed and a new transaction is started to fully wrap the program execution in transactions. When the program is executed, consecutive comparisons of the results of the redundant computations enable to detect errors, and let the transaction abort in the event of an error. In contrary to the signature-based approach in this work, errors are detected early, but at the cost of

additional inserted instructions for comparison. Although instruction level parallelism in out-of-order processors reduces the performance impact of the additionally inserted instruction, the rather tight coupling in a single instruction stream prohibits a fully parallel execution of the redundant parts on two processor cores. Memory loads and stores are outside the sphere of replication, since the redundant instructions work on the same memory, in contrary to a separated virtual memory for redundant processes.

3.4 Redundant Execution of Multi-threaded Applications

The redundant execution of parallel applications, particularly with multiple threads and synchronization in between, is difficult, since the interleaving of the threads and the unpredictable order of threads entering critical sections leads to an indeterministic execution of the program. However, to compare redundant threads or processes, they are required to execute the identical instructions and work on the same data. Especially software approaches are prone to this problem, and mechanisms are required to control the indeterministic behavior of parallel threads.

3.4.1 RomainMT

RomainMT enforces determinism for multi-threaded application to enable a redundant execution on the L4 microkernel [Döb+14]. On externalization events, which for example are system calls, the states of the redundant threads are compared to detect errors. This requires the redundant threads to be in an identical state to avoid the false detection of an error. By enforcing the same locking order of mutexes on every execution of a program, the observable behavior will be identical, but only if no race-conditions exist and no lock-free atomic memory accesses occur. RomainMT targets the Pthread library for multi-threading, alike this work, by providing customized implementations of the lock and unlock functions in the uClibc C standard library. The approach to achieve determinism through replication-aware lock and unlock functions, a table in the shared memory is allocated with a specified maximum size, which limits the number of different mutexes that can be used by the application. This lock table data structure is protected by a spin-lock, upon which a thread spins until it is its turn to lock the mutex. RomainMT relies on an external checkpointing and recovery mechanism for DMR, but for triple modular redundancy, forward error correction is possible by selecting the two error-free threads. In this case, the mismatching pair of threads is identified and replaced by a copy of the correct replicas.

3.4.2 FaulTM-multi

FaulTM-multi [Yal+13] is an extension of FaulTM that supports multi-threaded applications. Since it is a hardware-focused implementation, only the underlying synchroniza-

tion instructions like *compare and swap* can be observed. If such an instruction occurs, the currently active transaction is validated, i. e. checked for errors and committed, and the synchronization instruction is executed. Transactions are also validated before thread create and join operations. Diverging states in pairs of redundant threads are not possible, since only the leading thread writes to the shared memory. However, Input incoherence is possible, but only if shared data is accessed without a preceding synchronization instruction, which is a race-condition and considered as a programming mistake. In such case, the validation of the redundant transactions may fail, which falsely indicates an error. The approach also supports optimistic synchronization of transactional applications by adjusting the wrapping transactions to the synchronization transactions. Both redundant processors execute the synchronization transaction, which is nested in the wrapping transaction, and the same read- and write-sets are used. This allows to detect synchronization conflicts between the synchronization transactions, as well as mismatching write-sets of the redundant wrapping transactions. A wrapping transaction is only validated when the inner synchronization transaction ends and is trying to commit. Error detection is performed before conflict detection to avoid the false abortion of a synchronization transaction. In case of a conflict, the wrapping transaction aborts too, and is simply restarted.

3.4.3 Redundant Execution of OpenMP and Message-passing Programs

Further approaches for redundant execution of parallel applications exist, although they target specific programming models, and thus cannot be used with general multi-threaded applications that are based on Pthreads.

A technique for OpenMP has been proposed with Edge-TM [Pap+17], with the objective to provide a rollback mechanism for intermittent errors that result from aggressive dynamic voltage scaling beyond safe level. Decreasing the voltage too far results in erroneous computations, which can be detected through redundant execution and comparison of the transactional write-sets. The implemented transactional memory system is not capable of optimistic synchronization, since the conflict detection is replaced with logic for error detection. Instructions to start and end the resilient transactions are inserted at the boundaries of the OpenMP function calls.

An error detection and recovery mechanism for message-passing applications is described by Villamayor et al. [Mon+17]. Redundant threads are synchronized before sending a message, and the data of the message is compared to detect errors. Only the leading threads sends the message, which is duplicated on the receiving side to be fed into both redundant threads. Thus, the sphere of replication is around the individual nodes, with the output comparison happening before sending a message non-redundant through the network, where the input replication takes place to pass the message to both redundant receiving threads. This approach does not feature a recovery mechanism, as

it is not based on transactional memory. To recover from transient errors, distributed checkpoints are used, which have to be created explicitly by the software.

3.5 Comparison of the Related Work

The approaches described in the previous sections pursue different objectives for a fault-tolerant execution. This results in varying methods for error detection and recovery. Table 3.1 lists the related work described in the previous sections, grouped by the type of their implementation: hardware and software methods, techniques based on transactional memory, and hybrid approaches. The check marks and the crosses in the rows indicate if the corresponding feature is available in the approach of the given column. The individual features are discussed in the following paragraphs.

Table 3.1: Feature matrix to compare the key properties of this approach to related work

	Hardware				Software				TM			Hybrid	
	Lockstep [Ber+05]	TMR [Ber+05]	SRT [Rei+00]	DCC [LaF+07]	SWIFT [Rei+05]	SRMT [Wan+07]	PLR [Shy+09]	RomainMT [Döb+14]	FaulTM [Yal+10]	FaulTM-multi [Yal+13]	LBRA [Sán+14]	HAFT [Kuv+16]	this approach
Space redundancy	✓	✓	×	✓	×	✓	✓	✓	✓	✓	✓	×	✓
Time redundancy	×	×	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓
Signatures for ED	×	×	×	✓	×	×	×	×	✓[1]	✓[1]	✓	×	✓
Loosely coupled	×	×	✓	✓	×	✓	✓	✓	✓	✓	✓	×	✓
Multi-threading	×	×	×	✓	×	×	×	✓	×	✓	×	×	✓
Error recovery	×	✓	✓[2]	✓	✓	×	×	×	✓	✓	✓	✓	✓
COTS hardware	×	×	×	×	×[3]	✓[4]	✓	✓	×	×	×	✓	✓
Memory separation	–	–	–	–	×	×	✓	✓	×	×	×	×	✓
Application specific	×	×	×	×	✓	✓	✓	✓	×	×	×	✓	✓

Space and Time Redundancy Lockstep approaches are space-redundant, since the redundant execution happens in parallel on different processors. Methods with a single thread, where instructions are duplicated within a single instruction stream, do not possess space redundancy. In contrary, they feature time redundancy, since the redundant instruction is executed later in time and not exactly in parallel, as it is the case in lockstepping.

Signature-based Error Detection Without an accumulation of the data that needs to be compared between the redundant parts, the complete data has to be exchanged and compared. An alternative is to generate a signature of the data, which is to be compared, for example with CRC, and to exchange and compare only the signatures. Techniques with cycle-by-cycle comparison like lockstep and TMR do not require signatures, due to the immediate comparison of the signals of the processors, and approaches without redundant threads and processes like SWIFT can directly compare the data in the registers. The FaulTM approaches compare the write-sets of the transactions, which also reduces the amount of data to exchange and compare (see Label 1 in Table 3.1).

Loosely Coupled Redundancy A loose coupling allows the redundant threads or processes to run apart, due to different clock rates of the processors, or because of scheduling in a multi-process system. Lockstep and TMR systems enforce a tight coupling to enable the cycle-by-cycle comparison, while the other approaches feature specific points in the execution where the redundant parts are compared. In contrary to the other software approaches, in SWIFT and HAFT, the instructions of a program are duplicated within the same thread. The resulting execution of the redundant instructions on a super-scalar out-of-order processor will be not as tightly coupled as on a cycle-by-cycle lockstep processor, but also not as loosely as in redundant threads that can be thousands of instructions apart. Methods with loosely coupled redundancy feature a defined sphere of replication with mechanisms for input replication and output comparison at the boundary between the redundant parts and the non-replicated rest of the system.

Multi-threading Support To enable redundant multi-threading, the execution in the redundant threads needs to be identical, which requires determinism in the synchronization between the thread pairs. Only approaches that explicitly provide mechanism to avoid indeterminism can be used for parallel application with multiple threads. This can be implemented in hardware by reacting on the synchronizing atomic instructions, or in software by intercepting the synchronization functions of the thread library.

Error Recovery Mechanism Not all of the described approaches provide a recovery mechanism to restore the execution in case of a detected error. The methods that are based on transactional memory leverage the available checkpointing mechanism for

backward error recovery, while TMR systems use a voter to select the correct data for forward error recovery. The remaining approaches rely on an external mechanism to create checkpoints and to recover, if required. SRTR enhances the SRT method with a rollback mechanism, and thus enables to recovery from detected errors (see Label 2).

COTS Hardware The hardware approaches require extensive modifications to the processor cores and to the pipelines, and thus are not available on COTS hardware. Also, the fault tolerance techniques based on transactional memory rely on special implementations of a HTM with additional enhancements, and thus cannot be used with a COTS processor with HTM. SWIFT targets the Intel Itanium architecture, which is a commercial processor, but became obsolete by now (Label 3). The SRMT approach is usable on a COTS architecture when used without the additional hardware queue, which is used to exchange data between the redundant threads (see Label 4). The remaining methods do not rely on customized hardware.

Memory Separation Redundant processes allow the separation of the processes through virtual memory, which prevents modifications in the memory of the other redundant process due to a corrupted address that resulted from an error. The redundant thread approaches require additional checks on the stores to avoid such memory corruptions. The hardware mechanisms have physically separated memory, thus an additional protection through virtual memory is not required.

Application Specific Fault Tolerance Fully hardware implemented redundancy mechanisms appear as a single processor or core to the operating system and the applications, and thus execute everything redundantly. The software implementations and the hybrid approaches require an instrumentation of the applications that are to be executed fault-tolerant. Although, as a consequence they allow to flexibly combine redundant and fault-tolerant execution of specific applications with the non-redundant execution of other applications that do not demand fault tolerance.

3.6 Summary

Different kinds of mechanisms for redundancy-based fault tolerance have been presented in this chapter, completely hardware-based, and software-only approaches, and hybrid techniques that build upon transactional memory for checkpointing. All approaches have benefits and drawbacks, depending on the intended usage. The redundancy approaches that are fully implemented in hardware, and transparent to the executed system and the application, offer comprehensive error detection capabilities. However, lockstep processors require a full duplication of the whole processor, and a TMR system is required to satisfy the demand of being fail-operational, leading to three times the

cost, plus the cost for the voter. This cost can be seen as the actual price to pay for the additional hardware, the additionally required space, or the multiplication of energy dissipation that the system has to be capable of. Further, the increasing complexity of the processor cores, and variable properties like clock frequencies render a full cycle-by-cycle lockstep execution unfeasible. A trend to cope with the divergences between redundant duplicates is to move the error detection mechanisms up in the stack of abstraction layers. For example, the NonStop Advanced Architecture builds logical cores on a redundant SMT processor, as described in Section 3.1.1. PLR draws the boundary of the sphere of replication on the system call level, which still allows a transparent and uninstrumented redundant execution of an application (see Section 3.2.4). However, to achieve a shorter interval for error detection, an additional instrumentation mechanism is required, like in the other software and hybrid approaches. The appropriate program transformation can be performed during the compilation of the program, or on the binary.

Leveraging already available redundancy in hardware, e. g. by using multiple threads in an SMT processor like SRT, or cores in a multi-core processor, allows to dynamically set up redundancy in a commodity system with no or only little customization. Transactionally memory is used in multiple fault tolerance techniques to provide a checkpointing mechanism to rollback in case of a detected error. HAFT and the approach described in this work leverage the HTM of Intel and thus can be used on existing COTS processors.

Fault tolerance approaches work well for single-threaded programs or multiple unrelated processes. Although, the indeterminism inherent in multi-threaded applications imposes challenges for redundant execution, since diverging states in redundant threads lead to falsely detected errors, rendering the error recovery mechanism unfeasible. Deterministic and thus reproducible behavior has to be enforced where the identical execution of the redundant threads is required, which is the case for synchronization mechanisms. In hardware-based approaches like FaulTM-multi, the leading thread only is responsible for synchronization on atomic instructions, and the trailing thread continues with the state of the leading thread (see Section 3.4.2). Software techniques that build upon the Pthreads-API record the locking order of the leading threads and enforce that order in the trailing threads, as described for RomainMT in Section 3.4.3.

The challenges to provide software mechanisms for error detection on COTS processors have been researched, and techniques have been proposed like SRMT and PLR. However, these require additional software-implemented mechanisms for checkpointing to provide a fault tolerant execution. The transactional memory techniques leverage the existing HTM hardware to enable error recovery, although they require customized hardware. Hybrid approaches promise to provide an integrated solution for a fault-tolerant execution on COTS processors, while leveraging the existing HTM. Yet the redundant execution of multi-threaded applications requires to be incorporated into a software approach.

4

Redundant Execution of Single-Threaded Applications

Contents

This chapter describes the implementation for fault-tolerant execution of single-threaded applications on COTS x86 processors with transactional memory support through Intel TSX. The fault-tolerance layer is a software-based approach which leverages existing hardware for error isolation and for improved performance. For this reason, the existing hardware transactional memory is used for checkpoint creation and to roll back to an

error-free state, which therefore also influences the implementation. Various restrictions and limitations which come along with TSX, the transactional memory implementation of Intel, need to be considered. The general idea of the approach is to duplicate a process and repeatedly compare the memory accesses and register contents of both processes. The instructions of the process are encapsulated in transactions at function boundaries.

To enable a redundant and fault-tolerant execution of existing applications, mechanisms are required that provide duplication, error detection, and recovery. The execution of both redundant processes is compared through CRC checksums of the important data in subsequent basic blocks. A program is instrumented during compilation to enhance the code with additional instructions to create the signatures for error detection. The remaining functionality is implemented in a dynamically linked library, which provides the functionality to set up the redundant execution, for transactional wrapping, and to handle detected errors. Transactional memory allows to rollback the execution in one process in case of a detected error, and provides an error-free checkpoint to setup the redundant execution again after recovering from the error.

The structure of this chapter is as follows: first, the execution model is described, based on loosely-coupled redundant execution. The following sections explain instrumentation, signature exchange, and error detection and recovery. Potential hardware extensions to further increase the performance of the fault-tolerance approach are proposed afterwards. At the end of this chapter, the evaluation of the implemented approach is presented, and the results are discussed.

4.1 A Hybrid Hardware/Software Approach for Fault-Tolerance

A program that is to be executed fault-tolerant is instrumented during the compilation of the program to add the additional code that is required to enable error detection. The remaining functionality to manage the redundant execution, and to rollback in case of an error, is provided by a dynamically linked library.

4.1.1 Target Platform

The target platform of this approach is a server system or a personal computer with a newer Intel CPU, either of the Core or the Xeon family. The processor must not be older than the “Haswell” generation for Core or “Broadwell-EP” for Xeon, since these are the first CPUs with support for TSX [Ham+14]. The instrumentation requires the LLVM infrastructure, and a POSIX-compatible environment is needed for the library. Thus, the preferred operating system is GNU/Linux.

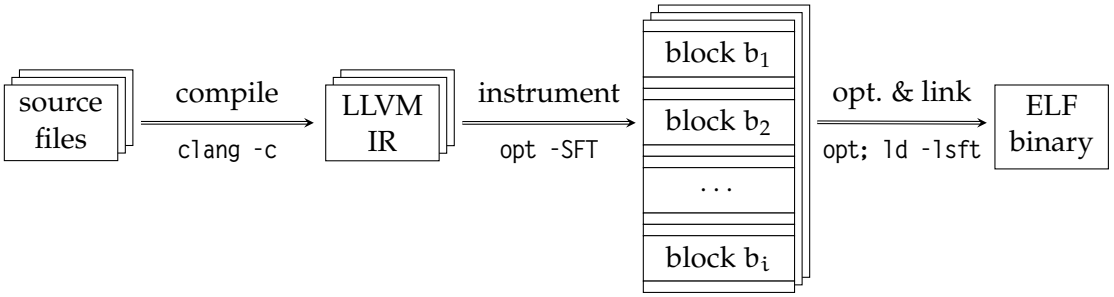


Figure 4.1: The process from the source code of a program to an instrumented binary for fault-tolerant execution.

4.1.2 Transactional Execution for Error Containment

Figure 4.1 depicts the instrumentation of a program on its source code. First, the LLVM frontend `clang` generates the *intermediate representation* (IR) of the input source files. This IR is independent of the target hardware and is used for optimizations in the compilation toolchain. A custom optimization pass, called by the optimizer tool `opt`, modifies the given IR to insert additional instructions to enable the signature-based error detection. Since the further functionality for redundancy management and error recovery is implemented in a dynamic library, calls to the appropriate functions provided by that library are inserted.

The instrumentation on function-level builds *dependable blocks* of multiple subsequent basic blocks, and inserts instructions to generate a CRC checksum of the required values that are created inside a dependable block. The generated CRC checksum is the *signature* of the dependable block, as it is calculated of the concatenation of all data words that leave the dependable block. As described in the introductory section on information redundancy (see Section 2.1.5), CRC-32 guarantees to detect transient errors within a data block of $2^{32} - 1$ bits. Additionally, call instructions to the prefix and suffix functions are inserted for each dependable block, which call the code implemented in the dynamically linked library. The modified IR of the program is then passed through the actual optimization, and afterwards is compiled by the backend into machine-specific code and linked into an ELF binary.

4.1.3 Error Detection and Recovery with Loosely-Coupled Redundancy

The instrumented program binary has been enhanced to generate signatures of dependable blocks, and to call the block wrapping functions. However, the software fault-tolerance library is needed to execute an instrumented program redundantly, and to provide the support to detect transient errors and to recover from them. On startup of the program, the full process of the program is duplicated, which entails the isolation of

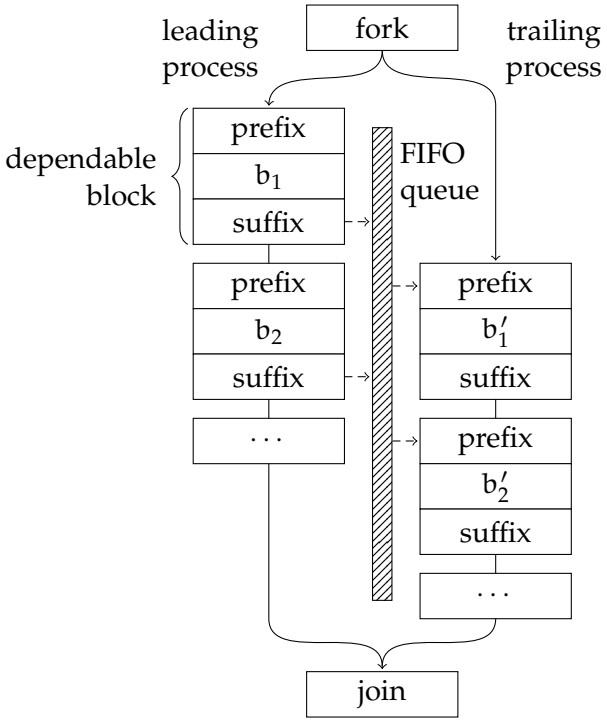


Figure 4.2: Overview of the loosely-coupled redundancy approach

both redundant processes through the memory protection because each process has its own virtual memory address space.

The prefix and suffix functions, which are called at the beginning and the end of each dependable block, implement the exchange of the signatures between the processes, and the error detection and recovery. To leverage the implicit checkpointing mechanism of TSX, Intel's hardware transactional memory implementation, the execution model has to adapt to its requirements and restrictions. Since data cannot be exchanged between transactions, the signatures have to be exchanged before the transaction is started. This results in a subsequent execution of the dependable block in one process and its redundant counterpart in the other process, with the signature exchange between. The process that executes its dependable blocks first and sends the signatures to the other process is referred to as the *leading process*, and the signature consuming process is the *trailing process*. The trailing process wraps its dependable blocks in transactions, and compares the signatures before committing a transaction. This allows to recover from errors in the trailing process, but more effort is required to recover errors in the leading process.

Figure 4.2 depicts the redundant execution with a leading and a trailing process. After the initial fork, a dependable block is executed in the leading process, and its signature is put into the FIFO queue, where it can be read by the trailing process. The FIFO

queue allows the leading process to precede the trailing process until the queue is full to increase the utilization of both cores that execute the redundant processes.

The repeating comparison of block signatures, which are exchanged through a FIFO queue, results in a loosely-coupled redundant execution with recoverability from transient errors on a COTS processor. The combination of instrumentation and redundancy, combined with the checkpoints that are generated by means of the transactional memory, leads to a hybrid hardware/software approach.

4.1.4 Error Model

The described fault-tolerance approach targets transient faults from environmental radiation, which lead to bit flips in the CPU, and thus to erroneous data, for example in a register. The main memory however is assumed to be ECC-protected, and thus is resistant against such faults. Since the redundant processes are executed on different physical cores, and not in an SMT fashion on a single physical core, a transient error in one core cannot occur identically in the other core. Additionaly, the execution is also temporal redundant, since the code in the trailing process is executed later than the same code in the leading process. Multiple transient errors may occur within a single dependable block, but the guaranteed detection depends on the CRC checksum, the size of the checksum data, and if the erroneous bits are consecutive. Since the latter is unlikely for multiple transient errors, single transient errors are guaranteed to be detected, and multiple errors are at least very likely to be detected. This includes also *latent errors*, which have occurred in the leading process at an earlier point in time, but are detected only when the signatures are compared in the trailing process. Even when only single transient errors are assumed, a latent error may lead to multiple errors that are detected simultaneously, if another error occurs additionally. Permanent errors are not the target of this fault-tolerance approach, but they will be detected through mismatching signatures. However, a mitigation method is not implemented.

4.2 Instrumentation

The LLVM compiler framework allows transformations of a program at various points, among which is the optimization on the intermediate representation (IR). To enhance a program with the needed instructions for signature generation, and to shape the dependable blocks with the inserted calls to the prefix and suffix functions, a *function pass* is implemented that processes each function in the IR of a source file.

4.2.1 Functions and Instructions

Programs written in higher languages are structured into functions to group the instructions for a specific task, which provide an interface to increase the composability.

Definition 4.1 *A function* f *consists of a sequence of basic blocks* β_i*:*

$$f = (\beta_i)$$

Functions are subdivided into basic block, which are a sequence of instructions. Basic blocks begin and end at potential changes of the control flow, for example on jumps and conditional branches. As a consequence, the execution order of the instructions in a basic block is sequential.

Definition 4.2 *Each basic block* β *contains a sequence of instructions* λ_i*:*

$$\beta = (\lambda_i)$$

4.2.2 Dependable Blocks

A dependable block is denoted as b, and like a function, it consists of multiple instructions of one ore more basic blocks.

Definition 4.3 *A dependable block* b *consists of a sequence of instructions of one or more subsequent basic blocks. Call and return instructions are not allowed in a dependable block. A dependable block is always surrounded by a call to the block prefix function and a call to the block suffix function.*

$$b = (\mathit{prefix}, \lambda_i, \ldots, \lambda_j, \mathit{suffix})$$

Dependable blocks are generated during the program instrumentation. Figure 4.3 shows the transformation of two basic blocks into dependable blocks. The first basic block β_1 contains four instructions $\lambda_1, \ldots, \lambda_4$, of which the third instruction is a function call. Since a dependable block cannot contain a function call, the first dependable block b_1 is closed before the call, and the next dependable block begins with the remaining instruction λ_4. All following instructions $\lambda_5, \ldots$ are appended to the second dependable block b_2, until the next call or return instruction. The block prefix and suffix functions wrap the dependable blocks into transactions in the trailing process. The exchange of signatures and the error detection is also implemented in these two functions.

Function calls are not allowed to be part of a dependable block to avoid nested transactions during the execution. Nested transactions are possible with TSX, but are not suitable for the checkpointing mechanism, since only the state before the beginning of the outermost transaction is stored, and nested transactions roll back all surrounding transactions as well. Further, the capacity of a transaction is limited, which can be considered during instrumentation by automatically splitting dependable blocks if they become too large. If nesting is allowed, the amount of data that is read and written within multiple dependable blocks inside of a single transaction can no longer be estimated.

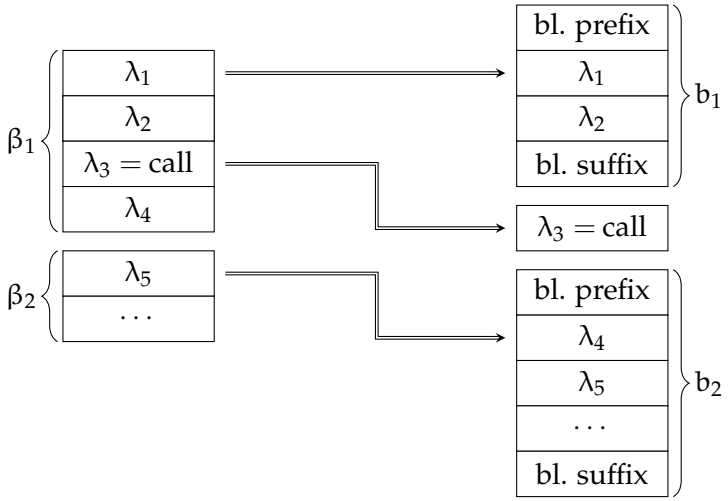


Figure 4.3: The instrumentation of basic blocks results in dependable blocks, which span over multiple basic blocks, but are split on calls.

4.2.3 Critical Values

Resulting from the SSA characteristic of the LLVM IR, an instruction takes one or more values as argument, and the result is also a value, which however cannot be overwritten, since a value can only be assigned once. A value can be used somewhere else in the scope of the function, and multiple uses of a value are possible. To enable the fault-tolerant execution through instrumentation, values are distinguished between *critical values* and normal values. A critical value is a value that is passed to a store instruction, or if it is used outside of the dependable block in which it has been assigned. All remaining values are considered as normal, and they do not require further consideration.

The primal objective is to detect errors in a dependable block before committing its surrounding transaction. It is not possible to recover errors that originated in an earlier dependable block, because the correct data of the values cannot be recovered. By including the data of all critical values in the signature, the error detection mechanism can ensure that the data of the program is error-free at the beginning of each dependable block. The goal is to calculate only a minimal signature of only the required values, which are the critical values, and not of all values that are created in a dependable block. Errors that are detected at the end of a dependable block thus originate from inside this block and can be recovered by re-executing the dependable block.

In Listing 4.1, instructions are shown that use the values of previous instructions, for example in Line 3, the instruction λ_3 is an addition that uses the values `%1` and `%3`, which are the results of the load λ_1 in Line 1 and the previous addition λ_2 in Line 2. The instructions span over two dependable blocks b_1 and b_2, since the call in between leads to a split. The uses of the values in the instructions of Listing 4.1 are depicted in

Listing 4.1 Stored values and values used in another dependable block are critical

```
1 %1 = load i32, i32* %0, align 4                    ▷ critical value
2 %3 = add nsw i32 %1, %2
3 %4 = add nsw i32 %1, %3                            ▷ critical value
4 store i32 %3, i32* %0, align 4                     ▷ critical stored value
5 ; ---                                              ▷ end of dependable block
6 %5 = call i32 @fn()
7 ; ---                                              ▷ begin of dependable block
8 %6 = add nsw i32 %1, %2
9 %7 = add nsw i32 %4, %6
```

Figure 4.4, with the load and store addresses omitted for brevity. The call in λ_5 splits the instructions in two dependable blocks: b_1 on the left, and b_2 on the right. All values that are created in one block and used in another are critical, plus the values that are written to memory by a store instruction. The uses of values are represented in the figure by arrows, and the arrows that leave a dependable block represent a critical value at the origin of the arrow. In the example, `%1` from λ_1, `%4` from λ_3 and `%2`, which is a value created before λ_1, are the critical values that are used by λ_6 and λ_7, which both are part of the subsequent dependable block. Additionally, `%3` is critical, since it is written to memory by the store in λ_4.

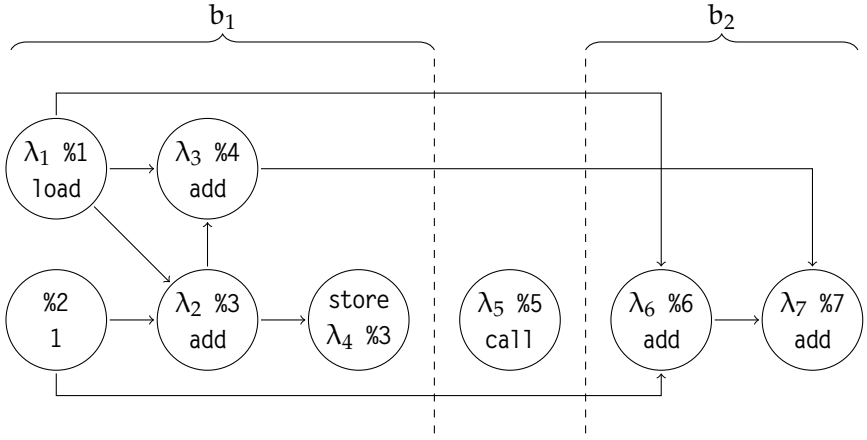


Figure 4.4: Critical values have uses across the boundaries of dependable blocks, identifiable by the horizontal arrows that cross the vertical lines of the block boundaries.

Listing 4.2 Calculating a CRC32 checksum of the data in two registers

```
1 movq    $0x0, %rax
2 crc32q  %rbx, %rax
3 crc32q  %rcx, %rax
```

4.2.4 CRC for Signatures of Dependable Blocks

The SSE 4.2 extension, which is available in the instruction set architecture of the Intel Core processors since the "Nehalem" architecture, provides the instruction `CRC32` to accelerate CRC checksum calculation [Int13b, pp. 4–5]. This instruction accumulates a 32 bit CRC32 checksum of the 64 bit source register in the specified destination register.

As described in Section 2.1.5, CRC allows to detect multiple bit errors, and thus is a feasible method to generate a checksum of a block of data to detect transient errors. The critical values that are required to be included in the signature are subsequently accumulated into a 32 bit CRC checksum. Listing 4.2 shows an example with the initialization of the checksum accumulation register `%rax`, and two subsequent CRC operations. First, the checksum for the 64 bits of the content of register `%rbx` is calculated, and afterwards the content of register `%rcx` is accumulated in the next round of checksum calculation. At the end, the register `%rax` contains a CRC32 checksum of 128 bits of data, stored in the two registers `%rbx` and `%rcx`.

The maximum size of data of which a checksum can be generated with the guarantee to detect errors depends on the checksum polynomial and the block length of the input data. Intel uses the Castagnoli polynomial `0x11EDC6F41` [Int13c, pp. 191–193], which offers a minimum hamming distance of 4 for up to $2^{31} - 1$ bits of input data [Cas+93]. Thus, the CRC polynomial used by the hardware-supported CRC calculation allows large dependable blocks with practically no limitation on the number of data values that are to be included in the signature, since the size of the dependable blocks is limited by the size of the instrumented function and is further divided by function calls.

Definition 4.4 *The calculated signature of the critical values of a dependable block* b *is defined as a function:*

$$s : b \rightarrow \textit{CRC of all critical values } v \in b$$

4.2.5 Function Instrumentation

The instrumentation to enhance a program with instructions to enable a fault-tolerant execution is implemented as a *function pass* for the LLVM toolchain, which is applied on all functions of the input IR during optimization. All functions are instrumented, except

the functions used by the C/C++ library for program startup, for example for memory initialization.

Algorithms 4.1 to 4.5 describe the function instrumentation as it is implemented in the LLVM optimization pass, except some extra functionality for corner cases, which is omitted for brevity. The instrumentation is split into three parts: (1) a signature is created over the data of a dependable block, (2) a dependable block is split at function calls, and (3) calls to the block prefix and suffix functions are inserted at the begin and the end of the function. Since the instrumentation modifies the basic blocks, and because the instructions that are to be inserted depend on the values of the signature instructions, the actual order of these parts is different, as shown in Algorithm 4.1.

First, the function is checked against the blacklist to exclude for example the memory initialization functions of the standard library (see Line 2). If a function is on the blacklist, the instrumentation terminates without any modification. Otherwise, the call to the dependable block prefix function is inserted at the beginning of the function (see Line 6). The beginning of the function is found through the first basic block β_0 and its first instruction, $\beta_0(0)$, in lines 5 and 6. Then all basic blocks are iterated over, and the last values for the local and the remote signature in the preceding basic blocks are searched, except for the first basic block, which takes the initial signature values (in lines 8 – 13 of Algorithm 4.2). Multiple preceding basic blocks are possible due to merging program flow, for example after a conditional basic block. The resulting conflict is resolved at the end of the instrumentation in Line 43 of Algorithm 4.1.

Signature Generation

To add the signature generating instructions, all instructions of each basic block are iterated over, and the critical values are inserted into the signature (see Algorithm 4.3). Critical values are all values that leave the dependable block because they are used somewhere else. Due to the static single assignment (SSA) property of the LLVM IR,

Algorithm 4.1 Insert block prefix at the beginning of a function

```
 1 function RUN_ON_FUNCTION(f)
 2     if f ∈ blacklist then
 3         return
 4     end if
 5     β0 ← f(0)                                          ▷ first basic block
 6     sr ← INSERT_BLOCK_PREFIX(β0(0), before)            ▷ insert block prefix
 7     sl ← ∅                                             ▷ local signature empty
       ...
43     INSERT_PHIS(f)        ▷ merge signature values of multiple preceding blocks
44 end function
```

Algorithm 4.2 Iterate over all basic blocks and find signatures of predecessors

```
    ...
 8  for all β ∈ f do                                  ▷ iterate over all basic blocks
 9      if s_l = ∅ then                                ▷ initialize signature value
10          s_l ← 0
11      else
12          (s_l, s_r) ← GET_SIGNATURES_FROM_PREDECESSORS(β)
13      end if
        ...
```

Algorithm 4.3 Iterate over all instructions of a basic block and insert signatures

```
        ...
14      for all λ ∈ β do                  ▷ iterate over all instructions in basic block
15          ν ← ∅
16          if number of uses of λ > 0 then
17              if λ used in other bb of f ∨ λ used in call then
18                  ν ← λ
19              end if
20          else if λ is store then
21              ν ← get operand of λ                ▷ use operand of store instruction
22          end if
23          if ν ≠ ∅ then
24              INSERT_SIGNATURE_INSTRUCTIONS(λ, ν)
25          end if
26      end for
27  end for
    ...
```

values are not overwritten, and are newly assigned on every modification. Values that are used only within the block can be omitted from the signature, since the instrumentation is transitive, and only the last value in the chain of consecutive instructions is needed to be included in the signature. If an error occurs in an earlier instruction, the error either gets masked, or it proceeds to the next instruction.

A signature is a hashed representation of the contents of all critical values that are assigned inside of a dependable block. To generate that signature, 32 bit CRC is used, since the Intel x86 instruction set architecture provides a CRC checksum generation instruction, which speeds up the signature calculation at runtime.

Arguments for function calls are used in the same block, but require to be included in the signature, too (Line 17 in Algorithm 4.3). This is important, because the dependable

block will be closed before the call instruction, and thus the arguments need to be included in the signature.

Special consideration is required for store instructions as in this case the operand has to be included in the signature, and not the value (see lines 20 and 21 in Algorithm 4.3). Stores write to memory, which is outside of the dependable block, and thus needs to be included in the signature. Writing an erroneous value to memory can be detected through the mismatching signatures, and recovery will be possible because of the transactional execution.

If a value is found that is needed to be included in the signature, the appropriate instructions are inserted (see lines 23 – 25 in Algorithm 4.3). A value cannot be directly inserted into the signature, if its type does not match the 64 bit integer of the signature. Additional conversion instructions are inserted depending on the actual type of the value, as described by Algorithm 4.6. All values in the SSA of the LLVM IR are strictly typed and thus require conversion if the type of the value does not match the type of the signature, for example pointers and floating point values. If the value is of the correct 64 bit integer type, it is assigned to the temporal variable *w* (Line 6). Integers of shorter length are extended to 64 bit, and the value of that newly created instruction is assigned to *w* (Line 8). Pointers and floating point values are also converted accordingly in lines 11 – 17. If the value has the correct type or a conversion instruction has been inserted, the CRC instruction is appended (lines 18 – 20).

Wrapping Dependable Blocks

To match the start of a dependable block at the beginning of a function, a dependable block is closed at the end of a function. Since a function may contain multiple return instructions, the call to the block suffix function is inserted before each of the return instructions, as shown in lines 38 – 40 of Algorithm 4.5.

Algorithm 4.4 Instrument function calls

```
    ...
28  for all β ∈ f do                                  ▷ iterate over all basic blocks
29      for all λ ∈ β do                  ▷ iterate over all instructions in basic block
30          if λ is call then
31              INSERT_BLOCK_SUFFIX(λ)
32              INSERT_BLOCK_PREFIX(λ, after)
33          end if
34      end for
35  end for
    ...
```

Algorithm 4.5 Insert calls to suffix function on return instructions

```
    ...
36  for all β ∈ f do                                  ▷ iterate over all basic blocks
37      for all λ ∈ β do                  ▷ iterate over all instructions in basic block
38          if λ is return then
39              INSERT_BLOCK_SUFFIX(λ)
40          end if
41      end for
42  end for
    ...
```

To avoid the nesting of dependable blocks when the program is executed, dependable blocks are split at function calls (see lines 30 – 33 in Algorithm 4.5). The call to the block suffix is inserted before each call instruction, and the call to the block prefix is inserted after that call instruction.

The insertion of the block prefix call instruction is handled by the function `insert_block_prefix` (see Algorithm 4.7), which takes an instruction λ and the relative position p as arguments. The call then is inserted before or after the specified instruction, according to the relative position.

If the prefix is inserted after a call instruction to split the dependable block, further modifications are required. The remaining code of the function is already instrumented with signature generating instructions. However, these accumulate the signature of the whole function successively, but after the split, the signature has to be resetted to not include values of a different dependable block. This is achieved by searching the next occurrence of the value that stores the local signature s_l, and the value for the remote signature s_r. The found values are the instructions that create these values as a result, and thus the operands of these instructions have to be replaced. To remove the use of the now invalid signature, the value in the operand of that instruction is replaces with an immediate of zero. In Algorithm 4.7, the signature value replacement is listed in lines 6 – 13.

The call to the block suffix is inserted similarly. Although, it is always inserted before the specified instruction, since this instruction is either a call, or a return instruction of the function. Empty dependable blocks can be generated, for example when multiple function calls occur subsequently, or when a function ends after a call. Such empty dependable blocks are removed to reduce unnecessary overhead. The basic block of the specified instruction is walked through backwards until the call to block prefix function is found. If this call is found, and no other instructions are in this basic block, it is considered as empty. The prefix function call is then removed, and no call to the suffix function is inserted.

Algorithm 4.6 Append the value of an instruction to the signature

```
1  function INSERT_SIGNATURE_INSTRUCTIONS(λ, v)
2      w ← ∅
3      λ_t ← λ
4      if v is integer then
5          if v is 64 bit then
6              w ← v
7          else
8              w ← INSERT_INSTRUCTION_AFTER(λ, ZEXT,v)
9              λ_t ← successor of λ
10         end if
11     else if v is pointer then
12         w ← INSERT_INSTRUCTION_AFTER(λ, PTRTOINT,v)
13         λ_t ← successor of λ
14     else if v is float then
15         w ← INSERT_INSTRUCTION_AFTER(λ, FPTOSINT,v)
16         λ_t ← successor of λ
17     end if
18     if w ≠ ∅ then
19         s_l ← INSERT_INSTRUCTION_AFTER(λ_t, CRC32,s_l, w)
20     end if
21 end function
```

The suffix function for the dependable block takes the local and the remote signature as parameter, which both have to be searched first. The list of instructions is walked through backwards from the given instruction until the signature values are found. By adding both of them to the arguments of the function call, the uses of both values are created and thus the dependency, which is important for optimization passes that may be done after the instrumentation.

Example

The Listing 4.3 shows the instrumentation of the IR code of Listing 4.1. Calls to the prefix and suffix functions are inserted at the beginning and the end (see lines 1 and 17). The two dependable blocks are divided by the call to the function `fn()`, therefore a call to the suffix function is inserted before the call, and a call to the prefix function is inserted after the call in Line 9 and Line 13, respectively. The instrumentation adds a CRC instruction for each critical value (see lines 3, 6, 8, and 16). The first invocation of to the CRC in each dependable block takes the initial value of zero as first parameter (see lines 3 and 16), and the subsequent invocations take the result of the previous CRC calculation as first parameter to accumulate the signature. When executed in the trailing process, the prefix

Algorithm 4.7 Insert the prefix at the beginning of a dependable block

```
1  function INSERT_BLOCK_PREFIX(λ, p)
2      if p = before then
3          r ← INSERT_CALL_BEFORE(λ, sft_prefix)          ▷ create call to prefix
4      else
5          r ← INSERT_CALL_AFTER(λ, sft_prefix)           ▷ create call to prefix
6          v ← find next value s_l                        ▷ find next local signature
7          for all λ_t ∈ basic block of λ do              ▷ iterate over this basic block
8              λ_t ← replace operand v with 0             ▷ replace signature value
9          end for
10         w ← find next value s_r                        ▷ find next remote signature
11         for all λ_t ∈ basic block of λ do              ▷ iterate over this basic block
12             λ_t ← replace operand w with 0             ▷ replace signature value
13         end for
14     end if
15     return r                                           ▷ return inserted call instruction
16 end function
```

Algorithm 4.8 Insert the suffix at the end of a dependable block

```
1  function INSERT_BLOCK_SUFFIX(λ)
2      if basic block of λ is empty then                  ▷ check if this basic block is empty
3          remove block prefix call instruction from basic block
4          return ∅
5      end if
6      v ← find previous value s_l                        ▷ find last use of local signature
7      w ← find previous value s_r                        ▷ find last use of remote signature
8      r ← INSERT_CALL_BEFORE(λ, sft_suffix, v, w)        ▷ create call to suffix
9      return r                                           ▷ return inserted call instruction
10 end function
```

function returns the signature of the leading process, which is stored in the value `%sig_r`, and passed back to the suffix function as second parameter.

4.3 Software Fault Tolerance Library

The Software Fault Tolerance (SFT) Library (libsft) is a support library implemented in C to provide the necessary functionality for the redundant execution of a program, and to enable the exchange of signatures of dependable blocks in between. Further, the transactional wrapping, the comparison of signatures to detect errors, and the recovery mechanism are implemented in this library.

Listing 4.3 Instrumented LLVM IR

```
 1 %sig_r = call i64 @sft_prefix()
 2 %1 = load i32, i32* %0, align 4
 3 %sig_l = call i64 @crc32(i64 0, i64 %1)              ▷ add %1 to signature
 4 %3 = add nsw i32 %1, %2
 5 %4 = add nsw i32 %1, %3
 6 %sig_l1 = call i64 @crc32(i64 %sig_l, i64 %4)        ▷ add %4 to signature
 7 store i32 %3, i32* %0, align 4
 8 %sig_l2 = call i64 @crc32(i64 %sig_l1, i64 %3)       ▷ add %3 to signature
 9 call void @sft_suffix(i64 %sig_l2, i64 %sig_r)
10 ; ---                                               ▷ end of dependable block
11 %5 = call i32 @fn()
12 ; ---                                               ▷ begin of dependable block
13 %sig_r1 = call i64 @sft_prefix()
14 %6 = add nsw i32 %1, %2
15 %7 = add nsw i32 %4, %6
16 %sig_l3 = call i64 @crc32(i64 0, i64 %7)             ▷ add %7 to signature
17 call void @sft_suffix(i64 %sig_l3, i64 %sig_r1)
```

Alternatively, the code for wrapping the dependable blocks with the transactional memory instructions and for the signature comparison could be inserted directly into the given program during the instrumentation. The encapsulation of these mechanisms in the dynamically linked library however entails the benefit of a simplified implementation in C instead of LLVM IR assembler, and a compilation seperate from the application, which allows enhanced compiler optimizations. Further, the stack of the program is implicitly preserved through the function calls.

The library provides two functions, `sft_prefix` and `sft_suffix`, which are called at the beginning and at the end of each dependable block through the call instructions that have been inserted by the instrumentation (see Section 4.2.5). The setup of the redundant execution is initiated through a constructor function, which is called automatically during program startup when the dynamic linker loaded the library.

4.3.1 Uniting Instrumentation and SFT Library

The SFT library is dynamically linked to the instrumented program on startup. This is achieved by inserting the dependency to the SFT library into the program binary during the linking of the compiled object files. To automatically initiate the redundant execution of the program, a constructor and a destructor are provided in the library, which are

Listing 4.4 SFT initialization with a constructor

```
1 __attribute__((constructor)) void sft_init() {
2     set_signal_handler();                    ▷ catch and handle SIGINT, etc.
3     sft_create_shm();              ▷ create shared memory segment for FIFO
4     sor_init();                          ▷ initialize sphere of replication
5     sft_fork();                    ▷ setup and start redundant execution
6     fifo_init(&shm->fifo,                          ▷ initialize FIFO queue
7         &shm->fifo_local);
8     time_start();                                  ▷ start time measurement
9 }
```

Listing 4.5 Ending the redundant execution

```
1 __attribute__((destructor)) void sft_deinit() {
2     time_stop();                                 ▷ stop timer measurement
3     sft_join();                        ▷ wait for trailing process and join
4     time_print();                                ▷ print out execution time
5 }
```

executed by the C library automatically during the program initialization and in the exit handler, respectively.

Listing 4.4 shows the constructor that initializes the redundant execution. First, a signal handler is registered to handle signals, which may arrive at one of the both redundant processes, correctly. In Line 3, a shared memory segment is created, which is later used for the allocation of the FIFO queue. Then, the sphere of replication is initialized to provide an interface for external functions that are not to be executed redundantly. In Line 5, the redundant process is created and initialized, and afterwards, the FIFO queue is created in the shared memory, and it is associated with both processes. Before returning to the original code of the program, the time measurement is started, which is used to benchmark the fault-tolerant and instrumented programs.

Upon exiting the program, the destructor shown in Listing 4.5 is executed, where first the timer is stopped. The redundant process is joined after both processes have finished their execution, and at the end the overall execution time is printed to the standard output.

Algorithm 4.9 Forking the redundant process

```
1  function SFT_FORK( )
2      x ← FORK( )                                  ▷ fork the redundant process
3      if x < 0 then
4          abort
5      else if x = 0 then                           ▷ child process (leading)
6          leading ← true
7          trailing ← false
8          P ← GET_PID( )
9          P′ ← GET_PPID( )
10     else                                         ▷ parent process (trailing)
11         leading ← false
12         trailing ← true
13         P ← x
14         P′ ← GET_PID( )
15     end if
16 end function
```

4.3.2 Redundancy Management

The function `sft_fork`, which is called from the library constructor function, replicates the original process of the application through the `fork` system call, creating a new process, which is an almost identical copy of the process, with the same virtual address space (see Algorithm 4.9). The original process and the newly forked process continue their execution after returning from the `fork` function. They can be distinguished by the return value of that function, which is zero in the new process, and otherwise holds the process ID of the child process, or a negative value in case the fork was not successful. Depending on the return value, the process IDs of the leading and the trailing process get assigned, and a global variable is set to identify the leading and the trailing process (see lines 6 – 7 and 11 – 12 in Algorithm 4.9). After returning from the redundancy initialization function, the program execution continues in both the original process and the newly created process, and both processes behave identical.

The newly created process P is named *leader*, and the original process becomes the *trailer*, P′, marked with an apostrophe. This distinction is required, because one process has to proceed the execution of a transaction to allow signature exchange between the processes. Parallelism is achieved by an interleaved execution of the leading and trailing processes.

Before the program terminates, both redundant processes have to be joined. This is managed by the function shown in Algorithm 4.10, which called from the destructor function upon termination of the process, for example when `exit()` is called. First, the FIFO queue is cleared to let the trailing process consume potentially buffered signatures.

Algorithm 4.10 Joining both redundant processes

```
1  function SFT_JOIN()
2      f ← false
3      FIFO_FLUSH(Q, Q^l)
4      if trailing then
5          while ¬f do
6              (c, s) ← WAIT()
7              if c = P ∧ s = exited then
8                  f ← true
9              else if c = −1 then
10                 abort
11             end if
12         end while
13     else
14         EXIT(0)
15     end if
16 end function
```

Since the original process is the trailing process, it has to wait until the leading process has exited, which is implemented by means of the `wait` system call (see Line 6 in Algorithm 4.10). If the leading process has terminated (Line 14), the trailing process leaves the loop and returns from the join function. Afterwards, the initially forked leading process is terminated again, and only the original process is left. Allocated shared memory segments, which are used for the FIFO queue, are released by the operating system when the process is finally terminated.

4.3.3 Signature Exchange

A communication mechanism is required to exchange the signatures of the dependable blocks from the leading process to the trailing process. Since comparison happens only in the trailing process, a uni-directional queue is sufficient, with a single writer and a single reader. The order of the signatures must be preserved to ensure the comparison of correct pairs of signatures. A FIFO queue implemented as a single-linked list satisfies the requirements and provides increased performance compared to a single, locked memory location.

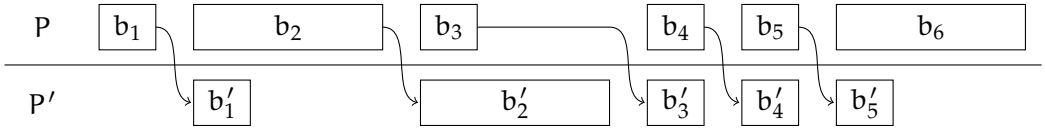


Figure 4.5: Signature exchange without FIFO

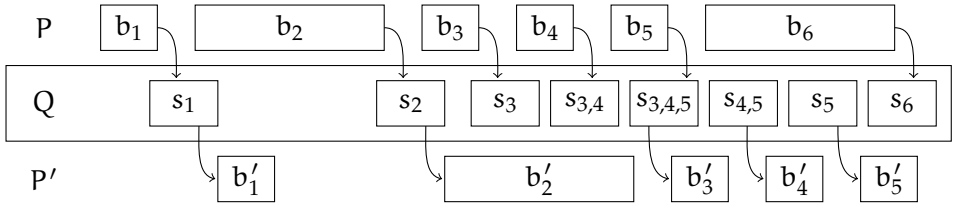


Figure 4.6: Signature exchange through lockfree FIFO queue

FIFO Queue for Signature Exchange

Figure 4.5 shows a simple implementation to exchange a signature between leading process P and trailing process P′ through a single memory location protected with locks for atomic access. A FIFO queue allows to buffer signatures in case the leading process runs ahead of the trailing process. An example of a FIFO implementation Q according to Definition 4.5 is shown in Figure 4.6. The queue allows the leading process, which writes into the queue, to advance further, instead of waiting for the trailing process, which reads from the queue. This decoupling increases the overall performance of the redundant execution. A FIFO queue with only a single reader and a single writer can be implemented lock-free.

Definition 4.5 *A FIFO buffer* Q *is defined as a sequence of signatures of dependable blocks:*

$$Q : (s(b_i), s(b_j), s(b_k), \dots)$$

without a restriction on the order of the dependable blocks. The buffer has a fixed capacity $|Q|$ *and the write position* Q^w *and the read position* Q^r*.*

The FIFO buffer allows to insert elements only at the current write position, and elements can be removed at the current read position. The individual elements of the FIFO buffer are referred to as Q_i, although they can only be accessed if $i = Q^w$ or $i = Q^r$ for read, respectively. Elements that are unused, i. e. elements beyond the write position that are not yet written, or elements that are already removed from the queue, are *empty*. The empty elements ε of a non-full FIFO buffer are subsequent: for $Q = (\varepsilon, ..., s_a, ..., s_b, ..., \varepsilon)$ and $a \leqslant i \leqslant b$, all elements $Q_i \neq \varepsilon$. The signature s_b may occur in Q before s_a if the buffer is *wrapped*. In this case, the queue is non-empty between s_a and the last element of the queue, and between the first element and s_b.

The FIFO buffer Q supports the insertion of data v at the current write position Q^w, if the buffer is not full, represented by $v \twoheadrightarrow Q$. The write position is incremented after the element is inserted to make the newly available data visible. The consuming process removes data from the queue at the current read position Q^r with $Q \twoheadrightarrow v$, where the read position is incremented afterwards to free the space for writing new elements into the queue.

Algorithm 4.11 Writing into a FIFO queue Q with local buffering

```
1  function FIFO_PUT(Q, v)
2      w ← Q^w                                          ▷ store global write position
3      Q_(w+w^l) ← v                                    ▷ store value in queue
4      w^l ← w^l + 1                                    ▷ increment local write position
5      if w^l ⩾ |Q^l| then                              ▷ local buffer is full
6          loop
7              if Q^o = 0 ∧ (|Q| − Q^w + Q^r) > |Q^l| then   ▷ normal state
8                  break                                ▷ elements available
9              else if Q^o = 1 ∧ (Q^r − Q^w) > |Q^l| then    ▷ wrapped state
10                 break                                ▷ elements available
11             end if
12         end loop
13         w ← w + |Q^l|                                ▷ increment write position
14         if w ⩾ |Q| then                              ▷ wraps over end of buffer
15             w ← 0                                    ▷ reset write position
16             Q^o ← 1                                  ▷ set wrap flag
17         end if
18         Q^w ← w                                      ▷ set global write position
19         w^l ← 0                                      ▷ reset local write position
20     end if
21 end function
```

Local Buffering

Since the signature exchange is on the critical path, an optimization of the FIFO buffer leads to a reduced overall performance overhead. The leading process that writes the signatures into the queue and the trailing process that reads from it exchange the data in the shared memory through the cache coherence. Writes and reads of individual signatures lead to subsequent cache line invalidation, and thus a local buffering to gather multiple signatures together increases the queue performance.

The FIFO buffer has local read and write pointers, r^l and w^l, which refer to the memory location inside the buffer Q. The local buffer has no dedicated memory to avoid unnecessary copying of data. Instead, the global read and writer positions are updated when a block of elements is read or written, respectively. The size of the block $|Q^l|$ is a multiple of the cache line size.

The code to write to the FIFO buffer with local buffering is described by Algorithm 4.11. When the put function is called, it is guaranteed that the signature can be inserted into an empty position in the buffer. Concurrent writing into the buffer cannot happen, since only one process writes to the buffer, and the free space is checked after insertion before returning. Thus, the signature can always be inserted at the beginning of the function

Algorithm 4.12 Reading from a FIFO queue Q with local buffering

```
1  function FIFO_GET(Q)
2      r ← Q^r                                          ▷ store global read position
3      if r^l ⩾ |Q^l| then                              ▷ local buffer is empty
4          loop
5              if Q^o = 0 ∧ (Q^w − Q^r) ⩾ |Q^l| then    ▷ normal state
6                  break                                ▷ elements available
7              else if Q^o = 1 ∧ (|Q| − Q^r + Q^w) ⩾ |Q^l| then   ▷ wrapped state
8                  break                                ▷ elements available
9              end if
10         end loop
11     end if
12     v ← Q_(r+r^l)                                    ▷ read element from queue
13     r^l ← r^l + 1                                    ▷ increment local read position
14     if r^l ⩾ |Q^l| then                              ▷ local buffer is empty
15         r ← r + |Q^l|                                ▷ increment read position
16         if r ⩾ |Q| then                              ▷ wraps over end of buffer
17             r ← 0                                    ▷ reset read position
18             Q^o ← 0                                  ▷ clear wrap flag
19         end if
20         Q^r ← r                                      ▷ set global read position
21         r^l ← 0                                      ▷ reset local write position
22     end if
23     return v                                         ▷ return value
24 end function
```

in Line 3. The position in the buffer is the sum of the global write position Q^w and the local offset w^l. If the incremented local offset exceeds the local buffer size $|Q^l|$, the global write position has to be moved forward to make the newly written elements visible to the reading process (see Line 5).

If the local buffer is full, i. e. enough elements are gathered to fill one or more cache lines, the global write position is incremented by the size of the local buffer (Line 13). This is only possible if enough space in the global buffer is free, which is tested repeatedly in a loop (see lines 6 – 12). When the reading process has consumed as much elements as fit in a local buffer, enough space is free in the buffer. The temporal write position w then is incremented by the size of the local buffer (Line 13), and tested if it exceeds the length of the global buffer, which is the case when the buffer wraps around, as described above. In this case, the wrap flag is set, and the temporal write position is set to zero, since the global buffer size is a multiple of the local buffer size. Finally, the global write position is updated, which in turn notifies the reading process if it is currently spinning

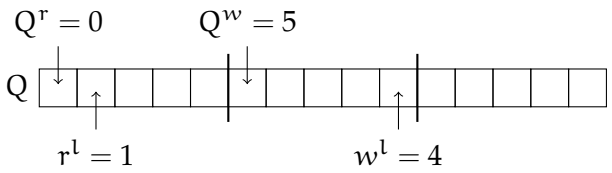


Figure 4.7: An example FIFO queue with $|Q| = 15$ and $|Q^l| = 5$

on an empty buffer. The local write offset is reset, since the global write position now points to the next block in the buffer.

Algorithm 4.12 explains the mechanism to receive an element from the FIFO buffer with local buffering. In contrary to the put function, `fifo_get` has to wait on the first call until elements of the writing process are available in the global buffer. Thus, the local read position r^l is initialized with a number exceeding the local buffer length to enter the waiting loop on the first call. The loop is ended when at least one local block is available in the global buffer, depending on the wrap state (see lines 4 – 10). In Line 12, it is guaranteed that an element is readable at the requested position, and the value is copied into the temporal variable v. The local read position is incremented, and if the local buffer is exceeded, the temporal global read position is enhanced by the length of the local buffer (lines 14 – 22). This marks the case where the local buffer is empty, i. e. the local read offset is equal to the local buffer size. The global read position can be safely incremented by the local buffer size, since the break conditions in the loop (lines 4 – 10) have ensured that the global write position is far enough away.

The lock-free implementation of the write and read operations is possible because only a single writing and a single reading process exists. Given the memory consistency of the designated x86 target, a correct ordering of the read and writes of elements in the buffer, and of the appropriate checks on full or empty buffers, is sufficient. If the reading and the writing processes operate on different local buffers, the cache invalidation due to the modification of the global read and write positions does not mutually impact the processes. Otherwise, if a process has to wait because of a full or empty global buffer, it can poll on the global write and read positions.

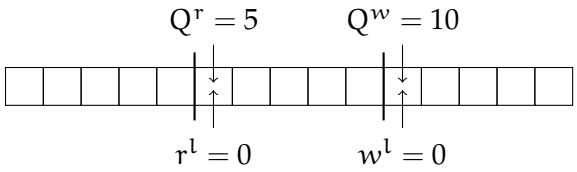


Figure 4.8: The FIFO queue of Figure 4.7 after reading four elements and writing one element

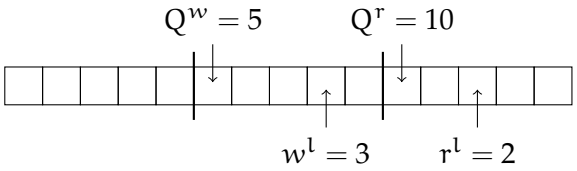


Figure 4.9: Further reads and writes in the FIFO queue of Figure 4.8 leads to a wrap ($Q^o = \mathit{true}$)

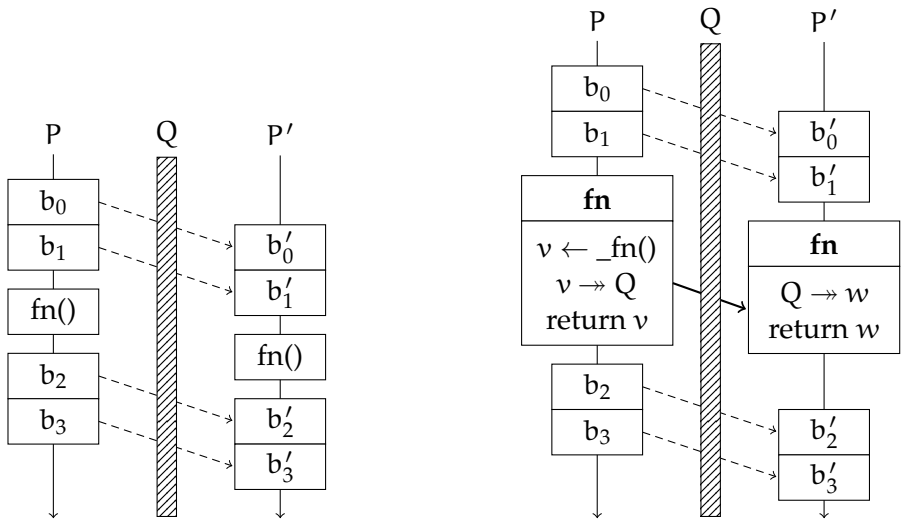
An example of a FIFO buffer with a size of $|Q| = 15$ is depicted in Figure 4.7. The local buffer size in this example is $|Q^l| = 5$, thus the FIFO consists of three local buffer parts, as indicated by the two dividing lines. One local buffer has already been made available in the global FIFO, since $Q^w = 5$, and the local write offset points to the last element in the local buffer. The reading process has consumed one element, thus the global read position is $Q^r = 0$, and the local read offset is $r^l = 1$. There are four more elements available to read from the current local buffer.

If the local buffer is consumed, the reading process has to wait, since then the global write and read position is equal, $Q^r = Q^w$, indicating an empty buffer. The local write buffer is released when its full, which happens when the writing process puts one more element into its queue. Then the global write position is incremented, making five more elements consumable by the reading process. In Figure 4.8, the reading process has consumed its local buffer, the writing process has put another element into is queue, and the global write position was incremented by the length of the local buffer.

The global write position will eventually arrive at the end of the queue, when enough elements are put into the FIFO, resulting in $Q^w + w^l = |Q|$. Depending on the global read position, space at the front of the allocated buffer is free because the elements have already been consumed by the reading process. In this case, the write position and the local write offset are reset to zero to continue writing elements into the queue at the beginning of the allocated memory. However, the global write position is now larger then the global read position, which has to be considered appropriately in the waiting loop in the write and the read function. The additional flag Q^o signifies the wrapped state of the buffer to ease the comparison of the read and write positions. In Figure 4.9, the global write position wrapped, while the global read position, which is logically always behind the write position, has an higher value. The wrap flag Q^o is set to true to correctly calculate the local buffers that are available to be consumed by the reading process. In this example, three more elements can be read from the current local buffer (at position r^l and the two elements to the right), and also from the next local buffer, which starts at the front of the queue. When the reader starts with that local buffer, the global read position is again lower than the write position, and thus the wrap flag can be reset.

4.3.4 Sphere of Replication

All functions which are part of the program are instrumented, as described in Section 4.2. External functions, which are not contained in the binary and hence linked by the dynamic linker on runtime, are not instrumented. These functions are outside of the *sphere of replication* and thus are not covered by the error detection mechanism. In most cases, the desired behavior is that both processes execute these functions on their own and independently. Potential side effects thus affect the leading and the trailing process identically, as shown in Figure 4.10a. However, functions exist that do not return the identical value if called with identical arguments, for example `gettimeofday`, which potentially returns different time values. Also the side effects of library functions may be different in both processes, which could lead to falsely detected errors.



(a) Duplicate and independent execution of the external function in both processes

(b) Interception to execute the function in the leading process only

Figure 4.10: Handling external function calls

External functions must be regarded with the consideration of the sphere of replication, as described in Section 2.1.8. Everything inside the sphere of replication is executed fault-tolerant, thus it requires instrumentation and redundant execution. Code outside of the sphere of replication should be executed only once, and the input parameters have to be checked for errors before passing them to the external function. In turn, the results of the external function have to be distributed to both redundant processes, and also the side effects must be identical. One possible interception of a function call is depicted in Figure 4.10b, where the desired function is executed only in the leading process, and the result is forwarded to the trailing process through the FIFO queue.

Listing 4.6 Type definition for the function pointer and association with the function

```
1 #include <dlfcn.h>
2
3 typedef int (*gettimeofday_fn_t)
4     (struct timeval *restrict tp, void *restrict tzp);
5
6 gettimeofday_fn_t gettimeofday_fn;
7
8 void sor_init() {
9     ...
10    gettimeofday_fn = (gettimeofday_fn_t)dlsym(RTLD_NEXT,
11                          "gettimeofday");
12    ...
13 }
```

The approach to implement the sphere of replication consists of two parts: (1) the replacement of the external functions in the dynamic linking process, and (2) the binding of the original external function and the call to it in the interception.

The dynamic linker can be leveraged to implement step 1 by modifying the search order for dynamic libraries. Unresolved symbols are assigned by the dynamic linker by searching through all dynamic libraries that have been specified when the program was linked. By inserting a custom shared library in front of the specified dynamic libraries in the binary, the dynamic linker can be tricked to link in functions that are implemented in a different shared library file. This mechanism allows to intercept external functions in a way to manage the redundant execution of the desired function.

The second step requires to find the original code of the symbol, which has been replaced in step 1. Each intercepted function has a function pointer in the SFT library with the identical signature of the corresponding function. In Listing 4.6, the function pointer type is defined in lines 3 and 4, and the pointer `gettimeofday_fn_t` is declared in Line 6. The initialization function of the sphere of replication assigns the addresses of the actual code of the intercepted functions to the function pointers, as shown in lines 10 – 11 for the `gettimeofday` function. As described above, the functions to intercept are part of dynamically linked libraries, which are not part of the compiled program. In contrary to these libraries, which are automatically linked to the program at startup, additional shared libraries can be opened manually, which is handled by the `dlopen` API on Linux. With `dlsym`, the address of a symbol in a specified loaded library can be searched. Since there is no difference between the manually loaded libraries and the automatically loaded libraries of the dynamic linker, `dlsym` can be used for all loaded

Listing 4.7 Calling the original function in the leading process

```
1 int gettimeofday(struct timeval *restrict tp, void *restrict tzp) {
2     int t;
3
4     if (is_leading) {
5         t = gettimeofday_fn(tp, tzp);
6         fifo_put(&q, (uint64_t)tp->tv_sec);
7         fifo_put(&q, (uint64_t)tp->tv_usec);
8         fifo_put(&q, (uint64_t)t);
9     } else {
10        tp->tv_sec = (time_t)fifo_get(q);
11        tp->tv_usec = (suseconds_t)fifo_get(q);
12        t = (int)fifo_get(q);
13    }
14    return t;
15 }
```

libraries. By passing the constant `RTLD_NEXT` to `dlsym`, the specified symbol is searched in the next loaded library. Since the SFT library is loaded before any other library, as it has been specified when the program was linked, the appropriate call of `dlsym` returns the address of the actual function.

Depending on the side effects and the dependencies on time or process data, the interception function has to be implemented differently. Listing 4.7 shows the implementation for the `gettimeofday` function, where only the leading process calls the original function (see Line 5). The original function can be called through the function pointer that has been assigned beforehand, as described above. The return value and also the time value has to be passed to the trailing process, which can efficiently be done by putting the data into the FIFO queue (lines 6 – 8). The elements of the FIFO queue have a size of 64 bits, thus the data that is to be sent through the queue has to be split into accordingly sized parts. When the trailing process arrives at that intercepted function, it has already consumed all prior elements in the FIFO queue, and thus can receive the data to resemble the data structure in the trailing process (lines 10 – 12), and to return the identical return value in both processes (Line 14).

4.4 Error Detection and Recovery

Errors in program execution can be detected by repeatedly comparing the execution of both redundant processes, based on the signatures of the executed blocks. A signature is

the CRC checksum of all critical values, which are the values that are written to memory or used in another dependable block (see Section 4.2.3). With this approach, single-bit transient errors are guaranteed to be detected, but a non-equal result of the signature comparison cannot identify the thread in which the error occurred.

4.4.1 Signature Comparison to Detect Errors

The error detection mechanism is shown in detail in Figure 4.11. Both redundant processes execute the same code, which has been enhanced with signature generating CRC instructions during the instrumentation, as described in Section 4.2. These signatures $s(b_i)$ of the values in the executed blocks (b_i) are written into the FIFO buffer Q at the end of each block by the leading process P (Step 1). From there, the trailing process P′ can read the signature at the beginning of a block (Step 2), and before this block b'_i is executed, a transaction is started (Step 3). At the end of block, of which a signature is created as well, the locally calculated signature is compared with the signature of this block in the leading process (Step 4). Only if both signatures are identical, the transaction is committed and the execution continues with receiving the signature of the next block (Step 5). If the signatures do not match, an error must have occurred in this block in one of both processes.

This error detection mechanism is derived from the given hardware restrictions, since the target platform is an existing x86 CPU with Intel TSX as a transactional memory implementation. The straight-forward solution would be to execute both the leading and the trailing process transactionally, and to compare the signatures before deciding if both transactions can commit. However, transactions do not allow to write data that can be read by another transaction without leading to a conflict, since the detection of such conflicts is the primary objective of the transactional memory system. TM implementations that support to pause a transaction or to execute specific memory accesses non-transactional exist, but Intel TSX strongly adheres to the standard TM properties, and it does not offer such extensions. As a consequence, it is not possible to exchange the signature from one process to another while both processes still execute the transaction. The described implementation circumvents this limitation through an interleaved execution, where first a block in the leading process is executed non-transactional, and later that same block is executed transactional in the trailing process. Error detection is still guaranteed, but the complexity of the recovery mechanism is increased, leading to a longer recovery phase. Although, the overall performance of the error-free execution is not impaired, since the interleaved execution with the FIFO queue allows to continue with the next blocks in the leading process independent of the trailing process.

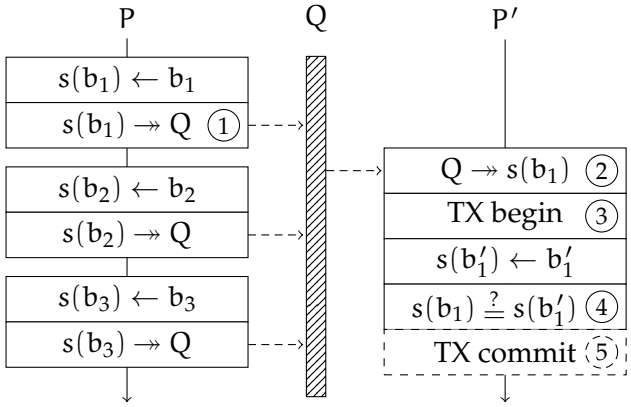


Figure 4.11: Detecting an error in redundant execution

4.4.2 Block Suffix: Sending Signatures and Signature Comparison

All dependable blocks end with the call to the block suffix function, which is shown in Algorithm 4.13. The leading process writes its locally generated signature, which the suffix function gets passed to as parameter s_l, into the FIFO queue (see Line 3 and Step 1 in Figure 4.11). The trailing process executes its dependable block transactional, and before committing the transaction, the signatures of both processes have to be compared in Line 5 (steps 4 and 5 in Figure 4.11). This comparison ensures that no error has occurred in any of the two processes, since a single bit error leads to mismatching CRC checksums. Even more errors in one block can be detected, as described in the background section on information redundancy (Section 2.1.5). The error model described in Section 4.1.4 does not assume the simultaneous occurrence of an error in both processes at the same time, and thus the signature comparison is sufficient to ensure the absence of errors if both signatures are equal. After the successful comparison, the transaction can be committed in Line 6, and the execution in the trailing process continues with the prefix function of the next dependable block.

If the signatures do not match, an error has been detected in one of the two redundant processes. The transaction is aborted explicitly, with the constant `error` as argument, indicating a detected error. Then the transactional memory system aborts the transaction, and the execution continues in the fallback path, which follows the `_xbegin` instruction that is placed in the block prefix function.

4.4.3 Block Prefix: Receive Signatures and Transaction Handling

The block prefix function is called from both processes at the beginning of each dependable block. Algorithm 4.14 describes the implementation of the prefix function. The actual functionality of the prefix is only executed in the trailing process, but the

Algorithm 4.13 The block suffix function writes a signatures and checks for errors

```
1  function SFT_SUFFIX(s_l, s_r)
2      if leading then
3          s_l ↠ Q                                   ▷ write to FIFO
4      else
5          if s_l = s_r then                         ▷ compare signatures
6              _XEND( )                              ▷ commit transaction
7          else
8              _XABORT(ERROR)                        ▷ abort if signature mismatch
9          end if
10     end if
11 end function
```

Algorithm 4.14 The block prefix function reads a signature and starts the transaction

```
1  function SFT_PREFIX( )
2      f ← ∅                                         ▷ flags for retries
3      s_r ← ∅                                       ▷ remote signature
4      c ← 0                                         ▷ counter for transaction aborts
5      BLOCK_BEGIN:
6          if trailing then
7              Q ↠ s_r                               ▷ read from FIFO
8              TX_RETRY:
9                  x ← _XBEGIN( )                    ▷ start transaction
10                 if x ≠ _XBEGIN_STARTED then       ▷ transaction was aborted
11                     if {_XABORT_EXPLICIT, ERROR} ∈ x then   ▷ explicit abort
12                         if f ≠ ERROR_RETRY then   ▷ re-execute transaction once
13                             f ← ERROR_RETRY
14                             goto TX_RETRY
15                         end if
16                         RECOVER( )                ▷ call recovery mechanism
17                         goto BLOCK_BEGIN          ▷ jump back to begin of block
18                     else
19                         c ← c + 1                 ▷ increment abort counter
20                         if SHOULD_RESTART_TX(c, x) then   ▷ decide to restart TX
21                             goto TX_RETRY
22                         end if
23                     end if
24                 end if
25         end if
26         return s_r                                ▷ return remote signature
27 end function
```

distinction between leading and trailing process in Line 6 is required because both processes continue their execution after the error recovery at the label `block_begin`. The trailing process reads a signature from the FIFO queue in Line 7, and starts a transaction (Line 9). If the return value of the transaction begin intrinsic equals the constant `_XBEGIN_STARTED`, the code is executed transactional, and the prefix function returns the received signature from the leading process (see Line 26). This signature is assigned to a value in the instrumentation, and thus is kept in a register or on the stack, depending on the compilation.

However, if the return value of `_xbegin` differs from `_XBEGIN_STARTED`, the transaction has been aborted, and the code after the transaction begin intrinsic is executed as the fallback path of the transaction (lines 10 – 24 in Algorithm 4.14). First, the abort reason is tested to distinguish explicit aborts due to a detected error and random transaction aborts (see Line 11). Randomly aborted transactions are re-executed depending on the abort reason and until a given number of retries is reached, which is decided by the function `should_restart_tx()` in lines 18 – 23. At this point of time, it is yet unknown in which process the error has occurred, and depending on the erroneous process, different actions have to be undertaken for error recovery. If the error occurred in the trailing process, the transaction can simply be restarted to re-execute the dependable block. This restores the registers and the memory to the state of the beginning of the block, which is expected to be error free. On the next try of the transaction, any transient errors have vanished, and the signatures will match, if not other error occurred. The process where the error occurred cannot be determined with only two redundant executions, and thus the transactional execution of the dependable block is simply retried once. If the error has occurred in the trailing process, the next execution will be error-free, and otherwise, the leading process has to be recovered. The flag `f` is tested in the fallback path to retry the transaction only once after an explicit abort (lines 12 – 15).

In case the retried transactional execution in the trailing process leads to another signature mismatch, the error has probably occurred when the dependable block was executed in the leading process. A re-execution of just the dependable block in the leading process is not possible due to the non-transactional execution. As a consequence, the leading process has to be restored completely, which is handled by the `recover()` function in Line 16. When the recovery function returns, a new and error-free leading process is set up, but it also continues its execution at this position. Thus, both processes jump back to the label `block_begin` in Line 5, where the processes are distinguished again.

4.4.4 Recovering the Non-transactional Leading Process

The recovery mechanism for the leading process is called after the transaction in the trailing process has been aborted and re-executed once to eliminate potential errors in the trailing process. If the error persists, i. e. the signatures are still unequal, the error is assumed to have occurred in the leading process, and thus it has to be recovered. Due

Algorithm 4.15 Killing and recreation of the erroneous leading process

```
1 function RECOVER( )
2     KILL(P, SIGKILL)
3     WAITPID(P)                          ▷ wait until leading process is stopped
4     MUNMAP(s)
5     s ← CREATE_SHM( )
6     P* ← FORK( )                        ▷ create new leading process
7     Q ← FIFO_INIT(s)                    ▷ initialize FIFO queue
8 end function
```

to the aforementioned limitation of TSX, a transactional execution of both processes is not possible, and as a consequence, the leading process is at least one dependable block ahead when the recovery mechanism is triggered. Since the leading process cannot roll back, the whole process is killed by means of the operating system. A new redundant process is created with `fork`, resulting in an identical copy of the trailing process. The created child process is the new leading process, which will continue at the beginning of the failed block. Figure 4.12 shows the abort and rollback of the transaction in P', if the signatures do not match due to an error in block b_1, as indicated by the error symbol. The leading process P is killed after a retry of the transaction was unsuccessful, and a new leading process P^* is created. The FIFO queue Q is also recreated to remove signatures of subsequent dependable blocks that the leading process already may have been written before the error was detected.

As described by Algorithm 4.15, the `kill()` system call is used to terminate the leading process immediately in Line 2. After the process has been stopped, which is observed with `waitpid()` on the process ID, the mapping of the shared memory segment is removed in the trailing process (see Line 4). A new shared memory segment is created before the new leading process is forked in Line 6, and afterwards the FIFO queue is initialized in both processes with the new shared memory segment (see Line 7).

The `recover()` function is called by the trailing process only, but both the trailing and the newly forked leading process return from this function. The separation of the control flows of both processes is handled after jumping back to the beginning of the block prefix function, as described in the previous section.

4.5 Hardware Extensions

Based on the software-only implementation that enables a functional fault-tolerant redundant execution on x86 COTS multi-core processes, additional hardware support can potentially increase the overall performance. The compromises that arise from the given hardware, especially the transactional memory system, entail additional overhead, which could be omitted on a customized processor. The main parts that contribute to the

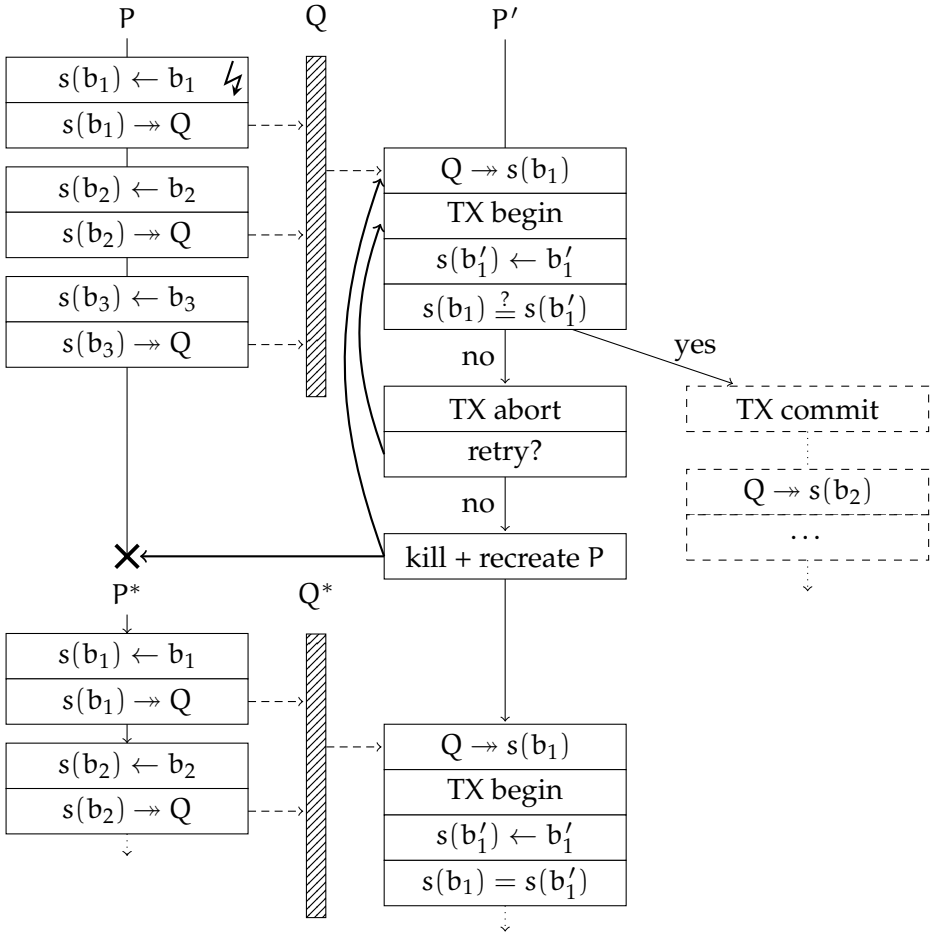


Figure 4.12: Illustration of the recovery process after detecting an error in the leading process

complexity of the approach and the resulting performance overhead are the signature generation mechanism, the FIFO queue for signature exchange, and the transactional memory, which is used for checkpoint creation and rollback. The signature generation further requires the instrumentation of the program to insert the additional instructions, which complicates the compilation and affects the optimization through the increased pressure on the register allocator.

4.5.1 Program Instrumentation for Signature Generation

Each single critical value is included in the signature, which results in a vast amount of CRC instructions that have to be executed. Additional registers are occupied to store the local and the remote signature, which then are not usable by the actual program

code. The instructions to generate the signatures are inserted into the program through instrumentation, which results in a larger amount of compiled program code. Due to the limited registers and the resulting need for additional stack memory, the execution of the program is slowed down further. In total, the signature generation mechanism is assumably the most relevant bottleneck.

A hardware mechanism to implicitly generate signatures would reduce that negative impact. Signatures could be possibly calculated implicitly by issuing a round of CRC calculation on every read from a register and on every write to a register or to memory. The accumulation can happen in a dedicated register, which is accessible from user-space software, for example through a memory-mapped register, or a dedicated port address, depending on the architecture. The comparison at the end of the dependable block then could read from this signature register, removing the occupancy of any general purpose register. Without the need for an instrumentation before the program can be executed, arbitrary and already compiled programs can be executed redundant and fault-tolerant, if the transactional wrapping can be also managed by hardware, or at the boundaries of system calls, for example.

4.5.2 FIFO Queue

Further improvement is possible by supporting the exchange of signatures in hardware. Currently, the signatures of the dependable blocks are implicitly transported through the cache hierarchy, when the data structure is read on the core that executes the trailing process. The local buffering ensures that only complete cache lines are transferred when the cache coherency protocol detects a request on invalidated data. However, the available cache for the redundant processes is reduced, which additionally impacts the performance. Further, exchanging signatures with help of the cache coherence does not scale well if parallel applications are executed redundantly, or when multiple redundant programs run on the same system, due to the pressure on the caches and interconnects in between.

Exchanging signatures in hardware without relying on the cache hierarchy requires a mechanism to send data uni-directionally between individual cores, additionally with buffering to form a FIFO queue. It is sufficient to connect pairs of cores, a queue between each single core is not required. The assignment of processes to cores can be handled by the software library.

4.5.3 Transactional Memory

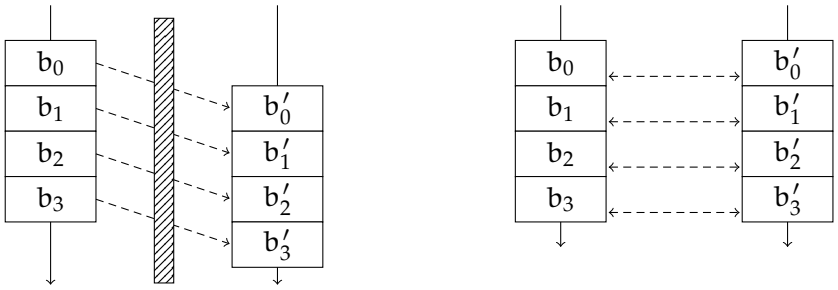
Customizing the transactional memory system can also increase the performance and the versatility of our approach. Transactions should not abort in the error-free case, since conflicts do not occur. However, transactions are limited in their size, which depends on

the cache, and are sensitive to interrupts and other system-related events. Also various instructions are not allowed to be executed inside of a transaction.

Extended transactions can be implemented that relax the guarantees a traditional transaction has to ensure, for example through allowing side-effects that do not originate from within the transaction, but which occurred concurrently. Such robust transactions would survive events like interrupts, and thus enable a guaranteed transactional execution, resulting in a better coverage and less overhead for the fault-tolerant programs. Escape actions allow to read or write data in or out of a transaction without triggering a conflict. This enables parallel transactional execution of both redundant processes. In this case, signatures can be exchanged before the commit phase, and the transaction will only be allowed to commit if the signatures match. Otherwise, both transactions are rolled back and restarted. Consequently, the interleaved execution of leading and trailing process would no longer be required, which thus would simplify the sphere of replication as well. These modifications do not harm the atomicity or the isolation of the transaction itself, but allow external events or explicitly allowed memory accesses to be ignored by the conflict detection mechanism.

4.5.4 Summary

The three proposed extensions to be implemented in a custom processor design could improve the overall performance through a simplified execution model for the redundant and fault-tolerant execution. Figure 4.13 shows a comparison of the original approach adjusted for COTS processors with Intel TSX (Figure 4.13a) with a simplified approach, which requires additional hardware support (Figure 4.13b). Transactions with support for escaping specific instructions allow the parallel execution of dependable blocks, since the signatures can be exchanged while the transaction is still active in both processes. The signatures of the dependable blocks are calculated by the dedicated hardware unit



(a) Original approach for software fault tolerance with TSX

(b) Redundant Execution with additional hardware support

Figure 4.13: Enhancements in a custom processor enable a simplified execution model.

in parallel, and can be exchanged through the hardware link between the cores on which the redundant processes are executed. This allows to compare both redundant dependable blocks before committing both transactions, leading to a simplified rollback mechanism in case of a detected error. In such case, both transactions can be rolled back, which provides an error-free state in both processes without the need to kill and fork one process.

4.6 Evaluation

The presented approach for fault-tolerant execution was evaluated to estimate its effectiveness and the performance overhead. The state-of-the-art benchmark suite SPEC2017 [Pan+18; Sta17] was used to provide data that is comparable to real-life applications.

4.6.1 Evaluation Environment and Methodology

The evaluation was run on a workstation with an Intel Xeon E5-2697 v4 that offers support for Intel TSX. Hyper-threading (SMT on Intel processors) and turbo-boost were disabled to execute each process on an individual physical core, and to ensure reproducible execution times, independent of the potential dynamic up- or downscaling of processor cores. The operating system was based on GNU/Linux 4.15, and the LLVM compiler infrastructure was present in version 7.0.0.

A subset of the SPEC2017 benchmark suite has been selected for the evaluation, with both integer and floating point benchmarks. This includes the benchmarks implemented in C and C++, but no FORTRAN benchmarks, as these require a front-end for LLVM like F18 [NVI19], which is currently under development and not yet released with support for generating an LLVM intermediate representation. The benchmarks have been configured with the *base* configuration that relies on the default optimization flags, while *peak* instead would allow to tune the optimizations per individual benchmark. The SPEC*speed* metric was used because it measures the execution time of the benchmark, while SPEC*rate* measures the throughput. Table 4.2 lists the selected benchmarks and their lines of source code in thousands, according to the SPEC website [Sta17].

Benchmark Configuration

Based on the provided configuration files, a configuration for the baseline was created, which uses the `clang` compiler of the LLVM toolchain instead of `gcc`. In lines 1–3 of Listing 4.8, the modified variables for the C and C++ compiler are shown, which are used for all benchmarks. The following lines 4–10 are set in the configuration file for the instrumentation, where additional flags are passed to the compiler. The baseline configuration has only the optimization parameter `-O3`, the flag to enable the AVX extension

Listing 4.8 Benchmark Configuration for LLVM with SFT

```
1 default:
2     CC  = clang-7
3     CXX = clang++-7
4     CLD = clang-7 -L$LIBSFTDIR/sft -lsft
5
6 default=base:
7     COPTIMIZE   = -mavx -msse4.2 -fno-discard-value-names \
8                       -Xclang -load -Xclang $SFTDIR/SFT.so -O3
9     CXXOPTIMIZE = -mavx -msse4.2 -fno-discard-value-names \
10                      -Xclang -load -Xclang $SFTDIR/SFT.so -O3
```

Listing 4.9 Preparing the Benchmarks for Execution

```
1 runcpu --action=setup --tune=base --rebuild \
2     --config=llvm-linux-x86 $benchmarks
3 runcpu --action=setup --tune=base --rebuild \
4     --config=llvm-linux-x86-sft $benchmarks
```

(`-mavx`), and the flag `-msse4.2`, which is required for the CRC instruction that is part of the Streaming SIMD Extensions (SSE) since version 4.2, for the default optimization flags for all benchmarks. To load the instrumentation pass in the LLVM pipeline, the `-load` flag and the path to the SFT module is passed to the compiler. In the listing, the SFT module is assumed to be located in a path stored in the variable `$SFTDIR`. Both parameters need to be prepended with the `-Xclang` flag to propagate the flag to the optimizer. The parameter `-fno-discard-value-names` is needed by the instrumentation tool to track the values that are used to store the remote and the local signatures, but it has no impact on the compilation into the target machine code, and does not influence the code optimization. The linker command in Line 4, which is set in the configuration for the fault-tolerant execution, contains the path `$LIBSFTDIR` to the fault tolerance library, and the library itself (`-lsft`), to let the benchmark dynamically link this library on startup.

The SPEC2017 benchmark suite provides the management script `runcpu` to build and run the desired benchmarks, which can be used to compile and instrument the benchmarks depending on the configuration for either the baseline or the instrumented version. Listing 4.9 builds all specified benchmarks for the baseline configuration in Line 1, and with instrumentation in Line 3. A list of benchmarks can be provided as the last parameter in the line (`$benchmarks`).

Listing 4.10 Measuring a Benchmark Execution

```
1 LD_LIBRARY_PATH=$LIBSFTDIR/$SFTCFG \
2     perf stat -e {tx-start,tx-abort,tx-capacity,instructions,cycles, \
3     cpu-clock} -x ' ' --post "echo ' *** PERF OUTPUT:'" $binary
```

Performance Measurement

The provided management script can be used to execute the benchmarks, and to measure the execution times, but a custom script is used to provide additional statistics of the benchmark executions. Since the execution of the instrumented benchmark depends not only on the configuration of the instrumentation module, the custom script allows to execute the benchmarks with differently configured versions of the SFT library. The script uses `perf` to measure the CPU time of the benchmarks used on the processor cores, and to gather information about the transactional execution. Listing 4.10 shows the execution of a benchmark program `$binary`, prepended with the `perf stat` command to record the performance counters specified by the parameter `-e`. These contain the transaction statistics counters, the number of instructions and cycles, and the CPU clock time. The environment variable `$LD_LIBRARY_PATH` is set beforehand to specify the path of the fault tolerance library, which allows to select the appropriate configuration of the library for the benchmark execution. The benchmark evaluation script captures the output of the program, and parses the data printed by the `perf` command. The average of the numbers of multiple executions is then written into the result file.

Table 4.1 lists the different configurations that were used to compile, instrument, and execute the benchmarks. The first column specifies the symbol that is used for abbreviation in the following sections, for example t_F represents the execution time of the fully fault-tolerant execution. This configuration loads the instrumentation module during compilation, and the library with all features is loaded during evaluation by means of the `$SFTCFG` variable. In contrary, the original benchmark configuration does not

Table 4.1: Benchmark configurations

	Configuration	Instrumentation	Library
O	original	*none*	*none*
F	fully fault-tolerant with TSX	`-load SFT.so`	FIFO, TSX
–	error detection, no TSX	`-load SFT.so`	FIFO
–	no signatures	`-load SFT.so --no-create-signatures`	FIFO
H	no signatures, no exchange	`-load SFT.so --no-create-signatures`	fork only

Table 4.2: Overview of the selected SPEC2017 benchmarks [Sta17] and the instrumentation statistics.

Benchmark	Type	Lang.	KLOC	#b	λ/b	λ_c/b	λ_s/b
600.perlbench	int	C	362	29,085	24.85	3.99	2.17
605.mcf	int	C	3	208	34.99	6.92	4.58
619.lbm	fp	C	1	89	89.01	6.43	4.71
620.omnetpp	int	C++	134	50,958	6.68	1.88	1.03
623.xalancbmk	int	C++	520	144,348	6.95	2.05	1.09
631.deepsjeng	int	C++	10	1,019	26.61	4.82	2.91
638.imagick	fp	C	259	35,377	13.25	3.24	1.75
641.leela	int	C++	21	7,442	6.61	2.09	1.01
644.nab	fp	C	24	3,089	17.05	3.81	2.07
657.xz	int	C	33	2,065	25.54	5.24	3.39

include the instrumentation, and the fault tolerance library is not linked. To estimate the overhead of TSX, a version is evaluated with full instrumentation, but without transactional wrapping, as described by Row 3 in Table 4.1. The signature generation overhead resulting from the CRC instructions can be measured by comparing the execution time with a configuration that does not insert the signature generating instructions during the instrumentation. Finally, the configuration specified by the symbol H resembles the best-case for the emulation of the proposed hardware enhancements, since it does not contain signature generation, no signature exchange, and no transactional wrapping.

4.6.2 Instrumentation Analysis

The statistics resulting from the instrumentation with the LLVM optimization pass, which is described in Section 4.2, are listed in Table 4.2. Beside the benchmark name, its type, which is integer (int) or floating point (fp), the language, and the number of source code lines in thousands (KLOC) is specified. Further, the number of dependable blocks that have been created during the instrumentation is listed (#b), as well as the ratio of values per dependable block (λ/b), the ratio of critical values per dependable block (λ_c/b), and the average number of stores per dependable block λ_s/b.

The C++ benchmarks `omnetpp`, `xalancbmk`, and `leela` stand out in the table, as their average number of values per dependable block is substantially smaller than for the other benchmarks. The number of values resembles approximately the size of the dependable block in instructions, since most instructions have a result that is assigned to a value. A low average of λ/b entails small dependable blocks, which necessarily results in a higher performance overhead due to the large number of function calls to the block prefix and suffix functions.

Table 4.3: Relative execution times and instructions per cycle

Benchmark	Type	t_F/t_O	t_H/t_O	IPC_F	IPC_H	IPC_O
600.perlbench	int	8.77	2.36	1.03	1.40	2.42
605.mcf	int	4.36	1.57	0.91	0.83	0.79
619.lbm	fp	1.24	1.01	0.97	0.71	0.73
620.omnetpp	int	22.04	6.20	0.81	0.63	1.12
623.xalancbmk	int	39.67	8.75	0.81	0.68	0.98
631.deepsjeng	int	10.25	3.11	0.96	0.93	1.68
638.imagick	fp	4.03	1.07	1.84	2.12	2.18
641.leela	int	35.29	9.46	0.82	0.56	1.25
644.nab	fp	2.86	1.21	1.42	1.75	1.78
657.xz	int	6.54	1.74	0.93	1.01	1.18

4.6.3 Performance Overhead

To analyze the performance overhead, the benchmarks were evaluated with different configurations of the instrumentation and the fault tolerance library. With full support for error detection and recovery, the average execution time is over four times of the original benchmark execution for the C benchmarks, and around 25 times for the C++ benchmarks. In the chart in Figure 4.14, this configuration is represented by the bars labeled t_F/t_O, which represent the execution time of the fault-tolerant configuration relative to the original benchmark execution.

To determine the sources of the performance overhead, individual features were turned off, which allows to determine the impact of the transactional wrapping, the signature generation, the signature exchange, and the instrumentation itself. This also allows to estimate the potential impact of the hardware enhancements that have been proposed in the previous section. The results of the corresponding configuration is labeled as t_H/t_O in Figure 4.14. The average relative execution time of the C benchmarks for this configuration is 190 %, and thus a little less than executing the benchmark twice consecutively.

Table 4.3 lists the individual relative execution times for the benchmarks, as well as the instructions per cycle (IPC) for the original (IPC_O) and the fault-tolerant configuration (IPC_F), and for the configuration with emulated hardware enhancements (IPC_H). The relative execution time for the fault-tolerant execution with TSX wrapping is listed in the column t_F/t_O. As in Figure 4.14, the column t_H/t_O in Table 4.3 lists the relative execution time for the fault-tolerant execution with emulated hardware extensions. The IPC values for the different configurations of the benchmarks are also depicted as a chart in Figure 4.15.

In some cases, the IPC of the configuration with hardware extensions is lower than the IPC of the fault-tolerant execution, for example for `lbm` and `leela`. This indicates

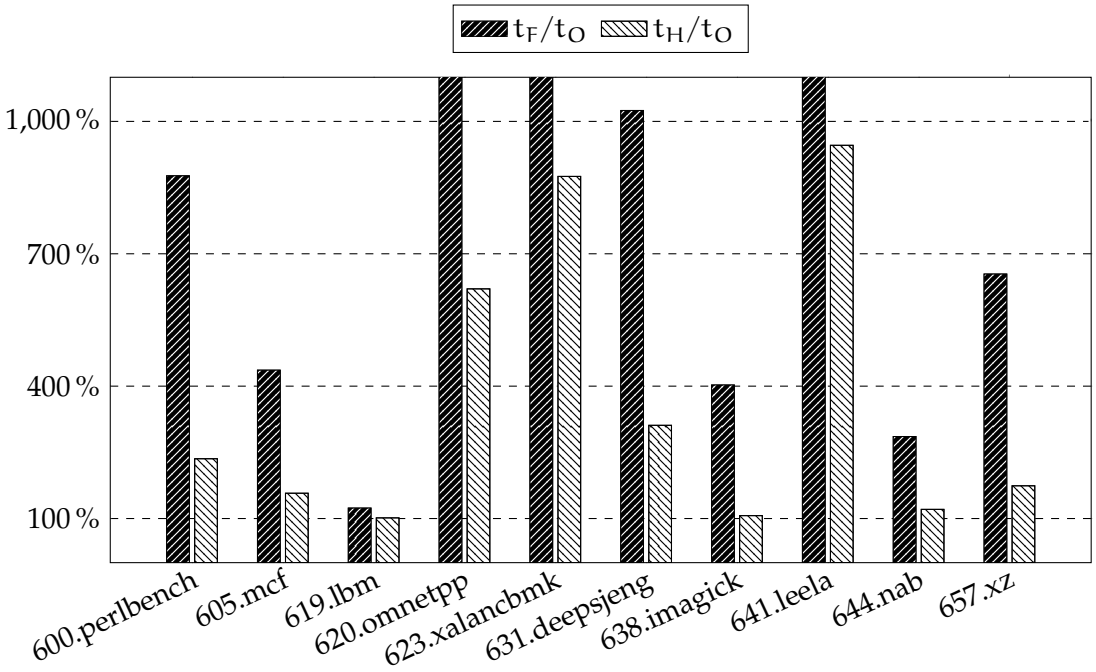


Figure 4.14: Relative execution time

that the processor is not fully utilized in configuration H, and some of the additional instructions for signature generation and exchange fit into free execution units of the pipelines. As a consequence, the IPC increases for the configuration F, but the execution time also increases. The C++ benchmark `leela` has many small dependable blocks and thus executes a lot of calls and branches for the prefix and suffix functions, which is also the case for the configuration with hardware extensions. However, the resulting overhead can be masked to some degree with the signature generation and exchange instructions.

With the execution times of the different configurations per benchmark, the sources of the performance overhead can be estimated. The difference of the execution times of the fully fault-tolerant execution with TSX and the configuration without TSX indicates the overhead of the transactional wrapping. By disabling the signature generation in the program instrumentation, no CRC instructions are inserted, which allows to measure the performance impact of the signature generation. The impact of the signature exchange can be estimated by additionally disabling the FIFO operations. When putting the individual time differences in relation, the overhead distribution can be expressed in percentage of the overall performance overhead. Due to the complex out-of-order cores of the evaluation platform, the IPC is not constant for the different configurations, which results in inaccurate calculations. Actually, different parts of the fully instrumented program are executed overlapping or in parallel, for example CRC calculations and the

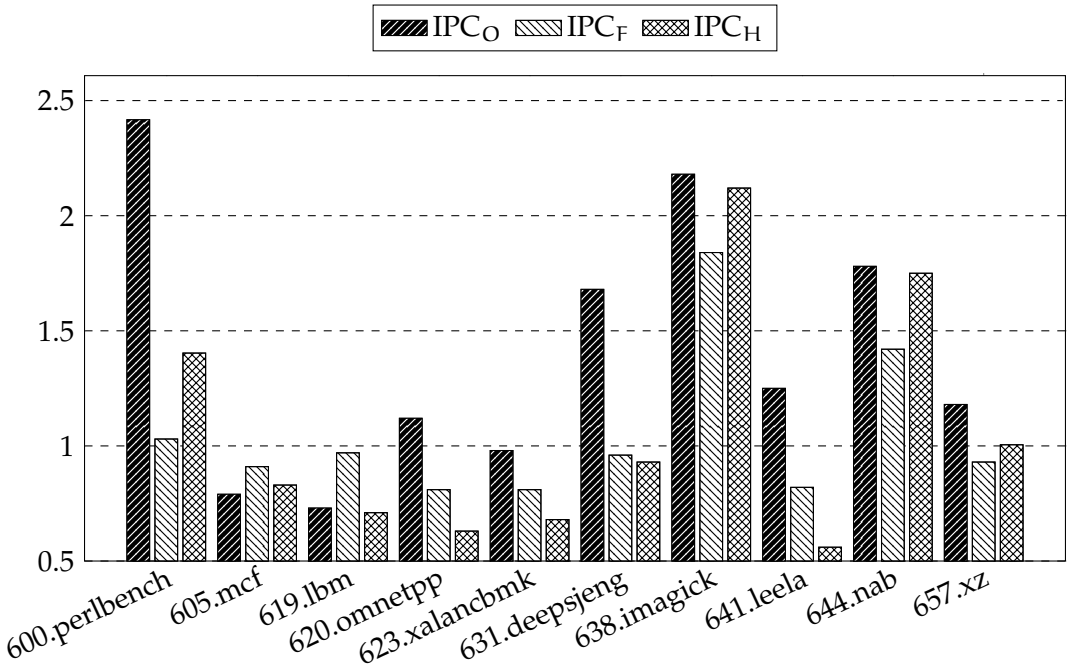


Figure 4.15: IPC comparison

original instructions of the program, or the code of the signature exchange and code of the prefix and suffix functions. Although, the resulting data gives a coarse estimation of the overhead distribution, which is depicted in the graph in Figure 4.16, and listed in Table 4.4. The following paragraphs explain the individual sources of overhead in detail.

Signature Generation

The second bar from top in Figure 4.16 shows the contribution of the signature generation to the overall performance overhead. For most benchmarks, the `CRC` instructions that are used to calculate the signatures come with only a small performance penalty. One reason is the detection of critical values in the instrumentation, and the CRC calculation of only these critical values. Further, the checksum can be calculated overlapping to the other instructions of the original program, since multiple instructions are in between subsequent CRC calculations, and the CPU can execute these calculations and the other instructions in parallel, due to its out-of-order pipeline.

However, the floating point benchmarks `imagick`, `nab`, and `lbm` exhibit a high relative amount of the performance overhead in the signature generation. The CRC checksum calculation instruction works only on general-purpose registers, and thus the floating point values need to be copied into such integer register beforehand. This entails an additional `MOV` instruction, and one more register is required to temporarily store the value that is to be accumulated into the signature. Additionally, floating point computations

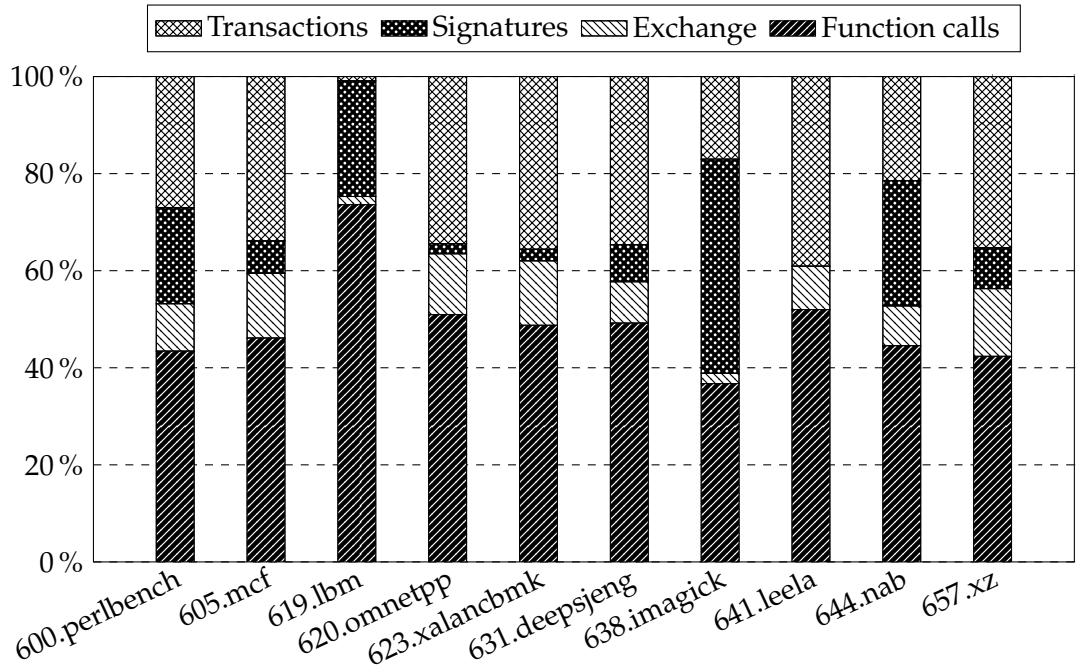


Figure 4.16: Overhead distribution

and CRC calculation can suffer a resource conflict in the pipeline, and thus are required to be executed sequentially.

Interface to the Software Library

The function calls into the dynamically linked library contribute a significant amount to the overall performance overhead of the fault-tolerant execution. As the lowest bar in Figure 4.16 shows, the relative overhead due to the call instructions is over 40 % for most benchmarks. The floating point benchmark `lbm` has rather large dependable blocks, although the function calls represent the majority of the performance overhead. Although, this benchmark executes only a small amount of dependable blocks, compared to the other benchmarks, and thus the overhead of signature exchange, and the influence of the transactional memory is small.

Transactional Memory

The transactional memory is another source of performance overhead, since each dependable block is wrapped with a transaction in the trailing process. TSX does not guarantee a non-conflicting transaction to commit eventually, and thus aborts due to other reasons like cache overflows, exceptions, and interrupts may occur. Such aborts are transparent to the executed program, since the affected transaction is executed again,

Table 4.4: Overhead distribution

Benchmark	Transactions	Signatures	Exchange	Function Calls
600.perlbench	26.99 %	19.83 %	9.78 %	43.40 %
605.mcf	33.88 %	6.66 %	13.36 %	46.10 %
619.lbm	0.92 %	23.80 %	1.72 %	73.57 %
620.omnetpp	34.45 %	2.07 %	12.54 %	50.93 %
623.xalancbmk	35.59 %	2.39 %	13.29 %	48.73 %
631.deepsjeng	34.63 %	7.66 %	8.53 %	49.18 %
638.imagick	17.04 %	44.13 %	2.11 %	36.72 %
641.leela	39.08 %	0.00 %	8.95 %	51.97 %
644.nab	21.44 %	25.84 %	8.22 %	44.51 %
657.xz	35.30 %	8.40 %	13.96 %	42.34 %

and the chances to commit successfully are high for the second try of a transaction. The resulting overhead of aborted and restarted transactions consists of the time needed to rollback the transaction plus the time needed to re-execute the code inside of this transaction.

Within Figure 4.16, the fraction of the performance overhead related to transactional memory instructions is shown in the bars at the top. Table 4.5 lists the relative execution time of the fully fault-tolerant configuration with TSX, the number of executed transactions per benchmark, together with the average percentage of aborted transactions (x_a/x), and of those the average percentage of aborts due to the overflowing cache capacity ($x_{a|c}/x_a$).

The number of transaction aborts is relatively low, except for the floating point benchmark `imagick`, where about 6 % of transactions abort, but only 3 % of those due to an overflowing cache. Floating point operations may lead to aborts during a transaction, for example when the state of the floating point unit is modified.

Signature Exchange

The benefit of the FIFO queue for signature exchange between the redundant processes was evaluated with different FIFO queue sizes and different local buffer sizes. Figure 4.17 and Table 4.6 show the speedup of the program execution of a single benchmark, relative to the benchmark execution without a FIFO queue. The configuration of the FIFO queue without a local buffer is identical to a local buffer of length 1, where each single signature has to be written immediately into the shared memory, and the reading process can only get one signature out of the global buffer simultaneously.

A micro benchmark was used to estimate the influence of the FIFO queue. To measure the maximum effect, the benchmark should be limited by signature exchange, and thus it was implemented with nested loops and function calls, since the instrumentation

Table 4.5: Transactional Execution Statistics

Benchmark	t_F/t_O	Instructions	Transactions x	x_a/x	$x_{a\|c}/x_a$
600.perlbench	8.77	$2.20 \cdot 10^{13}$	$9.76 \cdot 10^{10}$	0.57 %	0.04 %
605.mcf	4.36	$2.11 \cdot 10^{13}$	$1.18 \cdot 10^{11}$	0.43 %	2.43 %
619.lbm	1.24	$1.55 \cdot 10^{13}$	$2.09 \cdot 10^{7}$	0.01 %	48.94 %
620.omnetpp	22.04	$3.34 \cdot 10^{13}$	$2.48 \cdot 10^{11}$	0.00 %	10.96 %
623.xalancbmk	39.67	$6.90 \cdot 10^{13}$	$5.09 \cdot 10^{11}$	0.00 %	0.27 %
631.deepsjeng	10.25	$2.50 \cdot 10^{13}$	$1.45 \cdot 10^{11}$	0.51 %	0.05 %
638.imagick	4.03	$2.62 \cdot 10^{14}$	$1.05 \cdot 10^{11}$	5.63 %	3.17 %
641.leela	35.29	$9.64 \cdot 10^{13}$	$7.11 \cdot 10^{11}$	0.01 %	2.83 %
644.nab	2.86	$6.33 \cdot 10^{13}$	$1.76 \cdot 10^{11}$	0.00 %	42.68 %
657.xz	6.54	$9.14 \cdot 10^{13}$	$4.91 \cdot 10^{11}$	1.03 %	0.48 %

Table 4.6: FIFO queue size vs. program speedup

	local FIFO size					
FIFO size	1	16	64	256	1024	4096
2^{10}	3.4	11.7	21.0	31.2	–	–
2^{13}	3.0	11.9	22.6	34.0	36.5	38.9
2^{16}	3.3	10.3	22.2	34.6	38.4	39.3
2^{19}	3.0	10.4	20.0	34.2	38.1	38.8
2^{22}	2.9	11.5	22.5	34.0	37.4	37.7
2^{25}	3.8	10.9	20.8	31.2	34.3	35.3
2^{28}	7.4	11.7	18.6	21.1	22.8	23.9

splits dependable blocks on call instructions. The resulting dependable blocks are almost empty, but the calls to the prefix and suffix functions are always inserted, and thus the 10,000 iterations of both loops lead to 200,020,001 executed dependable blocks, hence the same number of signature exchange operations were performed. The measurements of the execution times were taken without TSX to avoid a negative impact due to aborting transactions.

The performance of the single-element local buffer is already better than without a FIFO queue at all, the program execution is at least 2.9 times as fast. Without a FIFO queue, the shared memory buffer holds only a single signature, and thus the redundant processes have to wait on each other until a signature is placed in the buffer by the leading process, or read by the trailing process, respectively.

However, increasing the local buffer size leads to a substantial performance improvement, and peaks for this specific benchmark with a size of 4,096 elements. The speedups decline for a FIFO queue sizes larger than 2^{28}, since the benchmark executed only ca.

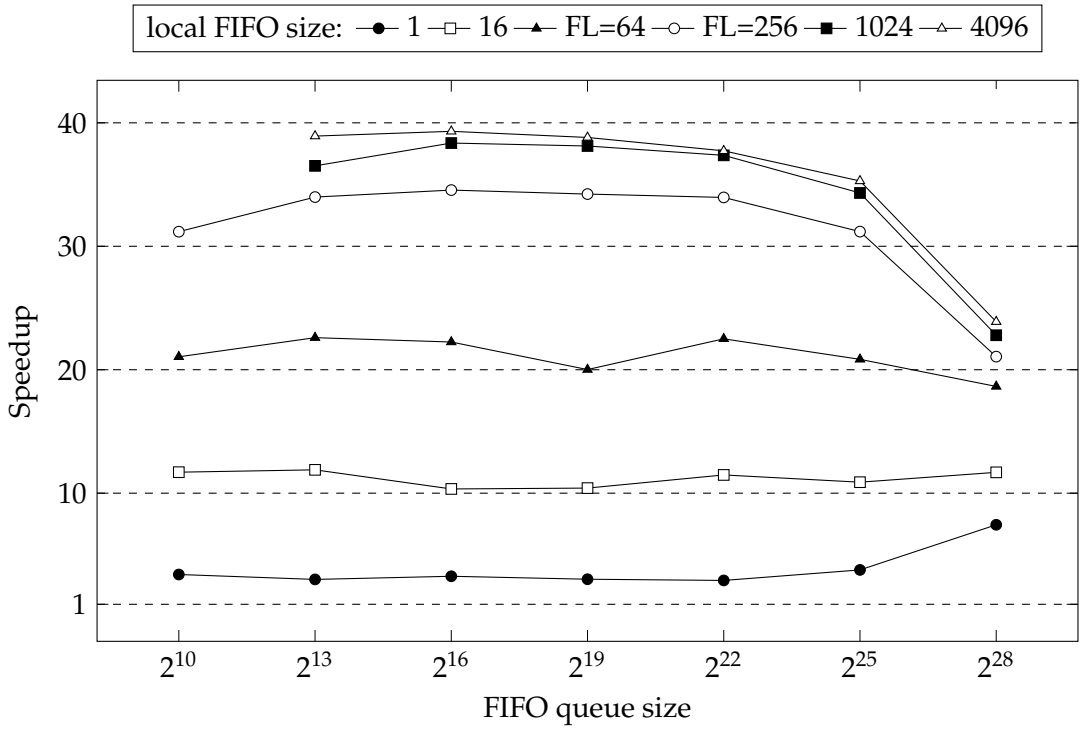


Figure 4.17: FIFO queue size vs. program speedup

$2^{27.5}$ signature exchange operations. No data is available for configurations where the local buffer size is greater or equal than the FIFO queue size, since the FIFO queue size must be a multiple of the local buffer size.

4.7 Limitations

The software-only fault tolerance approach without hardware modifications exhibits some limitations on the general applicability for a redundant execution of arbitrary applications.

First, the source code of the program that is to be executed fault-tolerant must be available, since the instrumentation is part of the compilation process. This further restricts the feasible programs, since a front-end for the LLVM toolchain is required for a programming language of the application. For FORTRAN for example, a front-end that generates a suitable intermediate representation for the instrumentation module is not yet available and still under development. Although, as long as the instrumentation is required to generate appropriate dependable blocks, the instrumentation on the LLVM IR is less complicated than binary instrumentation on an arbitrary executable in x86 machine code, as previous work has demonstrated.

The characteristic of object-oriented C++ programs, which feature virtual functions that result from the inheritance through the class hierarchy, prevents the inlining of small functions. This results in necessarily small dependable blocks, since these have to be split on function calls. In C programs, function inlining is possible more often, which results in larger dependable blocks, and thus in less transactions, fewer signatures that are to be exchanged, and a reduced number of calls to the prefix and suffix functions.

A fully software-implemented redundancy mechanism has to exhibit fragments of code that are not executed redundantly, and thus are susceptible for transient errors. Additionally, undetected errors occurring in such regions of the software fault tolerance library can not be mitigated, since no redundant data is available for recovery. However, every redundancy mechanism has some non-redundant unit that is prone to errors, for example the comparator in a DMR lockstep processor. For software solutions, this vulnerable region of code should be minimal to reduce the chance of errors. Other kinds of redundancy, for example duplicated instructions and redundant data structures, could help to reduce the risk for undetected errors, and are part of potential future work. The redundancy is also suspended during recovery, where the potentially erroneous leading process is killed and then restored from the trailing process. An error occurring in the trailing process during the recovery phase could affect the state of the newly created process, and lead to an undetectable error. After the `fork`, both processes are identical, and the signature comparison after the program execution is resumed cannot detect errors that occurred before the start of this dependable block.

The sphere of replication at the boundary of the redundantly executed program requires a manual implementation of the interception of all external functions that cannot be executed twice in both processes. This is the case when an external function returns different data when called multiple times in an identical process state, for example `gettimeofday`. Other external functions may possess side effects on the system or the process, and thus are not repeatable, for example when writing into a file, which changes the content of the file and also the file handle. The fault tolerance library used in the evaluation contains the interceptions for the non-repeatable functions that are used in the benchmarks, but the sphere of replication cannot support arbitrary external functions without manual adaption.

4.8 Summary

In this chapter a method was described to execute a single-threaded application fault-tolerant on an existing x86 processor. A program is instrumented during compilation, and duplicated on program startup. Repeated comparison of the redundant processes enables error detection, and transactional wrapping provides the checkpointing and rollback capability. The applicability of the approach was evaluated and shown with

benchmarks of the SPEC2017 suite. The results indicate that potential for performance improvement exists, when some of the functionality is provided by the hardware.

The sphere of replication handles external functions to ensure that the states of the redundant processes remain identical. However, to support multi-threaded applications, additional functionality is required. Multi-threaded programs exhibit indeterministic behavior, since the order of threads on synchronization mechanisms, for example when locking a mutex, is not identical across multiple executions. Thus, the locking order of threads in redundant processes can diverge, which can lead to different states in both process, and as a consequence, errors could be detected spuriously. The next chapter describes a mechanism for synchronization in redundant applications, and further proposes a technique for recovery of multiple threads in case an actual error is detected.

5

Fault-tolerant Execution of Multi-threaded Applications

Contents

Based on the approach for the redundant execution of a single-threaded application that was described in the previous chapters, the fault-tolerance library now is extended to support multi-threaded applications, too. This entails the need for additional mechanisms for thread control to repress indeterministic behavior. The execution of parallel applications in most cases is not deterministic and thus not reproducible, for example when two threads contend for a mutex to enter a critical section. Communication be-

tween threads that requires synchronization can lead to different interlacing of threads every time the program is run.

Redundant execution requires determinism to ensure identical redundant processes. However, the state of both redundant processes may be different if the locking order of a mutex varies between both processes. To facilitate the redundant execution of multi-threaded applications, the execution order of the threads and the entrance into critical sections requires an accurate monitoring and control.

The control of redundant threads is independent from the software fault-tolerance (SFT) introduced in Chapter 4. This allows to execute multi-threaded applications on parallel hardware that already provides mechanisms for error detection and recovery. The redundant synchronization mechanisms rely on a specific ordering of the program execution in both redundant processes, which is entailed by the SFT layer. In this chapter, the RMT layer is introduced, which builds upon this ordering, but can reproduce the desired behavior for critical sections by itself if required. Main benefits of the separation of SFT and RMT beside the flexible choice of the underlying hardware are independent and simplified development, as well as an easier but more detailed evaluation.

In this chapter, the execution model for redundant multi-threaded applications is explained. Based on the requirements for redundant concurrency, the creation of new threads and synchronization between redundant threads is described. A recovery mechanism to restore multiple threads is shown in Section 5.4. The approach is evaluated by means of the PARSEC benchmark suite in Section 5.5. Limitations and extensions to this approach for future applications are discussed at the end of this chapter.

5.1 Execution Model

The execution model is nearly the same as for the single-threaded approach: applications with no interference between the threads require only the creation of the threads in the redundant process and can be executed without further assistance. Thread creation and the synchronization between redundant threads is managed by the RMT layer, which is based on the SFT layer for redundant execution (see Figure 5.1). The synchronization mechanisms are handled by the RMT layer, which coordinates the execution of the redundant threads.

The structuring of the program provided by the instrumentation in the SFT layer and the resulting execution order of blocks in the redundant processes is the foundation for synchronization in the RMT layer. Although, software fault-tolerance may not be required if a given multi-core CPU already provides these features itself. In this case, the RMT layer works without FIFO queues between the thread pairs, but uses barriers to ensure the same ordering where needed. This configuration further allows for an independent development, testing, and evaluation of the RMT layer.

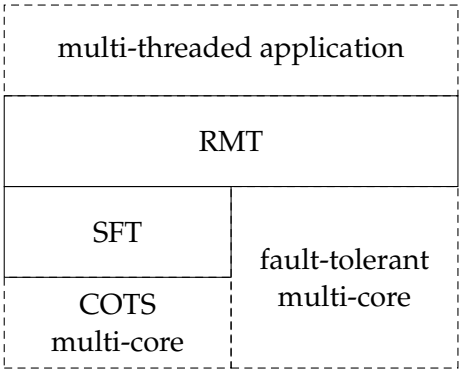


Figure 5.1: The RMT layer builds on the SFT layer for fault-tolerant execution.

For COTS multi-core CPUs, the SFT layer with instrumentation is required to provide a fault-tolerant execution. Every thread in the original process has a corresponding thread in the redundant process that executes the identical instrumented blocks. Both threads of a redundant pair are connected through an individual FIFO queue to send the signatures calculated in the blocks of the leading thread to the trailing thread. Additionally, further information can be exchanged through these FIFO queues, e. g. for synchronization management. The instrumentation phase is unchanged and operates on blocks independent of the thread context the blocks will be executed within. This partitioning into identical blocks in both redundant threads is beneficial for parallel execution in the error-free case as well as during recovery.

Definition 5.1 *A redundant program* $\hat{P}$ *consists of one or more threads* T_i *in the leading and the trailing process:*

$$\hat{P} = \{P, P'\} = \left\{\{T_i\}, \{T_i'\}\right\}$$

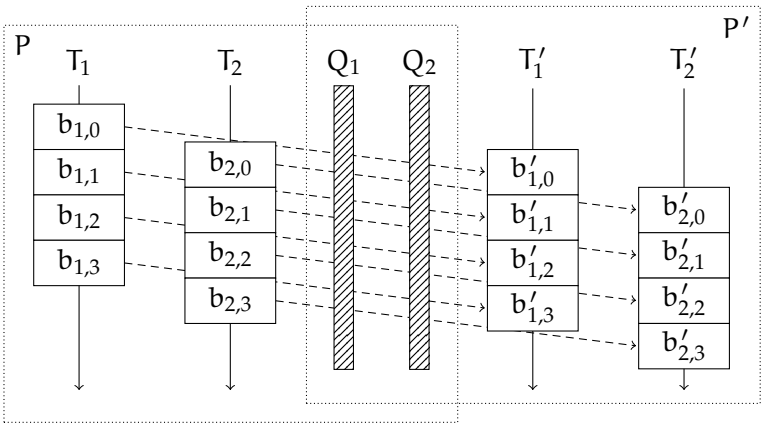


Figure 5.2: A process P with two threads T_1, T_2 and their corresponding redundant threads T_1', T_2' in process P'. A pair of threads T_i, T_i' shares a FIFO queue Q_i.

The executed program, $\hat{P}$, now consist of more than one thread per redundant process (see Definition 5.1). Figure 5.2 shows an application with two threads T_1 and T_2 running in the same context of process P. Their corresponding redundant threads are T_1' and T_2', within the context of P'. A pair of redundant threads (T_1, T_1' and T_2, T_2') exchanges signatures through individual FIFO queues Q_1 and Q_2.

A multi-threaded application where no interaction or influence between any threads occurs can be executed redundantly without any further modifications except the setup of the redundant threads and the signature exchange buffers. However, parallel applications possess at least minimal communication between threads to distribute input data and to gather results for multi-threaded applications. Depending on the application and the kind of parallelization applied on the algorithms, more fine-grained interaction may occur, which requires a deeper investigation of the applied synchronization mechanisms.

To enable a transparent management of redundant threads, existing Pthread function calls are intercepted. This allows the fault-tolerance library to execute the required additional functionality to setup redundant threads and to manage synchronization.

5.1.1 Instrumentation

The instrumentation mechanism described in Section 4.2 is already designed thread-independent. Signatures are accumulatively calculated on *values*, which the compiler backend maps to either registers or stack variables. At the end of each block, the signature is passed as argument on the stack to the block suffix function. The block prefix and suffix functions are provided by the SFT layer, which now require a small modification to support threads: Since the FIFO buffer is referenced to by a pointer in the process memory of the leading and the trailing process, multiple threads would use the same FIFO queue. In the RMT layer, this specific pointer is replaced and stored in *thread local storage*, which now allows each thread to store its own pointer to the FIFO queue.

Figure 5.3 shows a thread T_0, which is either the first thread of the program or, in case of a single-threaded application, the only thread of the program. Its redundant counterpart, T_0', is executed in the trailing process and thus is separated in memory. The instrumentation splits the program into subsequent blocks, which are shown in the

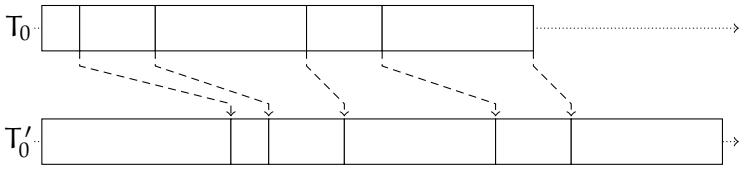


Figure 5.3: A thread sends the signatures of its executed blocks to its redundant counterpart.

figure as hatched boxes. At the borders of the blocks, calls to the block prefix and suffix functions are inserted. In the figure, the block borders with these calls are represented by the narrow lines between the blocks. The block suffix function in the leading thread writes the signature of the previous block into the FIFO queue, where the prefix function in the trailing thread of this block reads the signature from the FIFO queue. The dashed arrows indicate this communication, although without showing the queue.

5.1.2 Concurrency Observations and Restrictions

During the execution of a multi-threaded application, concurrent behavior in two or more threads can be observed. For redundant multi-threaded programs, concurrency can be observed between dependable blocks.

Definition 5.2 *A concurrent execution of two blocks in different threads* T_i *and* T_j *with* $i \neq j$ *where both blocks at least overlap chronologically is described as:*

$$b_{i,k} \parallel b_{j,l}$$

For a concurrency observation, it is not required that the threads of both blocks belong to the same process. One may be a leading thread while the other is a trailing thread. E. g. $b_{0,5} \parallel b'_{0,3}$ describes the overlapping execution of block 5 in leading thread 0 and block 3 in its trailing thread (see Figure 5.4). Further, $b_{0,5} \parallel b_{1,5}$ states that block b_5 is executed in parallel in the leading threads T_0 and T_1.

In contrary to an observation that describes potential concurrent behavior, a concurrency restriction implies that a specific concurrency observation must never occur.

Definition 5.3 *A chronological order between two blocks in two threads* T_i *and* T_j *with* $i \neq j$ *where one block* $b_{j,l}$ *cannot start before another block* $b_{i,k}$ *has started too, is described as:*

$$b_{i,k} \preceq b_{j,l}$$

Definition 5.4 *A chronological order between two blocks in two threads* T_i *and* T_j *with* $i \neq j$ *where one block* $b_{i,k}$ *must finish before another block* $b_{j,l}$ *can start is described as:*

$$b_{i,k} \prec b_{j,l}$$

Theorem 5.1 *If a chronological order between two blocks exists where one must finish before the other block can start, a parallel or overlapping execution of these two blocks cannot occur:*

$$(b_{i,k} \prec b_{j,l}) \vee (b_{i,k} \succ b_{j,l}) \implies b_{i,k} \nparallel b_{j,l}$$

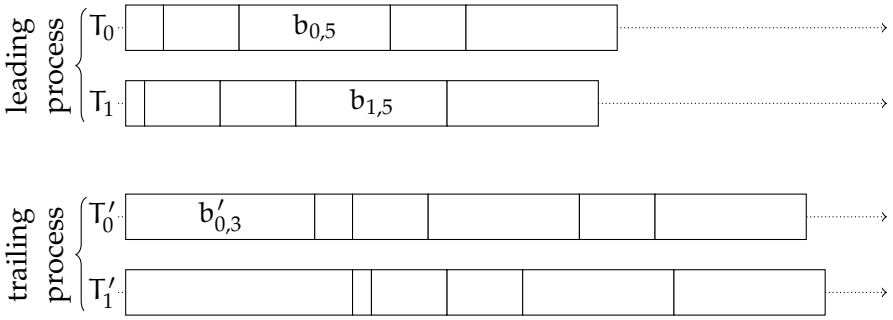


Figure 5.4: Concurrent execution of two leading and trailing threads

5.1.3 Compatibility to Concurrency Restrictions of the SFT Layer

The block-based instrumentation in the SFT layer and the signature exchange through the FIFO queues already restrict the concurrency between associated leading and trailing threads. The FIFO guarantees a strict partial order on the execution of blocks of the same thread and its redundant counterpart: $b_{i,k} \prec b'_{i,k}$. The order within the thread is fixed by the control flow of the program, so $b_{i,k} \prec b_{i,k+1}$. However, the FIFO queue allows the leading thread to advance until the buffer is full. This makes the execution $b_{i,k+n} \prec b'_{i,k}$ with $n > 0$ possible.

If the RMT layer is used without the SFT layer (i. e. without software fault-tolerance and thus without instrumentation and FIFO queues), the subordinated execution of the blocks in the trailing process, $b_{i,k} \prec b'_{i,k}$, has to be enforced explicitly. This is implemented with a buffer per-thread allocated in shared memory, which holds single 64 bit values and is protected by a `ready` flag. With this, the resulting order is identical to a FIFO queue with size 1. Thus, the requirement that trailing blocks cannot be executed before their corresponding leading blocks is satisfied.

Definition 5.5 *Putting a value v into a FIFO queue Q_i is defined by an operator:*

$$\twoheadrightarrow: v, Q_i \quad \rightarrow \textit{puts } v \textit{ into Queue } Q_i$$

Getting the first element out of a FIFO queue Q_i and storing the value in v is defined as:

$$\twoheadrightarrow: Q_i, v' \quad \rightarrow \textit{gets from Queue } Q_i \textit{ and stores in } v'$$

The FIFO operations are written with a simplified notation: $v \twoheadrightarrow Q_i$ to put into the FIFO and $Q_i \twoheadrightarrow v'$ to get from the FIFO. Since reading out of the FIFO queue only occurs in the trailing thread, the destination variable for the signature is written as v' to signalize its location in the process-local memory of the trailing thread. The SFT layer implements $v \twoheadrightarrow Q_i$ and $Q_i \twoheadrightarrow v'$ with a two-stage FIFO queue (see Definition 4.5 and Section 4.3.3).

Algorithm 5.1 Interception of the `pthread_create` function

```
1 _orig_pthread_create ← DLSYM(RTLD_NEXT, "pthread_create")
2                                  ▷ assign address of function in Pthread library

3 function PTHREAD_CREATE(t, r, f, a)
4     ...
5     _ORIG_PTHREAD_CREATE(t, r, f, a)          ▷ calls original pthread_create
6     ...
7 end function
```

5.1.4 Pthread Function Call Interception

Since the Pthread functions are usually linked at runtime through a shared library, all of these functions are outside of the instrumentation scope (see Section 4.2). All Pthread functions that require intervention are intercepted as described in Section 4.3.4. This allows for a transparent management, independent of the actual Pthread implementation. Figure 5.5 shows the interception of the external function `pthread_create`.

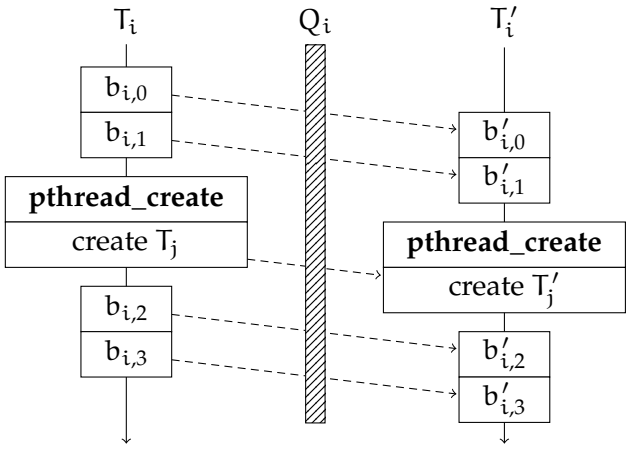


Figure 5.5: Interception of the `pthread_create` function call

The interception of `pthread_create` is described by Algorithm 5.1, which as a result catches any calls to this function and executes `create_thread` instead. This function is responsible for creating a redundant pair of threads.

5.2 Creating new Redundant Threads

Since redundant threads share e. g. a FIFO buffer, an association between the two of them is required. Threads can be dynamically created at run time, thus a static configuration

is not desired. By extending the thread creation mechanism, redundant threads can be created and associated at run time. This is implemented by wrapping the Pthread create method, as described in the previous section.

5.2.1 Thread Data Structure

Additional information must be stored for every single thread and for a redundant pair of threads. On creation of a new thread, a new instance of `rmt_thread_t` is allocated on the heap for the leading and the trailing thread. To support a flexible number of threads, their supplemental data structure is appended to a linked list. Beside the pointer to the next element of the list, the data structure contains a unique thread ID, a pointer to the Pthread thread instance, the pointer to the thread function and its arguments, and a pointer for per-thread statistics. A pointer to the communication data structure, which is placed in shared memory of both threads of the redundant pair, enables to access the FIFO queue from both threads.

Table 5.1: Overview of the thread data structure

Pseudo-code notation	Identifier in source code	Description
d_n	`uint64_t id`	numeric ID of the thread
d_s	`shm_t *shm`	pointer to shared memory
d_t	`pthread_t *thread`	pointer to Pthread object
d_x	`pthread_attr_t *attr`	pointer to Pthread attributes
d_f	`void *(*fn)(void*)`	thread function pointer
d_a	`void *arg`	thread argument
d_l	`fifo_local_t fifo_local`	local part of FIFO queue
d_r	`uint64_t flags`	flags for recovery states
	`counter_t *ctr`	pointer to per-thread statistics
	`char shm_fn[25]`	file name for shm. initialization
	`uint64_t recover_ip`	instruction pointer for recovery
	`uint64_t recover_sp`	stack pointer for recovery

5.2.2 Thread-local Storage

Since each redundant pair of threads has its individual FIFO queue for signature exchange and has additional data that must be stored per-thread, a thread-local storage for such data is required. Instances of the thread data structure are allocated in the heap and thus are accessible by all threads of the process, but not from threads of the other redundant process.

The wrappers for the Pthread methods, as well as the block prefix and suffix functions of the SFT layer, require to access the correct thread data structure. Since the instrumenta-

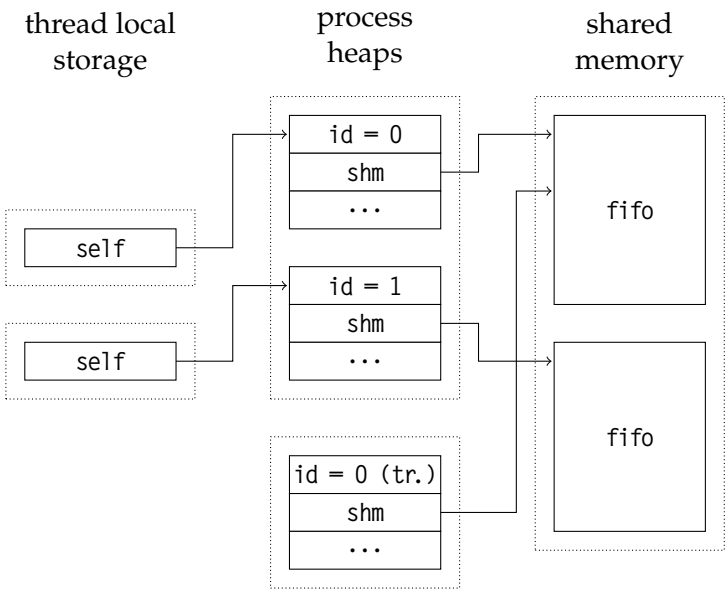


Figure 5.6: Accessing the thread data structure through a pointer in thread local storage

tion is not aware of threads, a pointer in the *thread local storage* references the appropriate data structure. The C library guarantees that this variable can be used per individual thread.

Figure 5.6 shows two pointers, named `self`, of two different threads. The pointers are placed in the thread local storage and thus contain different referenced to the thread data structures located in the process heap. The FIFO buffer is placed in a separately assigned location in the shared memory between the redundant processes to allow both processes to read and write data of the buffer. The thread data structure contains a pointer `shm` to this shared memory location. Each thread in both redundant processes has a thread data structure for itself, with the shared memory pointer referencing the identical FIFO queue. To link the threads of a redundant pair to their FIFO queue, the pointer `shm` in the thread data structure references the same FIFO queue in the shared memory.

5.2.3 Thread Duplicate Creation

The thread creation mechanism is listed in Algorithm 5.2 and the sequential execution in the threads is shown in Figure 5.7. First, memory is allocated to store the thread data structure d. The pointer to the Pthread object, the thread function pointer, and the arguments are assigned to the corresponding elements in the thread data structure (lines 4 – 6). In line 8, the thread ID of the new thread is assigned in the leading thread. Unique thread IDs are guaranteed by an atomic *fetch-and-add* operation. Since the following operations depend on the thread ID, no race conditions can happen if multiple redundant threads are to be created simultaneously. The assigned thread ID of the leading thread

Algorithm 5.2 Create a redundant pair of threads

```
1  function PTHREAD_CREATE(t, r, f, a)          ▷ t: Pthread, r: Pthread attributes,
2                                                 f: thread function, a: arguments
3      d ← MALLOC()
4      d_t ← t
5      d_f ← f
6      d_a ← a
7      if leading then
8          d_n ← thread_ctr++                    ▷ atomic increment of thread counter
9          d_n ↠ Q_i                             ▷ put thread ID into FIFO
10         s ← SHM_OPEN(...)                     ▷ open a shared memory file
11         d_s ← MMAP(..., s, ...)               ▷ map shared memory
12         d_s ↠ Q_i                             ▷ write shared memory address into FIFO
13     else
14         Q_i ↠ d_n                             ▷ get thread ID through FIFO
15         Q_i ↠ x                               ▷ get shared memory address through FIFO
16         s ← SHM_OPEN(...)                     ▷ open a shared memory file
17         d_s ← MMAP(x, ..., s, ...)            ▷ map shared memory with address from leader
18     end if
19     FIFO_INIT(d_s.f, d_l)                     ▷ initialize FIFO queue
20     return _ORIG_PTHREAD_CREATE(t, r, _RMT_PTHREAD_START, d) ▷ create thread
21                                                 with wrapper function
22 end function
```

is sent through the FIFO queue of the thread that creates the new thread (line 9). A file handle is opened to allocate a shared memory segment, which then is mapped into the virtual memory of the process (lines 10 and 11). This mapped address is put into the FIFO queue to let the trailing thread map to the same address.

The trailing thread first takes the thread ID from the FIFO queue and then the address of the mapped shared memory in the leading process (lines 14 and 15). With the thread ID, a file handle to the identical shared memory segment as in the leading thread is opened, and mapped to the identical address (lines 16 and 17). Both threads initialize the new FIFO queue in the shared memory and the local memory used for caching of the FIFO operations (line 19). Then, the actual thread creation method is called in both processes to create the new redundant pair of threads (line 20). To finalize the setup, a wrapper function and the thread data structure are passed to `pthread_create`.

In the thread wrapper function, the pointer `self`, stored in thread local storage, gets the thread data structure assigned (see Algorithm 5.3, line 2). This allows the newly created threads to easily access their FIFO queue and synchronization mutexes and conditional variables. Then, the actual function of the thread is executed, with the argument pointer that was stored in the thread data structure (line 3). Before returning the return value

Algorithm 5.3 Wrapper function for newly created threads

```
1 function _RMT_PTHREAD_START(d)                    ▷ d: thread data structure
2     self ← d                                      ▷ set self pointer
3     r ← d_f(d_a)                                  ▷ executes the function stored in d_f
4     FIFO_FLUSH(d_s.f, d_l)                        ▷ wait until FIFO is empty
5     MUNMAP(d_s)                                   ▷ unmap shared memory
6     SHM_UNLINK(...)                               ▷ delete shared memory file
7     return r
8 end function
```

of the thread function, the mapping of the shared memory is removed, and the shared memory file is deleted (lines 5 and 6). To do this safely, the leading thread has to wait until the trailing thread has finished and thus removed all elements out of the FIFO queue. This is guaranteed by the fifo flush method (line 4) that flushes the local buffer of the leading thread and then waits until the FIFO is empty.

5.2.4 Joining Redundant Threads

Cleaning up after the execution of a thread is simple. Since the function that is passed to `pthread_create` is wrapped by storing the thread function and the argument pointer in the thread data structure and by calling the thread wrapper function instead, the cleanup can be done in the wrapper. The first step is to wait for the trailing thread and to remove the mapping the shared memory for the FIFO queue, as described above. Finally, the thread data structure can be freed, and the thread wrapper function returns. As required by the POSIX standard, a thread has either to be detached or be joined, such call occurs somewhere in the program. Since the thread that creates new redundant threads is already redundant, also if it is the first thread of a program, the calls to `pthread_detach` or `pthread_join` are also executed redundantly. However, they need no further consideration because the cleanup is implemented already in the thread wrapper function.

5.3 Synchronization between Redundant Pairs of Threads

The RMT library implements the most commonly used synchronization mechanisms that are used by Pthread-compatible multi-threaded applications: mutexes, conditional variables, and barriers. Their implementation for redundant threads are described in the following sections.

All redundant synchronization mechanisms rely on the ordering guarantees of the underlying FIFO queues of the SFT layer. If the SFT layer is not to be used with the RMT layer, for example on a fault-tolerant hardware or for evaluation and testing purposes, a small data structure in the shared memory ensures the needed ordering. As described in

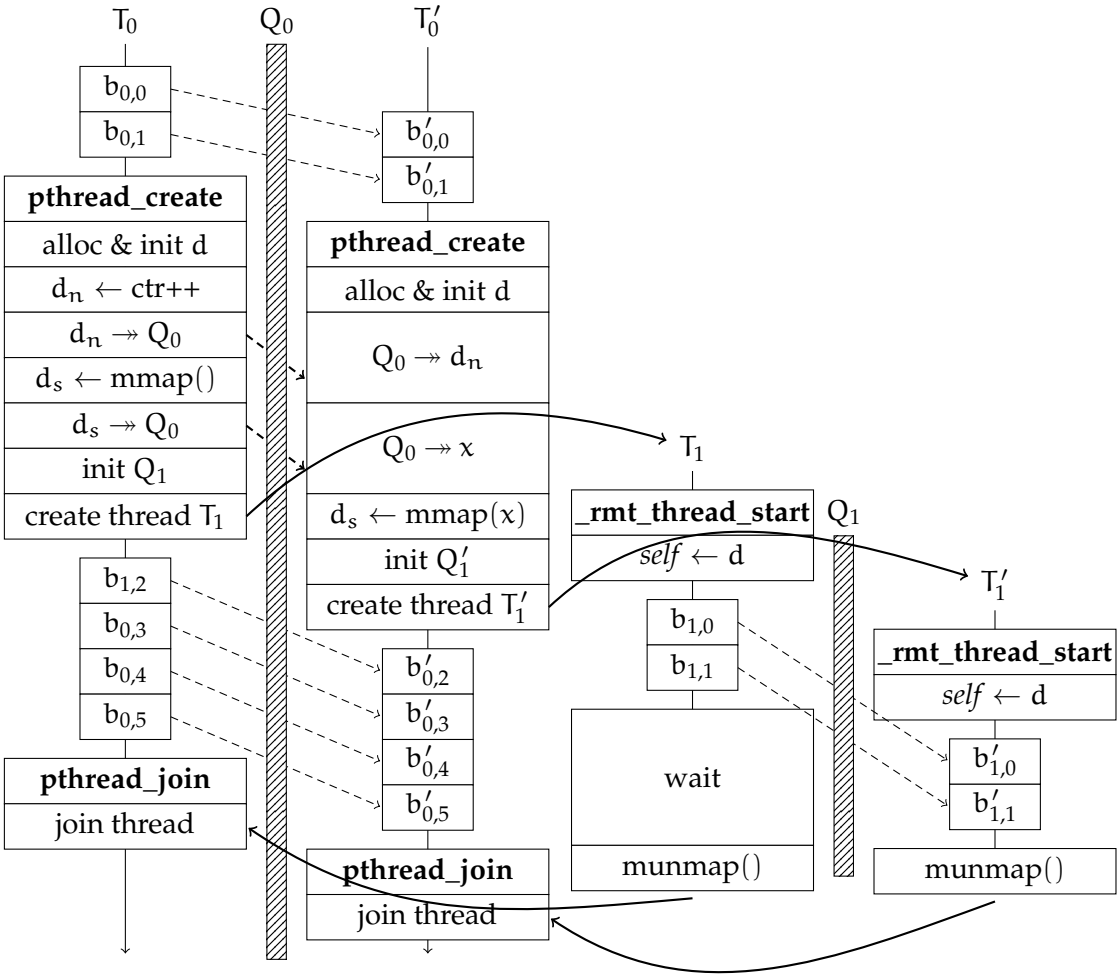


Figure 5.7: Creation of a new redundant pair of threads

Section 5.1.3, it contains a barrier and a 64 bit buffer with a ready flag to exchange the data between the redundant threads.

5.3.1 Barrier

Barriers are used to synchronize the execution of multiple threads at a specific point. All required threads have to wait for each other at this point and can continue if the last thread arrives at the barrier. The number of threads that have to arrive at the barrier is specified during the initialization of the barrier with `pthread_barrier_init`.

The redundant barrier makes use of the ordering that results from the communication through the FIFO queue. Algorithm 5.4 shows the implementation of the redundant

Algorithm 5.4 Redundant waiting on a barrier

```
1  function PTHREAD_BARRIER_WAIT(pthread_barrier_t *w)
2      FIFO_FLUSH(d_{s.f}, d_l)                          ▷ flush the FIFO
3      if leading then
4          r ← _ORIG_PTHREAD_BARRIER_WAIT(w)             ▷ wait for barrier
5          r ↠ Q_i                                       ▷ write into FIFO
6      else
7          r ← _ORIG_PTHREAD_BARRIER_WAIT(w)             ▷ wait for barrier
8          Q_i ↠ x                                       ▷ read from FIFO
9          ASSERT(r = x)
10     end if
11     FIFO_FLUSH(d_{s.f}, d_l)                          ▷ flush the FIFO
12     return r
13 end function
```

barrier. To enforce the synchronization in the leading and the trailing threads at the same time, the FIFO queue is flushed (Line 2). This lets the leading thread wait until the queue is emptied by the trailing thread. Then, the leading threads waits at the barrier until all other required leading threads arrive there as well (Line 4). After the barrier is passed, the leading thread puts the return value into the FIFO queue (Line 5) and flushes the FIFO again before returning (Line 11). The trailing thread also waits at the barrier to ensure that also every required trailing thread arrives at the barrier (Line 7). Then, the value that was put into the queue by the leading thread can be received (Line 8) and compared to the local return value.

An example with two redundant threads waiting at a barrier w is given in Figure 5.8. The redundant threads in the trailing process, T_0' and T_1', wait with a read operation on the FIFO queue Q_i until their counterparts in the leading process arrive at the barrier and put a value into the FIFO.

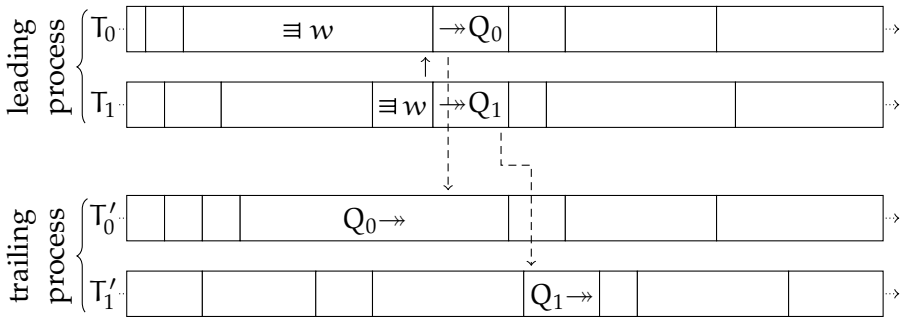


Figure 5.8: Two threads arrive at a barrier $\equiv w$

Listing 5.1 Example with two threads and a counter

```
1 int ctr = 0;
2 pthread_mutex_t m;
3
4 void thread_fn() {
5     pthread_mutex_lock(&m);
6     ctr++;
7     pthread_mutex_unlock(&m);
8 }
```

5.3.2 Mutex

The mutex is the most frequently used synchronization mechanism as it guarantees exclusive access to a locked region to ensure mutual exclusion. Such a region is framed with a call to a `lock` function and an `unlock` function. All threads that intend to enter this region have to call the `lock` function with the same mutex as argument.

The blocks $b_{i,k}$ of one thread T_i are ordered, but only within this thread. There is no global ordering of the blocks of all threads in a process, except that the trailing block $b'_{i,k}$ is always executed after its leading counterpart $b_{i,k}$. Without synchronization between threads, no information on the execution order of two blocks of different threads $b_{i,k}$ and $b_{j,l}$ with $i \neq j$ is available. The general redundancy approach described in Section 4.1.3 requires that redundant pairs of threads execute identical blocks that process identical data. An uncoordinated locking of mutexes for redundant threads would lead to inconsistency between pairs of threads. As a consequence, additional requirements arise for locking redundant mutexes: a mutex must be locked by the same thread in both processes, and both threads of a redundant pair either must get the mutex or not.

For example, see Listing 5.1 with a thread function that increments a global counter variable `ctr` that is protected by a mutex `m`. Let this function be executed in two threads redundantly, then an execution can be observed, where two blocks of different threads are executed while holding a mutex. Thread T_i has mutex m in block $b_{i,k}$ in the leading process, and thread T'_j has the corresponding mutex m' in block $b'_{j,l}$ in the trailing process. Without restricting the order when a thread requests a mutex, these invalid combinations of parallel executions may occur:

$i \neq j \wedge k = l$ The threads that get the mutex are not part of the same redundant pair. This may happen if the threads are scheduled differently in the trailing process. The thread that gets the mutex in the trailing process probably modifies data in the locked

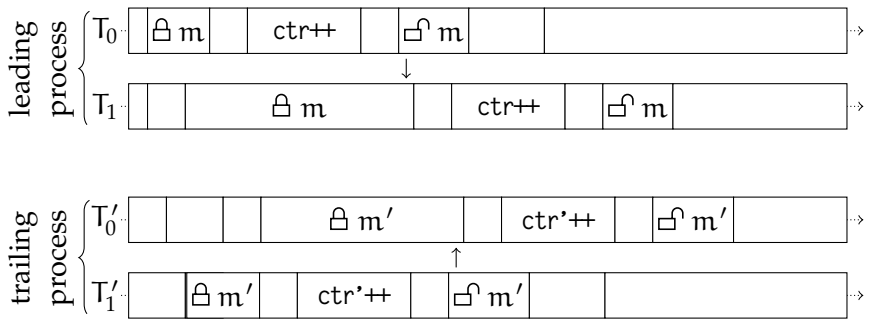


Figure 5.9: Uncoordinated locking of mutexes: In this example, T'_1 may obtain mutex m', but m is held by T_0. As a result, the value of ctr is not equal after the corresponding blocks.

region. If the actual trailing thread, which was intended to get the mutex, reads this data, it is inconsistent to the data read by its leading counterpart. As a consequence, the signatures possibly are unequal and thus are interpreted as an error.

$i = j \wedge k \neq l$ Both threads of a redundant pair get the mutex, but the threads are at different positions in the program flow. The leading thread is able to run ahead of the trailing thread because of the FIFO queue, and as a consequence, it may be at another call to the mutex lock function with the same mutex as argument, e. g. in the next iteration of a loop. This also leads to mismatching signatures, even if the threads are the same, because the mutex and the locked regions are not identical.

$i \neq j \wedge k \neq l$ A combination of the two invalid executions described above. By guaranteeing that both can never occur, this combination is also impossible.

Figure 5.9 shows the uncoordinated locking of redundant mutexes that violates the aforementioned requirements, based on the code in Listing 5.1. In this example, T_0 gets the mutex in the leading process P, but in P′, the trailing thread T'_1 arrives earlier than T'_i at pthread_mutex_lock (represented as m). Thus, the mutex m is hold by T_0 and m' is hold by T'_1, but not by T'_0 as required. During the execution of the threads, the counter ctr is incremented. When the program terminates, the counter has the correct value, independent of the locking order. However, if the counter value is examined for a specific block $b_{i,k}$ for the threads T_i and T'_i, the value of ctr can be unequal to ctr′. This leads to mismatching signatures, which will be wrongly treated as a detected error. In the example, the mutex m is locked concurrently in the blocks $b_{0,i}$ and $b'_{1,i}$, where the counter ctr is incremented. This resembles the first invalid combination, where the blocks are identical, but the threads are different in the leading and trailing process. To

ensure the correct locking of a redundant mutex, the function call interception has to enforce the locking order of the leading process in the trailing process, and the leading thread has to wait until the trailing thread arrives at the same block.

Wrapping mutex objects

To obtain the correct order when a mutex is to be locked, an additional counter is required per every mutex. The number of mutexes used by an application is not limited and not static, so an approach is required to assign counters to mutexes on run time. A mutex is placed somewhere in memory and has a size of 40 bytes, as currently defined by POSIX. Since a `pthread_mutex_t` is an opaque type, the programmer should not assume anything about the content or the size of objects of this type. Figure 5.10 shows an object `mtx` of a mutex, its address (`&mtx`) points at the first byte of the mutex object.

The mutex wrapper w contains 64 bits at offset 0 for a header w_h to hold a defined and constant value when initialized (see Figure 5.11). After that, the actual mutex w_m is placed, followed by the counter w_c. The data of the original mutex is replaced to hold a pointer to the newly allocated location of the mutex wrapper. This allows to access the wrapper with only one additional dereferencing.

The Pthread mutex lock and unlock functions expect a pointer to an initialized mutex as argument, so the wrapper functions can rely on this, too. Since a pointer to the wrapper is expected at the beginning of the mutex object, it must not be `NULL` if the mutex is already wrapped. If the dereferenced mutex pointer contains `NULL`, the mutex was not yet wrapped, which is the case when a mutex is to be locked for the first time. In this case, a new wrapper mutex object is allocated and initialized, and the data of the actual mutex is copied into the mutex wrapper. Then, the data of the original mutex is overwritten with a pointer to the newly allocated wrapper mutex. Algorithm 5.5 shows the procedure of wrapping the mutex.

Locking a Mutex in Redundant Threads

Main objective of the redundant mutex locking mechanism is to ensure the correct ordering of threads entering a critical section that is protected by a mutex. This is implemented by recording the locking order in the leading process and enforcing the identical order in the trailing process. The ordering enforcement consists of two steps: (1) exchange data from leading to trailing block, and (2) wait until it is the current thread's turn to lock.

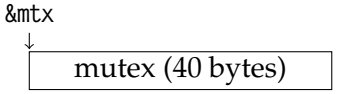


Figure 5.10: A mutex `mtx` as defined in POSIX

Algorithm 5.5 Wrapping a mutex object

```
1  function PTHREAD_MUTEX_LOCK(pthread_mutex_t *mutex)
2      m ← *mutex
3      if m = NULL then
4          w ← MALLOC( )                          ▷ allocate memory for mutex wrapper
5          w_h ← 0xFEDCBA98                       ▷ initialize header
6          w_m ← *mutex                           ▷ copy mutex data
7          w_c ← 0                                ▷ initialize counter
8          *mutex ← w                             ▷ copy wrapper pointer into mutex
9          m ← w                                  ▷ assign mutex wrapper
10     end if
11     ASSERT(m_h = 0xFEDCBA98)
12     ...                                        ▷ actual locking of the mutex
13 end function
```

For an efficient implementation, each mutex has an individual counter that is stored in the mutex wrapper. This counter is incremented every time the mutex is locked. The value of the counter in the leading process is sent to the trailing process (step 1), thus enforcing to the order $b_{i,k} \prec b'_{i,k}$. The trailing process receives the leader's counter value through the FIFO and spins until the counter of the mutex in the trailing process is identical to the counter of the leading process' mutex (step 2). Algorithm 5.6 shows the implementation of this approach after unwrapping the mutex.

The application of the redundant mutex locking mechanism is shown in Figure 5.12 and Figure 5.13. The icons 🔒 and 🔓 symbolize the locking and unlocking of the wrapped mutexes m in the leading process and m' in the trailing process. The waiting for the correct counter value is represented by the symbol ↻.

In Figure 5.12, step 1 of redundant mutex locking is shown. The counter value w_c that is put into the FIFO queue in the leading thread can be received by the trailing thread only after the leading thread locked the mutex. In the example, T'_1 arrives earlier at the mutex lock call, but has to wait until its leading thread locked the mutex and had put the counter into the queue. This allows the thread T'_0 to lock the mutex in the trailing process first, which results in the desired and correct order.

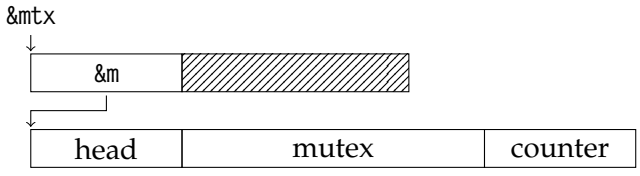


Figure 5.11: Wrapping of a mutex object to store supplementary data

Algorithm 5.6 Redundant locking of a mutex

```
1  function PTHREAD_MUTEX_LOCK(pthread_mutex_t *mutex)
2      ...                                              ▷ test mutex wrapper
3      w ← *mutex                                       ▷ unwrap mutex
4      if leading then
5          _ORIG_PTHREAD_MUTEX_LOCK(w_m)
6          w_c ↠ Q_i                                    ▷ put counter into FIFO
7      else
8          Q_i ↠ c                                      ▷ get counter through FIFO
9          while c ≠ w_c do                             ▷ wait until counters are equal
10             _MM_PAUSE()
11         end while
12         _ORIG_PTHREAD_MUTEX_LOCK(w_m)
13     end if
14     w_c ← w_c + 1                                    ▷ increment counter
15 end function
```

If the leading and trailing threads are more apart from each other, i. e. the counter values of both leading threads are already put in the FIFO queues, step 2 is required to obtain the order. Figure 5.13 shows that T_1' is the first to arrive at the call to the mutex lock function, and both threads T_0' and T_1' do not have to wait for the queue. To resolve this race condition, the counter of the locked mutex in the leading process is compared with the counter of the mutex in the trailing process. In the example, thread T_1' received a higher value for the counter and thus has to wait until the mutex counter w_c' arrives at the expected value. This happens when the thread T_0' locks the mutex and increments the counter. After the atomic increment, the other thread is allowed to lock the mutex, which eventually happens when thread T_0' unlocks it.

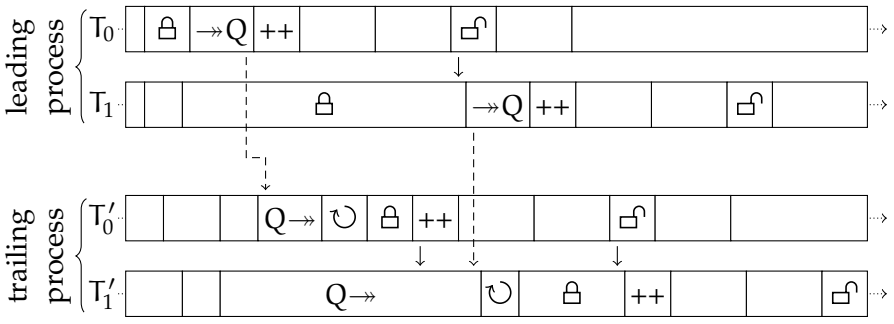


Figure 5.12: Example for step 1 of the redundant mutex locking

Listing 5.2 Typical code that waits for a condition

```
1 pthread_mutex_lock(&m);
2 ...
3 while (v != 1) {
4     pthread_cond_wait(&c,&m);
5 }
6 ...
7 pthread_mutex_unlock(&m);
```

5.3.3 Conditional Variable

Conditional variables are used inside of locked regions that are guarded by mutexes where they wait for a condition to change while a mutex is still occupied to ensure exclusive access on shared data. To allow other threads to change the condition, the mutex is implicitly unlocked while the thread waits on the condition variable. Listing 5.2 shows a typical implementation of a thread waiting for variable v to change to 1. A condition variable c is used to wait for the signal from another thread.

The other thread changes the value of v to 1 and then signals one of any threads that are waiting on condition c (see Listing 5.3). If multiple threads are waiting on the same conditional variable, it is unknown which thread gets signaled by `pthread_cond_signal`. Alternatively, `pthread_cond_broadcast` can be used to signalize all waiting threads.

For redundant threads, the leading and the trailing thread must wait for the same condition to change. For another pair of threads that signalize a changed condition, it is required that the same thread in both redundant processes is signaled.

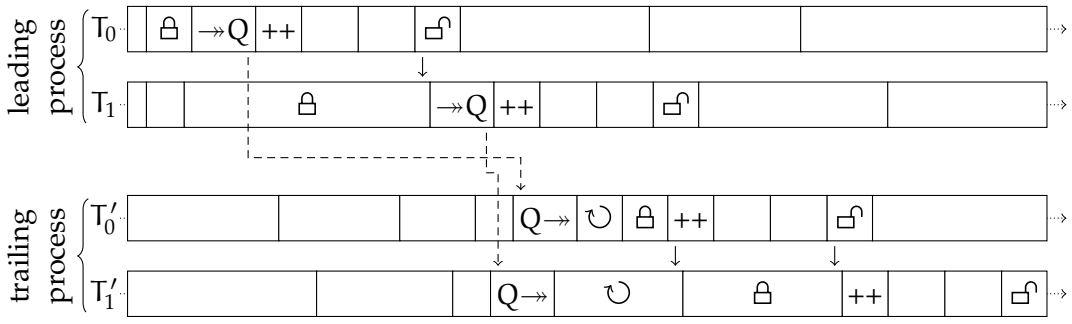


Figure 5.13: Example for step 2 of the redundant mutex locking

Listing 5.3 Typical code that notifies about a changed condition

```
1 pthread_mutex_lock(&m);
2 ...
3 v = 1;
4 pthread_cond_signal(&c);
5 ...
6 pthread_mutex_unlock(&m);
```

Algorithm 5.7 Redundant waiting for a condition

```
1  function PTHREAD_COND_WAIT(pthread_cond_t *c, pthread_mutex_t *mutex)
2      w ← *mutex                                          ▷ unwrap mutex
3      if leading then
4          r ← _ORIG_PTHREAD_COND_WAIT(c, w_m)              ▷ wait for condition
5          r ↠ Q_i                                          ▷ write into FIFO
6      else
7          _ORIG_PTHREAD_MUTEX_UNLOCK(w_m)
8          Q_i ↠ r                                          ▷ read from FIFO
9          _ORIG_PTHREAD_MUTEX_LOCK(w_m)
10     end if
11     return r
12 end function
```

Waiting for a Condition

Since the thread that receives the signal for the condition is unknown in advance, and because it cannot be controlled, the waiting and signaling is handled only by the leading threads. However, it is possible to let a trailing thread wait until its leading thread continues its execution. This can simply be done by communicating through the FIFO queue, as listed in Algorithm 5.7. First, the mutex is unwrapped in line 2 as described in Section 5.3.2. Then the leading thread waits for the signaling of the condition and afterwards puts the return value into the FIFO queue (lines 4 and 5). The trailing thread waits until it can read out of the queue (line 8) and then returns the return value of the leading thread. However, to signal the condition, the protecting mutex has to be unlocked. Before waiting for the data in the queue, the mutex is unlocked and locked again afterwards. Here, the original mutex lock and unlock methods are used, not the wrapped functions with the additional functionality for redundant threads, since the ordering is already ensured by the queue.

Algorithm 5.8 Redundant signaling of a condition

```
1 function PTHREAD_COND_SIGNAL(pthread_cond_t *c)
2     if leading then
3         r ← _ORIG_PTHREAD_COND_SIGNAL(c)          ▷ signal condition
4         r ↠ Q_i                                    ▷ write into FIFO
5     else
6         Q_i ↠ r                                    ▷ read from FIFO
7     end if
8     return r
9 end function
```

Signaling and Broadcasting a Condition

Signaling and broadcasting a condition is simple, since only the leading thread has to be signaled. Algorithm 5.8 shows the implementation for `pthread_cond_signal`, but the broadcast is identical, except that `_orig_pthread_cond_broadcast` is called inside. The return value, which in most cases is zero, is put into the FIFO queue (line 4), where the trailing thread then can read (line 6). Even if the return value may not be required, the communication is important to let the trailing thread wait for the leading thread.

Figure 5.14 shows an example with two redundant threads where T_0 and T_0' wait on a condition variable. The redundant pair T_1 and T_1' sends the signal for the changed condition, which is implemented by waiting on data from the FIFO queue. The threads in the leading process wait on the conditional variable, and signal one thread or all threads, and afterwards write to the FIFO, from which the trailing threads read. Thus, the trailing thread T_0' can only proceed if its corresponding leading thread T_0 received the signal for its conditional variable.

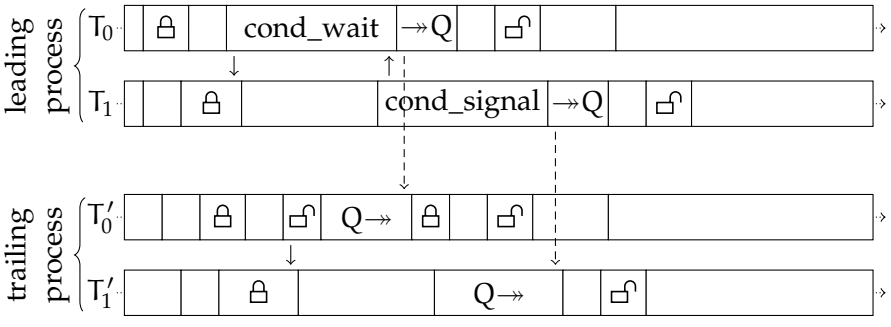


Figure 5.14: Example of waiting for a condition and signaling

5.4 Error Recovery for Multiple Threads in the Software Fault Tolerance Layer

In the previous sections, the redundant execution of multi-threaded application with the RMT layer was described. Indeterministic behavior in the leading and the trailing threads due to synchronization is prevented, which avoids false-positives in the error correction because of mismatching signatures. However, the approach for error recovery that was introduced in Section 4.4.4 is not applicable to programs with multiple threads without further modifications. Additional functionality is required for a correct restoration of all threads of the program. The recovery mechanism of the SFT layer needs to be enhanced to allow the RMT layer to correctly recover all threads.

The redundant multi-threading layer assumes a correct and error-free execution of the redundant threads, which is provided by either fault-tolerant hardware or through the software fault-tolerance layer on COTS hardware. Although, only the RMT layer is aware of the threads of the application, and the error detection and recovery mechanisms of the SFT layer is unaware of threads and their contexts. To enable redundant fault-tolerant execution of multiple threads on COTS hardware with software fault tolerance, both layers require enhancements and an interaction to support error recovery for multiple threads.

The error recovery mechanism for single-threaded applications is depicted in Figure 5.15. A signature of the data of each block b_i is calculated in both the leading process P and the trailing process P'. The signature of the leading process $s(b_i)$ is sent to the trailing process through a FIFO queue Q and received, then a transaction is started in the trailing process. Before committing the transaction, the signature from the leading process is compared with the locally calculated signature $s(b'_i)$. When a signature mismatch is detected, indicating an transient error in one of the processes, the transaction in the trailing process is aborted. After the implicit rollback, the recovery mechanism is triggered, which kills the leading process. Then, a new leading process $P*$ is created with `fork`, letting both the trailing process and the new leading process continue their execution identically. Both processes return from the recovery function and jump to the beginning of the current block. There, the leading process executes the block again and puts the signature into the FIFO queue, from where the trailing process receives the signature and also executes the block again. The transient error is now gone and both processes are recovered, independent of the process in which the error occurred.

This approach works well for the single-threaded case, where the execution is able to resume from this state. However, forking a process that contains multiple threads creates a new process with only a single thread, as defined by the POSIX standard. This thread is the thread which called the `fork` system call. For multi-threaded processes, additional effort is required to migrate all other threads into the new leading process, consisting of four parts: (1) stopping the other trailing threads, (2) aborting and recreating the leading

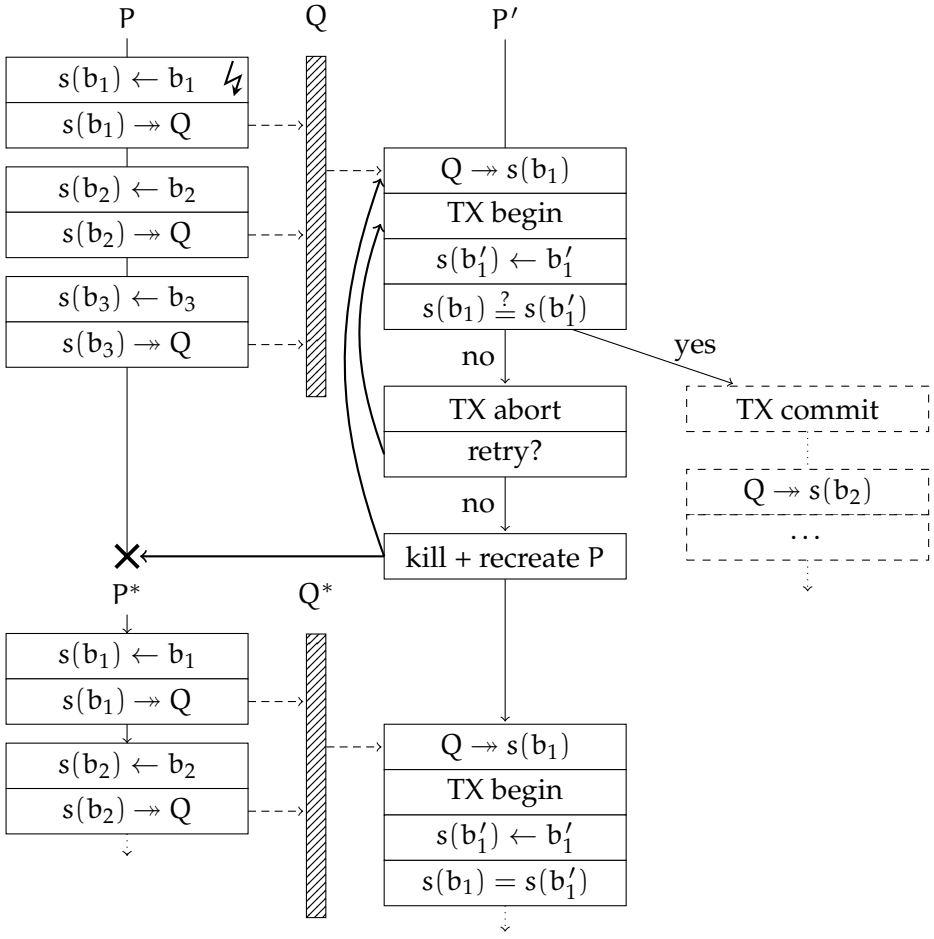


Figure 5.15: Basic recovery approach for single thread application

process, (3) creating the threads in the new leading process, and (4) restoring the thread data. Additionally, the block prefix that is executed at the beginning of every block, needs to be extended.

5.4.1 Thread-Support for Error Recovery

Algorithm 5.9 describes the enhanced version of the prefix function, which is executed at the beginning of a block. It is almost identical to the single-threaded version (see Algorithm 4.14), with an enhancement to the transaction abort handler. The variable that previously stored only the transaction retry flag is now stored as d_r in the thread data structure d (see Section 5.2.1) to be accessible from other threads. A new flag FORCE_ABORT, to be set in d_r of the thread data structure, is introduced to forcefully abort other threads, which is checked when a transaction got aborted (lines 17 – 19).

Algorithm 5.9 Extended block prefix with support for multi-threaded applications

```
 1 function SFT_BLOCK_BEGIN( )
 2     d_r ← d_r ∩ ¬ {ERROR_RETRY}                          ▷ clear retry flag
 3     c ← 0                                  ▷ counter for transaction aborts
 4     BLOCK_BEGIN:
 5         if trailing then
 6             Q_i ↠ x                                  ▷ read from FIFO of thread
 7             TX_RETRY:
 8                 s ← _XBEGIN( )                                ▷ start transaction
 9                 if _XBEGIN_STARTED ∉ s then                ▷ transaction was aborted
10                     if {_XABORT_EXPLICIT, ERROR} ∈ s then              ▷ explicit abort
11                         if ERROR_RETRY ∉ d_r then          ▷ re-execute transaction once
12                             d_r ← d_r ∪ {ERROR_RETRY}
13                             goto TX_RETRY
14                         end if
15                         RECOVER( )                          ▷ call recovery mechanism
16                         goto BLOCK_BEGIN                  ▷ jump back to begin of block
17                     else if FORCE_ABORT ∈ d_r then
18                         RECOVER_THREAD( )                ▷ call thread recovery method
19                         goto BLOCK_BEGIN
20                     else if SHOULD_RESTART_TX(c, s) then ▷ decide if TX should restart
21                         goto TX_RETRY
22                     end if
23                     c ← c + 1                                ▷ increment abort counter
24                 end if
25         end if
26 end function
```

In the abort handler (beginning at Line 9), the transaction usually is restarted if no error is reported (Line 21). However, if the flag for an explicit abort as well as the error detection value is set in the return code of `_xbegin`, the recovery mechanism is triggered. Identical to the single-thread approach, the transaction of the trailing block is aborted if an error is detected. The explicit transaction abort rolls back to the `_xbegin` instruction, where the execution continues in the fall-back path (lines 10 – 23). Unequal signatures signal that an error occurred, but it is unknown if the leading or the trailing thread is affected. If the error happens in the trailing thread, the transient error will not occur again during the re-execution of the transaction, and thus will lead to identical signatures on the next try (lines 11 – 14). As a consequence, the transaction is restarted with the same signature from the leading process, and only if the second try fails again, the actual recovery mechanism is triggered.

5.4.2 Stopping the Trailing Threads

The threads are independent from each other, except when a synchronization mechanism is called explicitly. Without further control, the trailing threads would continue their execution until the FIFO queues are empty and no more signatures can be read. The fast stop of all other threads ensures a correct state when the leading process is recreated and also avoids potential error spreading into other threads. Figure 5.16 shows two pairs of threads with a transient error occurred in block $b_{1,k}$ of leading thread T_1, marked with ↯. This error is detected when the signatures of the leading and its redundant block $b'_{1,k}$ in the trailing thread are compared. In the figure, this block is marked with ⊠.

As described above, the transaction is re-executed once to eliminate transient errors in the trailing process (step 1 in Figure 5.16). If the signatures are still unequal after the second try, the leading process has to be rolled back, and all other threads are required to stop immediately to start the error recovery. The trailing thread that now is required to recover the whole process is henceforth referred to as *recovering thread*. In the general case, the other trailing threads may either be in a transaction, or waiting on an empty FIFO queue until a signature is available to read. A thread may further execute a non-instrumented function or the wrapper function of a synchronization mechanism.

Inside a Transaction

Programs with a high coverage execute trailing blocks inside of transactions most of the time. Since a program instrumentation that leads to a high coverage is most desired, a trailing thread usually is executing a transaction. The principal objective of transactional memory is to abort a transaction when a conflict occurs on data that is in the read- or write-set of the transaction and is modified somewhere else. This is leveraged in the recovery mechanism by reading a variable that is accessible by all other trailing threads. The thread that detected an error in its transactional block writes a *force abort* flag into

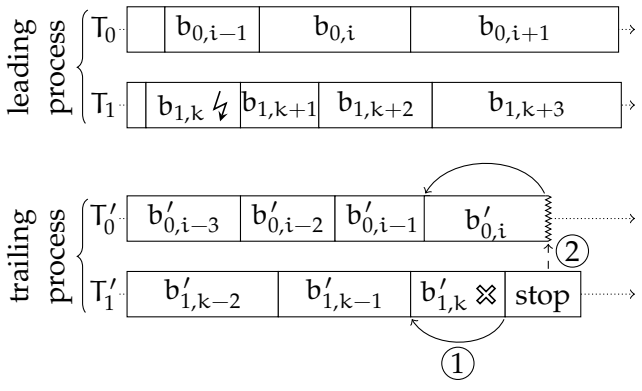


Figure 5.16: Stopping the trailing threads on error recovery

this variable in the fall-back path of its aborted transaction, leading to an abort of all other transactions that are currently running (see step 2 in Figure 5.16, and Line 17 in Algorithm 5.9). These transactions then resume execution in their fall-back path, where the abort status and presence of the abort flag is checked. Then, the thread recovery function is called.

Waiting on an Empty FIFO Queue

A thread may be spinning in the block prefix function due to an empty FIFO queue, although this case occurs rarely because of the leading thread running ahead. However, under certain circumstances the recovery mechanism can suffer a deadlock if one or more threads are waiting for signatures to become available in the FIFO queue. To resolve this, each signature buffer also has a flag variable that can be accessed through the thread data structure and its reference to the FIFO queue. If an *force abort* flag is detected in this variable, the FIFO method sets the local buffer pointers to the next block and thus lets the trailing thread intentionally consume garbage data. This allows the thread to enter the transaction where the abort flag is detected again and the thread recovery method is called. The garbage data that was read from the FIFO is never used to compare signatures, since a new signature is fetched from the buffer after the rollback.

Outside the Sphere of Replication

Functions outside of the sphere of replication are not executed transactional (see Section 4.3.4) and thus cannot be recovered. These functions often have side effects on the process or the overall system, so an abort and a re-execution will potentially result in a different state. The best solution here is to wait until the outside function returns and the prefix function is called again at the beginning of the next block. Then, the abort flag can be detected.

During Synchronization

A thread that is executing a synchronization function, e.g. `pthread_barrier_wait`, is in a similar state as when a function outside of the sphere of replication is executed. Although, waiting until this function returns may not be feasible if it depends on the thread that is currently recovering.

For example, T_0' can be waiting at an barrier for the other threads of the program to arrive there, too (see Figure 5.17). Since the execution of the threads is mostly independent, except for the barrier synchronization, thread T_1' probably executes dependable blocks before reaching the barrier as well. An error that is detected in such a block $b_{1,k}'$ and cannot be corrected by re-executing the transaction, triggers the recovery mechanism, which first stops all other threads. However, thread T_0' is currently not executed since it

is waiting at the barrier, and thus causes a deadlock. First the barrier must be reached by T_1', but the error recovery has to be executed immediately.

All synchronization mechanisms are under control of the RMT layer, as required for the correct synchronization of redundant threads. As a consequence, no calls to functions can exist that are outside of the sphere of replication but also contain dependencies between threads. This allows to implement the required functionality into the synchronization wrapper functions. A particular flag is set before calling the actual synchronization methods to inform the recovering thread that these threads are not required to be stopped. When another thread has to recover while a thread waits for a synchronization event, the thread is already blocked.

Threads receiving the abort flag execute the thread recovery function, which is described in a subsequent section. This functionality is also executed in the synchronization wrapper.

During Recovery

A small chance exists that two threads detect a transient error almost simultaneously. As described by the fault model (see Section 4.1.4), transient errors occur only due to single event upsets, and two errors do never happen at the same time. Although, since errors can only be observed through mismatching signatures, transient errors can only be detected at the end of a dependable block. The recovery mechanism is only relevant for errors that occur in a leading thread, since transient errors in trailing threads are corrected by re-execution of the transactions around the dependable blocks. The signatures that are produced in the leading threads are buffered in a FIFO queue, thereby decoupling the execution of the trailing blocks from the leading blocks. With multiple threads, multiple errors can occur in the leading threads at different instants of time, but their detection in the trailing threads can coincide with each other, resulting in possible race conditions during the recovery mechanism.

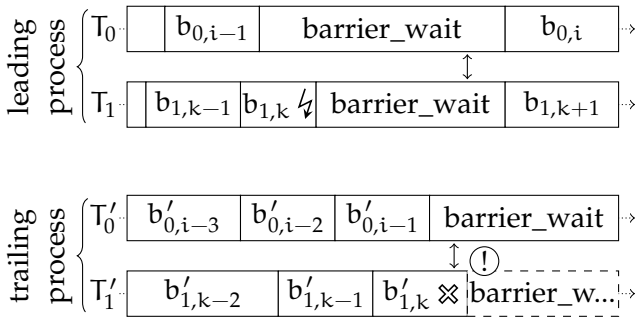


Figure 5.17: An error is detected while another thread is waiting at a barrier, resulting in a deadlock.

This race condition can be avoided with mutual exclusion on entering the recovery mechanism. A process global flag is atomatically set by the first thread that enters the `recover` function. The other threads either see this flag already or fail on writing it, and then call the `recover_thread` function, as the unrelated trailing threads do.

This results in a single thread executing the recovery mechanism, while all other threads block inside of the thread recovery functions. The leading thread recovery of the recovering thread then is sufficient to recover also the erroneous leading threads of the other threads that detected an error.

5.4.3 Abort and Recreation of the Leading Process

The force abort flag to stop all trailing threads is set in the `recover` function that is called by the recovering thread. Algorithm 5.10 is based on Algorithm 4.15 for single-thread recovery, extended with the functionality to stop the other threads (lines 2–7) and recreate them again (lines 13–20). Before killing the leading process, the force abort flag is set in the thread data structure of all threads. Eventually all other threads abort their transactions and enter the thread recovery method (see Line 18 in Algorithm 5.9). Each thread then clears its force abort flag to acknowledge the abort and entrance into the recovery mode.

Algorithm 5.10 Abort and recreation of the leading process

```
 1 function RECOVER( )
 2     for all t ∈ threads do
 3         t.d_r ← t.d_r ∪ {FORCE_ABORT}                          ▷ set abort flag
 4         while FORCE_ABORT ∈ t.d_r do                           ▷ wait until flag is cleared
 5             _MM_PAUSE( )
 6         end while
 7     end for
 8     KILL(P, SIGKILL)                                           ▷ kill leading process
 9     WAITPID(P)                                                 ▷ wait until leading process is stopped
10     MUNMAP(s)                                                  ▷ remove old buffer
11     s ← CREATE_SHM( )                                          ▷ create new buffer
12     P* ← FORK( )                                               ▷ create new leading process
13     THREAD_INIT( )                                             ▷ initialize recovering thread
14     if leading then
15         for all t ∈ threads do                                 ▷ create other leading threads
16             if t.d_n ≠ i then
17                 _ORIG_PTHREAD_CREATE(t.d_p, t.d_x, _RMT_PTHREAD_RECOVER, t)
18             end if
19         end for
20     end if
21 end function
```

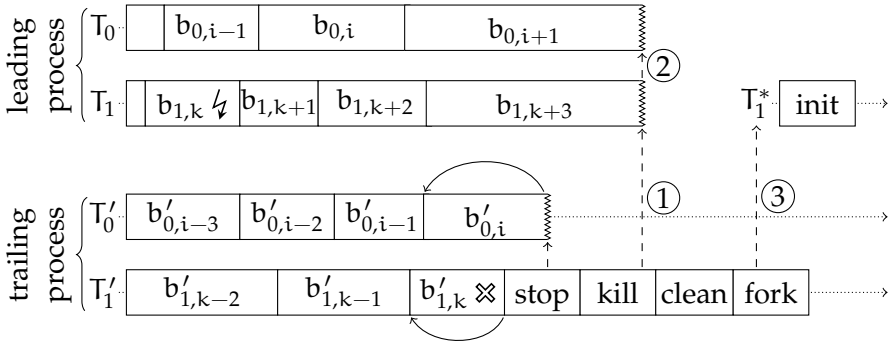


Figure 5.18: Abort and recreation of the leading process

When all trailing threads are stopped, the leading process can be killed by sending the signal `SIGKILL` (step 1 in Figure 5.18). The operating system immediately stops the process and all of its threads (step 2). After completion, the recovering thread cleans up the inter-process shared-memory buffer and creates a new one to be used by the new leading process. This restores the initial state before a new leading process is created. Then the `fork` is issued (step 3) and the shared buffer is initialized.

The thread ID of the recovering thread is preserved to guarantee identical thread IDs beyond the error recovery. Before the execution in the newly created leading process can continue, all other threads have to be recreated and set up appropriately.

5.4.4 Thread Creation in the New Leading Process

When a new thread is created with the wrapped `pthread_create` (see Section 5.2.3), the data structure containing the thread information is appended to a global linked list. During the setup of the redundant execution, the list is created with the initial thread of the program and its data, and new threads are appended accordingly in both processes. Thus, the list and the referenced data structures are identical in both the leading and the trailing process, and the newly created leading process can use the copy of the list and the data of the trailing process.

After the new leading process P^* with its single recovering thread T_i^* is created, the linked list of threads is walked through. The recovering thread is also part of this list, but it may not be the initial thread. Based on the unique thread IDs, the recovering thread can be identified in the list and ignored, while all other threads are created with `pthread_create`. Since further steps are required for restoration, the newly created threads execute an additional wrapper function. The thread recover wrapper function gets a pointer to the thread data structure `t` passed as argument. To create a new leading thread that is identical to the killed thread of the original leading process, the stack and the attributes of the thread are taken from the thread data structure (see Table 5.1 for a list of the elements of the thread data structure).

Listing 5.4 Calling a function in C

```
1 int function1() {
2     int a;
3     ...
4     a = function2(0x05)
5     ...
6     return 1;
7 }
```

5.4.5 Thread Data Recovery

All trailing threads are eventually in the recovery handler, as described in the previous section. Independent of the program execution, it is known that the calling function is either the block prefix `sft_block_begin` or the RMT synchronization wrapper. This allows to use the instruction pointer and the stack pointer from within the recovery handler to restore the threads. A thread context further consists of its registers, which are not needed to be restored explicitly because they are all saved in the stack frame of the calling function.

The thread restoration utilizes the program organization that is defined by the System V AMD64 calling convention [Lu+18, Section 3.2]. Listing 5.4 shows a call to `function2` with an integer argument with value 5. The return value is stored in the variable `a` and the function itself returns the value 1.

The assembler representation of the outside function `function1` is shown in Listing 5.5. The registers `%rbp` and `%r15` are part of the *callee-saved* registers, thus they have to be stored on the stack before modification (see lines 2–3). Return values are passed through `%rax`, which is *caller-saved* and thus has to be saved before calling another function (see Line 6). The first integer arguments are provided through registers, beginning with `%rdi` (Line 7). The `callq` instruction implicitly pushes the instruction pointer of the next instruction on the stack. To return to the calling function, the `retq` instruction pops the instruction pointer back from the stack and continues the program execution at this location.

The stack before calling `function2` is shown on the left side of Figure 5.19. The `callq` instruction pushes the address of the instruction after the call onto the stack, which in the example of Listing 5.5 is Line 9. The called function first pushes the base pointer `%rbp` onto the stack (see stack on the right side in Figure 5.19). Callee-saved registers are then saved on the stack and memory for local variables may be reserved by decrementing the stack pointer. Before returning, the stack pointer is incremented again and the registers are restored. This includes the stack base pointer in `%rbp`. The `retq` instruction then pops the saved address from the stack and lets the execution continue at this location. Now,

Listing 5.5 Calling a function with System V AMD64 calling convention

```
1 function1:
2     push %rbp                          ▷ push base pointer on stack
3     mov %rsp, %rbp                     ▷ save stack pointer
4     push %r15                          ▷ push other registers, r15 holds variable a
5     ...
6     push %rax                          ▷ backup register on stack
7     mov $0x5, %rdi                     ▷ store value 5 in register (argument)
8     callq function2                    ▷ call pushes IP of Line 9 on stack
9     mov %rax, %r15                     ▷ move result into Register 15
10    pop %rax                           ▷ restore register
11    ...
12    mov $0x1, %rax                     ▷ store return value in register
13    pop %r15                           ▷ pop other registers
14    pop %rbp                           ▷ restore base pointer
15    retq                               ▷ return to caller (pop IP from stack)
```

Listing 5.6 Backup stack and IP

```
1 mov  %rbp,  %rax                       ▷ stack pointer of caller
2 lea (%rip), %rbx                       ▷ current instruction pointer
```

Listing 5.7 Restore stack and IP

```
1 mov  %rcx, $0x8(%rbp)                  ▷ write IP in stack
2 mov  %rdx, %rbp                        ▷ replace stack base pointer
3 mov  %rbp, %rsp                        ▷ reset stack-frame to stack base
4 pop  %rbp                              ▷ reset previous stack base pointer
5 retq                                   ▷ return
```

the stack pointer in %rsp points again at the entry which holds the data of %rax (the fourth element from top in the stack on the left side in Figure 5.19).

The stack pointer and the instruction pointer can be saved into another register, as shown in Listing 5.6. The stack base pointer can be copied directly into another register (Line 1), and the instruction pointer can be copied with lea (load effective address), since a direct copy of %rip is not possible in x86.

With the knowledge of the calling convention, both stack pointer and instruction pointer can be restored to continue the execution at the saved location with the old stack. Basically, the target address of the return instruction retq is replaced on the stack, and the cleanup and return of the wrapper function for thread recovery is implemented manually with inline assembler. The stack base pointer %rbp is set to the stack pointer at the beginning of a function (see Line 3 in Listing 5.5). Thus, writing to the memory referenced to by the stack base pointer sets the stack pointer of the calling function. Eight bytes above in the stack, relative to the stack base pointer, the return address of the call is stored (see the dashed arrow in Figure 5.19). By writing to this address, the destination address for the return instruction can be modified. In Listing 5.7 the instruction pointer to be restored is assumed to be saved in register %rcx and the stack pointer is expected in %rdx. Before the return instruction is reached, the stack of the currently executed function has to be cleaned up. This can be achieved by simply copying the stack base pointer back into the stack pointer. Due to the calling convention, the stack base pointer that has been pushed on the stack at the beginning of the function, is restored by popping it off the stack. Finally, the return instruction jumps to the desired address specified in the register %rcx. After the return, the stack is in a correct state, as if the function would have returned as originally intended.

In the wrapper function for pthread_create, the stack is explicitly allocated and the address is saved in the thread data structure to guarantee identical stack addresses in both processes. This allows the newly created leading threads to use the same stack, since the same mapping is done in the leading and the trailing process, and thus persists after forking the new leading process. The newly created leading threads then operate on an implicitly created copy of the stack of the corresponding trailing threads, due

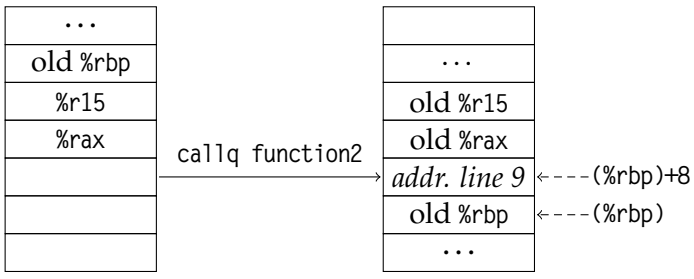


Figure 5.19: The stack before calling function2 (left) and inside the function (right)

to *copy-on-write* of the memory pages that have been duplicated at the fork. Thus, an explicit copying of the stack data from the trailing to the leading threads is not required.

The leading threads are recovered only when an error that was detected in the trailing thread occurred in its corresponding leading thread, since the trailing transaction is re-executed to eliminate potential errors in the trailing thread. All stacks of the leading process are quashed when the process is killed, and copied back from the trailing process when the new leading process is forked. As a consequence, the stacks of the new leading threads are guaranteed to be free of errors, according to the fault model specified in Section 2.1.4.

Default Thread Recovery

As explained in Section 5.4.4, the newly created leading threads execute a wrapper function to finalize the thread restoration. This wrapper function is described by Algorithm 5.11. The recovering thread controls the restart of the re-created threads through the flag RECOVER_GO, to ensure the correct restoration order. With the mechanism shown in Listing 5.7, the contexts of the re-created leading threads are restored, and their execution then continues at the previously saved instruction. The address of the instruction is obtained through the stack frame and is implicitly set when the `return` instruction is executed. In the general case, the instruction pointer was saved inside of the SFT block prefix function. The threads return from `recover_thread` and then jump to the label `BLOCK_BEGIN` (Line 4 in Algorithm 5.9). There, the actual separation into leading and trailing thread is done by evaluating the value of `is_leading`, to restore the behavior of one thread that runs ahead, putting signatures into the FIFO queue, and the other thread that compares the signatures inside of a transaction.

Algorithm 5.11 Thread restoration wrapper

```
 1 function _RMT_PTHREAD_RECOVER(t)                ▷ t: thread data structure
 2     self ← t                                          ▷ set self pointer
 3     while RECOVER_GO ∉ t.d_r do                  ▷ wait until flag is set
 4         _MM_PAUSE()
 5     end while
 6     t.d_r ← t.d_r ∩ ¬ {RECOVER_GO}                          ▷ clear flag
 7     restore stack pointer and instruction pointer
 8     pop stack frame
 9     return                          ▷ return to address saved on stack
10 end function
```

Recovering Synchronizing Threads

A special case for returning from the recovery mechanism is when a trailing thread is synchronizing when the error recovery is triggered. As described in Section 5.4.2, threads that enter a synchronization function set a flag upon entering the RMT wrapper for a synchronizing function. Additionally, the stack pointer and the instruction pointer are saved in the thread data structure, using the method shown in Listing 5.6. One or more trailing threads, except the recovering thread, may execute a synchronization function when the leading threads are about to be recreated. This is possible, since synchronization in the trailing threads cannot be aborted, as explained in the above mentioned section. However, the new leading threads are required to be recovered at a boundary of their dependable blocks.

The encapsulation of the synchronization functions by the RMT layer allows to return into this wrapper function instead of the SFT prefix function. The program then still operates correctly, since synchronization is always between dependable blocks, and the called synchronization mechanism is executed again after recovery.

5.4.6 Summary

Putting the steps depicted in the previous sections together, a restart mechanism for the threads of the leading process can be crafted (see Figure 5.20). The leading process has completely been killed with all its threads, and all trailing threads have aborted their transactions (or further operations, as described above). The trailing threads then store their instruction and stack pointers into the thread data structure (see Step 1). Then, a new leading process is forked from the recovering thread, which then is responsible to re-create all other leading threads (see Step 2). The wrapper function of the new leading threads adjusts the stack frame (Step 3) and thus lets the threads continue their execution at the same dependable blocks as their counterparts in the trailing thread (see Step 4).

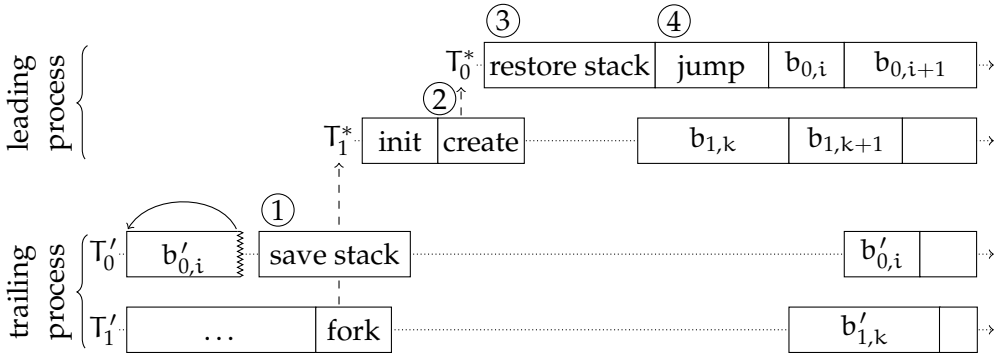


Figure 5.20: Restoring the threads in the leading process

The trailing threads continue their execution after the fork of the new leading process. However, the thread recovery functions return to the SFT block prefix function, where the threads now wait until signatures become available through the FIFO queue. Thus, they have to wait until the new leading threads are fully restored and continue to execute dependable blocks. The error recovery then is completed, and all threads operate the same way as they did before the error occurred. Through re-execution of the trailing transaction and the re-creation of the whole leading process, a correct and error-free state for all threads can be guaranteed.

5.5 Evaluation

The support for redundant multithreading through the proposed RMT extension imposes additional overhead to the fault-tolerant execution. This section gives an estimation of the additional costs by means of an evaluation of the approach with benchmarks of the PARSEC suite [Bie+08]. The minimal overhead of RMT, consisting of the interception of the synchronization functions, is observable in the single-threaded executions. Multiple threads lead to varying performance overhead, resulting in a specific speed-up behavior of the benchmarks. In this section, the influence of the synchronization mechanisms, which RMT is capable to handle, is investigated. Only the impact of the RMT library, which handles thread creation and synchronization in redundant processes, on the performance and the speedup is evaluated, without the underlying SFT library and the instrumentation. The benchmarks are also executable fully fault-tolerant with instrumentation and transactional wrapping, but a separate evaluation allows to estimate the overhead of the redundant synchronization mechanism.

5.5.1 Evaluation Environment and Methodology

The presented approach was evaluated on a dual-socket workstation with two Intel Xeon E5-2697 v4 "Broadwell-EP", with turbo-boost and SMT (hyper-threading) disabled to avoid transaction capacity problems. The workstation runs a GNU/Linux operating system with kernel version 4.15, the LLVM toolchain was used in version 7.0.0.

To test the approach for redundant execution of multi-threaded programs, benchmarks of the PARSEC benchmark suite [Bie+08; Bie11] have been selected. For all benchmarks, the native input set was used, as suggested for benchmarking on real hardware. Most of the PARSEC benchmarks have a configuration for compilation and execution with Pthreads, while some are only available for OpenMP. To select the appropriate benchmarks, the use of the Pthread synchronization mechanism in the benchmarks was evaluated with the `ltrace` tool, which records a trace of library calls. The resulting numbers from the counted occurrences of the Pthread functions during the benchmark execution with eight threads are listed in Table 5.2. Many of the benchmarks are *em-*

barrassingly parallel, i. e. no synchronization is required, since either no communication exists between the threads, or the communication is trivial. This is the case for `blackscholes`, `ferret`, `streamcluster`, and `swaptions`. However, especially `swaptions` is still interesting, since this benchmark repeatedly creates and joins new threads. In contrary, `ferret` creates a multiple of the specified threads, as discussed in the next section. The benchmark `facesim` stands out, as it executes a large number of lock operations, as well as conditional waits.

As in Section 4.6, the original configuration for the benchmark compilation is denoted by `O`, and `R` specifies the RMT configuration. Since no instrumentation is required for the RMT configuration, the benchmark compilation parameters are identical to the original configuration for `clang`, except the specification of the RMT library in `$LD_FLAGS` to insert the dependency to the shared library. The management script of the PARSEC suite was modified to evaluate the benchmark similar to the SPEC benchmarks, to obtain a comparable time measurement for both configurations.

5.5.2 Speedup Comparison and Performance Overhead

The speedups for the benchmarks were calculated independently for the original and for the RMT configuration. Thus, the speedup for 1 thread is 1.0 for both configurations. Since the speedup of a parallel program for different numbers of threads is a good measure of the scalability of the program, it is also useful to evaluate the impact of the redundant synchronization mechanism.

Figure 5.21 shows the speedup graphs for the selected benchmarks, with the number of threads omitted for brevity. Each benchmark was executed with 1, 2, 4, 8, and 16 threads, and the corresponding speedup value is marked by a dot for the original configuration ($s_O(n)$), and with a square for the RMT configuration ($s_R(n)$), respectively, for n threads. It is noticeable that the speedup of the RMT configuration is almost the same as for the original configuration, at least until the eighth thread for `bodytrack` and `facesim`. The `ferret` benchmarks creates a multiple of the specified threads, i. e. when started with 8

Table 5.2: Calls to synchronization functions in the selected PARSEC benchmarks with 8 threads

Benchmark	Thread Create	Mutex Lock	Barrier Wait	Condition Wait
blackscholes	8	0	0	0
bodytrack	9	550	524	1
facesim	7	78,784	0	1,628
ferret	34	0	0	0
streamcluster	48	0	0	0
swaptions	8	0	0	0

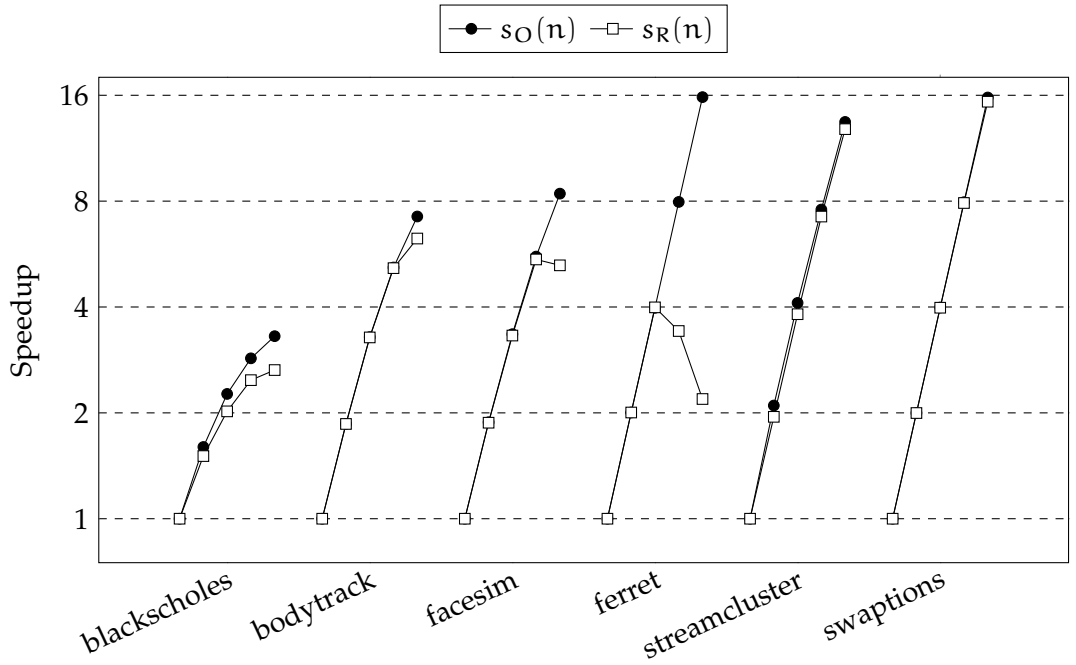


Figure 5.21: Comparison of the Speedups of the PARSEC Benchmarks with $n = \{1, 2, 4, 8, 16\}$ Threads

threads as parameter, the benchmark effectively runs 35 threads in parallel, and with 67 threads when started with a thread count parameter of 16. This is the reason for the declining speedup for the RMT configuration with 8 and 16 threads, since the system has 36 CPU cores, which is sufficient for the original benchmark, but the total number of 70 threads for the configuration with nominal 8 threads cannot be executed in parallel on the system. This further reduces the speedup for the execution with nominally 16 threads, which effectively runs 140 threads in the RMT configuration.

The speedup for `blackscholes`, which has no synchronization mechanisms is also listed in Table 5.4. The benchmark `facesim` locks mutexes and uses conditional variables, and its speedup values are shown in Table 5.5. In Table 5.6, the speedup of `ferret` is listed. The tables also show the execution times in seconds for both configurations ($t_O(n)$ and $t_R(n)$), and the relative execution time. This factor $t_R(n)/t_O(n)$ describes the performance overhead due to the redundant synchronization mechanisms, compared to the original configuration with the same number of threads n. In Figure 5.22, the relative execution times for the selected benchmarks and for 1 to 16 threads is depicted, with the first dot representing $t_R(1)/t_O(1)$, the second dot the corresponding value for two threads, and so forth.

The overhead of additional synchronization between the redundant threads, which is required for the mutex lock operation, can be seen in the decreasing speedup of `bodytrack`

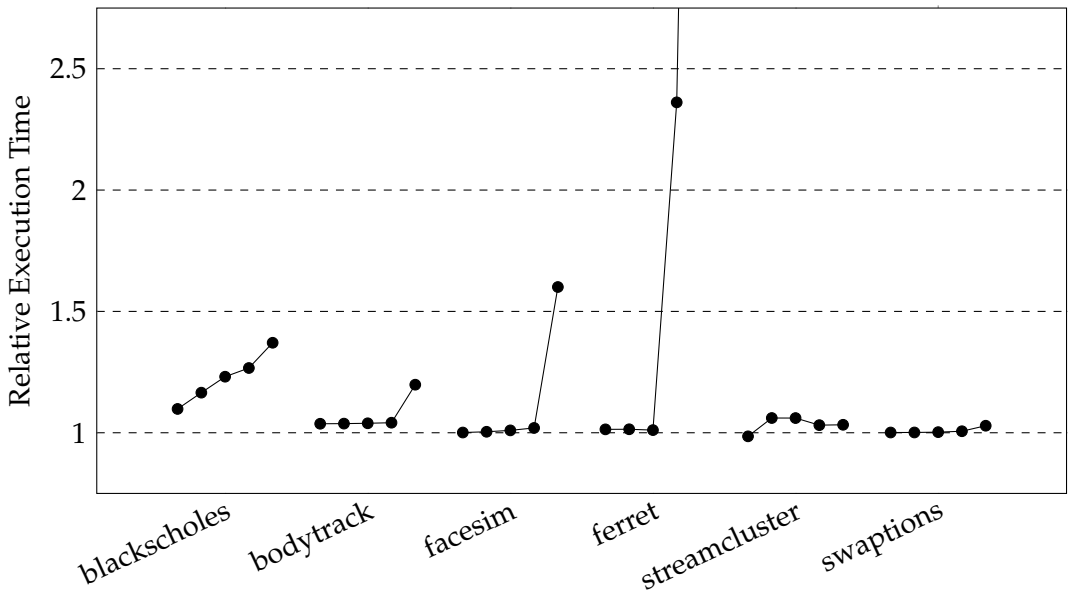


Figure 5.22: Performance of the redundant execution compared to the original benchmark execution

and `facesim` for $n > 8$ threads. Locking a mutex in the trailing process requires a thread to wait its turn, and an increasing number of threads rises the chances of a different order of calls to the mutex lock function in the trailing threads. Thus, more threads are potentially required to wait. Table 5.3 lists calls to the synchronization functions in the `facesim` benchmark in detail for $n = \{1, 2, 4, 8, 16\}$ threads. This benchmark also executes waits on a conditional variable, which implicitly unlocks and locks a mutex, and thus contributes additionally to the redundant synchronization overhead.

Table 5.3: Calls to synchronization functions in `facesim`

Threads n	Thread Create	Mutex Lock	Condition Wait
1	0	36,962	0
2	1	43,122	1,038
4	3	55,478	1,862
8	7	81,713	1,678
16	15	142,643	4,025

Table 5.4: Speedup and performance overhead of `blackscholes`

Threads	Ex. Time [s]		Speedup		Rel. Ex. Time
n	$t_O(n)$	$t_R(n)$	$s_O(n)$	$s_R(n)$	$t_R(n)/t_O(n)$
1	90.20	99.05	1.00	1.00	1.10
2	56.40	65.71	1.60	1.51	1.17
4	39.84	49.04	2.26	2.02	1.23
8	31.56	39.97	2.86	2.48	1.27
16	27.30	37.42	3.30	2.65	1.37

Table 5.5: Speedup and performance overhead of `facesim`

Threads	Ex. Time [s]		Speedup		Rel. Ex. Time
n	$t_O(n)$	$t_R(n)$	$s_O(n)$	$s_R(n)$	$t_R(n)/t_O(n)$
1	235.27	235.49	1.00	1.00	1.00
2	125.07	125.58	1.88	1.88	1.00
4	70.27	70.96	3.35	3.32	1.01
8	42.28	43.12	5.56	5.46	1.02
16	28.00	44.81	8.40	5.25	1.60

5.5.3 Error Recovery

The recovery mechanism described in sections 4.4.4 and 5.4 requires time to restart the leading process and thus entails an impact on the performance. In this section, the average amount of time that is required to recover from an error is evaluated.

Triggering the Error Recovery Mechanism

To measure the execution time overhead of the error recovery, a reliable mechanism is required to trigger the error detection. One possibility is to randomly inject faults into the memory of the tested application. However, this rather helps to test the error coverage, but does not lead to a reproducibly triggered error recovery. The SFT library allows to enforce the recovery mechanism for a specific thread T_i with a given block number k. This leads to an abort of the block $b'_{i,k}$ on every execution of the program. Further, the simulated error persists after the first restart of the transaction and thus triggers the recovery mechanism.

Error Recovery Benchmark

The benchmark itself does not significantly influence the evaluation result, since in the general case, all trailing threads are executing dependable blocks wrapped in a

Table 5.6: Speedup and performance overhead of ferret

Threads	Ex. Time [s]		Speedup		Rel. Ex. Time
n	$t_O(n)$	$t_R(n)$	$s_O(n)$	$s_R(n)$	$t_R(n)/t_O(n)$
1	394.06	399.58	1.00	1.00	1.01
2	196.50	199.32	2.01	2.00	1.01
4	99.03	100.11	3.98	3.99	1.01
8	49.50	116.90	7.96	3.42	2.36
16	24.93	182.47	15.81	2.19	7.32

transaction, which can be aborted immediately. To measure the recovery overhead for one pair of redundant threads, up to N pairs, a simple binary sort algorithm with a fixed input is executed. Each thread sorts an integer array with constant size, independent of the number of threads.

Table 5.7 lists the time in milliseconds that was spent in the recovery mechanism of the recovering trailing thread for a benchmark program that consisted of n redundant threads. The graph in Figure 5.23 shows an almost linear curve, when considering the logarithmic scale of the x axis, indicating that the recovery mechanism is scalable for a larger number of threads. The dependency of the recovery time on the number of threads stems from the waiting time of stopping the trailing threads, and from the sequential thread data recovery.

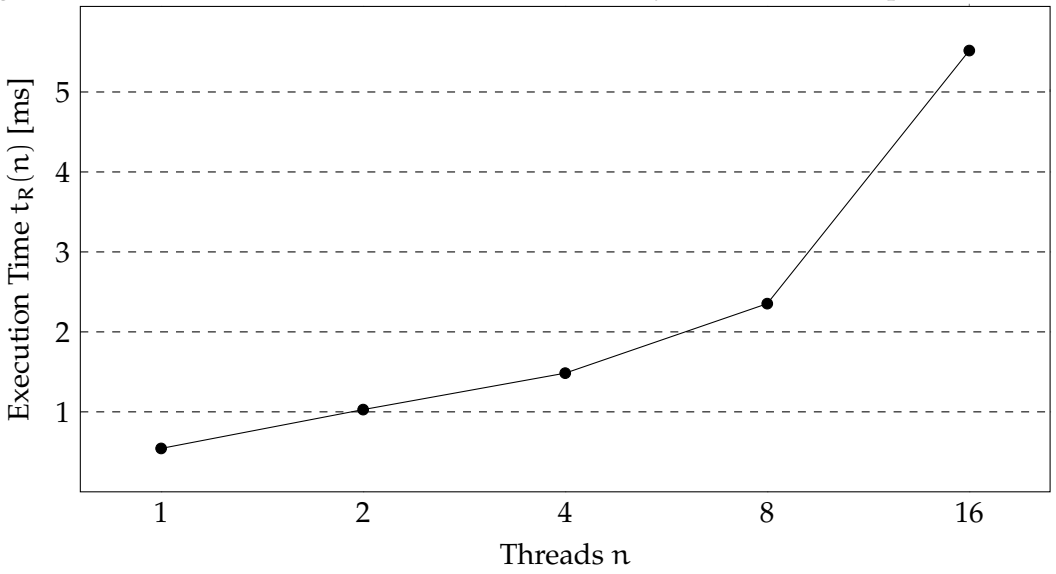
5.6 Limitations

The RMT layer enables the redundant execution of multi-threaded applications that rely on the synchronization mechanisms of the Pthread library. As a consequence, other kinds of parallel programs, for example those implemented with OpenMP, can still not be executed redundantly without further adaption, if a deterministic execution in both redundant processes cannot be guaranteed otherwise.

Table 5.7: Execution time overhead for error recovery with $1 \leqslant n \leqslant 16$ pairs of threads

Threads n	Ex. Time $t_R(n)$ [ms]
1	0.54
2	1.03
4	1.48
8	2.35
16	5.52

Figure 5.23: Execution time overhead for error recovery with $1 \leqslant n \leqslant 16$ pairs of threads



Synchronization mechanisms rely on atomic operations like *compare-and-swap*, but these can also be used directly in data structures or in the program, for example in lock-free data structures. Such atomic operations that are used directly in the program without encapsulation through the Pthread library are not supported until now. However, atomic operations are typically encoded as a single machine instruction, and thus the instrumentation tool, which is used in the SFT layer, could detect these instructions, and insert a interception call to the RMT layer instead. There, the redundant threads can be synchronized to ensure determinism in both redundant processes.

Since the hardware transactional memory is already used to wrap dependable blocks to obtain checkpoints for error recovery, optimistic synchronization with the traditional use of transactional memory is not possible without further provisions. The transaction instructions are part of the instruction set architecture, and thus the instrumentation mechanism could consider already existing transactional regions in the program by adjusting the begin and end of the dependable blocks accordingly. An additional distinction of the transaction aborts is required to correctly handle conflicts, which are currently irrelevant for the error detection mechanism. However, the redundant execution of transactions for optimistic synchronization demands further investigation to avoid diverging executions of the redundant pairs of threads, similar to the approach for mutual exclusion with locks, as described in Section 5.3.2.

5.7 Summary

This chapter described the RMT layer, which allows the redundant execution of multi-threaded programs. The synchronization functions of the Pthread library are intercepted by the sphere of replication, and the additional synchronization between the redundant threads ensures the identical order of the execution of critical sections in both processes. For this, a redundant thread has an auxiliary thread data structure to hold the references to the shared memory, and the FIFO buffer, which is separate per pair of redundant threads. The typically used synchronization mechanisms are support: barriers, mutexes, and conditional variables. The error recovery mechanism is extended to support the stopping of multiple threads in the trailing process, and to restore all threads in the new leading process, to fully restore all redundant pairs of threads.

6

Summary and Conclusion

Contents

This chapter summarizes the approaches and methods described in this thesis, and provides a look on future work. At the end, a conclusion of this thesis is given, based on the evaluation results.

6.1 Summary

Due to the increasing error rates that result from transient faults, countermeasures are also needed for future commodity multi-core processors. The traditional cycle-by-cycle lockstep execution is not feasible on such architectures, and thus new approaches are required. This was discussed in the introduction of this thesis, and in the background chapter (see chapters 1 and 2). Different techniques have been proposed to

loosen the tight cycle-by-cycle coupling of lockstep processors, both software and hardware approaches, as described in Chapter 3. Hardware transactional memory enables software-based fault-tolerance mechanisms to leverage the checkpointing mechanism of HTM to implicitly generate a checkpoint that can be used for rollback in case of a detected error (see Section 3.3).

A method for the fault-tolerant execution of individual applications on an x86 COTS processor was proposed in Chapter 4, where software-managed process-level redundancy is combined with an existing hardware transactional memory to provide a checkpointing mechanism for error recovery. The resulting hybrid hardware/software approach allows to detect transient errors in applications that have been instrumented during compilation with instructions for checksum generation. An optimization pass for the LLVM compilation toolchain was developed, where functions are split into dependable blocks. CRC instructions are inserted to generate a checksum of the critical values of each dependable block. These checksums resemble signatures of the dependable blocks, and by sending them from the leading process to the trailing process for comparison, errors in the execution of the dependable blocks can be detected. Error recovery is enabled through transactional wrapping of the trailing process, which provides an error-free checkpoint to roll back to. The functionality for signature exchange, process duplication, error detection, and recovery is implemented in a shared library. The applicability of the approach was evaluated with benchmarks of the SPEC2017 benchmark suite, and the results show a relative execution time of less than twice the time of the original benchmark execution. To improve the performance of the fault-tolerant execution on a COTS processor, different hardware extensions were proposed and investigated in the evaluation.

The indeterminism in multi-threaded applications requires special consideration in redundant systems, and thus a interface to the Pthread library for controlled synchronization between redundant pairs of threads was described in Chapter 5, as well as a mechanism for multi-threaded error recovery. The functional capability of these additions to the software fault-tolerance library were evaluated with benchmarks of the PARSEC suite. Additional synchronization, which is required for redundant locking of mutexes, and waiting on barriers and conditional variables, impacts the overall performance only for a higher number of threads and mutexes or conditional variables. Further, the error recovery mechanism for a multi-threaded program was shown to scale linearly with the number of threads.

6.2 Outlook on Future Work

The approach for fault tolerance with hardware transactional memory as presented in this thesis can be further extended. As the evaluation of the software fault tolerance mechanism showed in Chapter 4, different extensions to the hardware can increase the

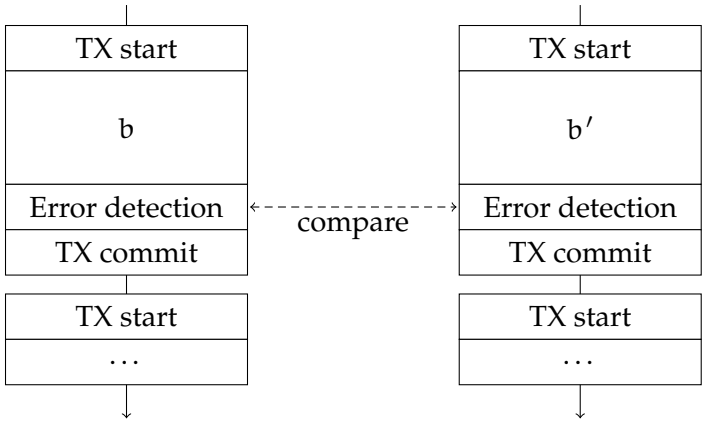


Figure 6.1: Parallel execution of redundant transactions

performance and the code coverage. The outcome of the investigations in this thesis further spawned research in custom hardware transactional memory designs for an embedded multi-core processor.

6.2.1 Hardware Transactional Memory in Embedded Multi-Core Systems

Based on the investigations in this thesis, a research project was started to design a custom hardware transactional memory for a multi-core processor that targets the embedded systems domain. The objectives of this project, which is funded by the German Research Foundation (DFG), are to investigate the usage of hardware transactional memory in embedded systems, considering the safety and real-time demands of this domain. A custom hardware transactional memory for an embedded multi-core processor allows to integrate all the necessary functionality that otherwise has to be provided by software, as in the approach described in this thesis. This includes a mechanism for the automatic wrapping with transactions, an implicit signature generation of the redundant transactions, and the repeated comparison to detect errors. The resulting system allows a redundant execution of processes that are continuously wrapped with transactions, managed by the hardware. The capability of the transactional memory to pause transactions, and to allow transactions with overflowing read- and write-sets, enable a parallel execution of the redundant transactions, as depicted in Figure 6.1. An investigation of the methods for transactional wrapping and error detection, especially for an efficient execution on an heterogeneous multi-core processor, are part of this project. Further research concerns the real-time requirements of embedded systems, and their compatibility with transactional memory.

6.2.2 Architecture Support in COTS Multi-Core Processors for Fault-tolerant Execution of Arbitrary Programs

The hardware implementation of some of the functionality for error detection and recovery can increase the code coverage of the redundantly executed program, and the performance overhead can be further reduced. Like hardware transactional memory, which is accessible through the instruction set architecture, additional parts of the required functionality can be implemented in the micro-architecture of a COTS x86 processor. In contrary to the research project that aims at a customized hardware transactional memory implementation for fault tolerance, individual extensions to an x86 processor could provide the necessary features for an efficient fault-tolerant execution. This can include the proposed hardware extensions of Section 4.5, which are a dedicated FIFO queue, an implicit signature generation, and robust transactions with support for escape actions. An architecture support for fault-tolerant execution could enable a hybrid method as proposed in this thesis, but without the required program instrumentation. The management of the redundant execution can for example be integrated into the operating system, or handled by a user-space program.

The application domain of COTS processors is continuously broadening, since the demand for performance is increasing, as well as the cost pressure. However, safety requirements still need to be satisfied, for example in recognition and control systems in self-driving cars. A high-performance COTS processor that offers the fundamental elements for a redundant and fault-tolerant execution of individual applications will be useful in this and other domains, where a tightly-coupled lockstep processor is too costly or even unfeasible for the requirement performance.

6.3 Conclusion

The presented methods in this thesis show that a fault-tolerant execution is feasible on COTS processors, and that TSX, the hardware transactional memory implementation of Intel, can be leveraged to implicitly provide checkpoints for error recovery. Additional mechanisms are required to enable the redundant execution of multi-threaded applications, of which the evaluation showed only little impact on the performance and speedup of the benchmarks. However, further hardware support would increase the applicability and the performance, for example to overcome the best-effort implementation of TSX for optimistic concurrency control, which would allow the parallel execution of redundant dependable blocks. To simplify the deployment of COTS processors in systems with fault-tolerance requirements, future processors could include additional instruction set extensions that provide hardware support for the error detection mechanisms.

Bibliography

[Ber+05] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. "NonStop® Advanced Architecture". In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 2005, pp. 12–21. DOI: 10.1109/DSN.2005.70.

[Bie+08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008, pp. 72–81.

[Bie11] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, 2011.

[Bla69] Black, James R. "Electromigration – A Brief Survey and Some Recent Results". In: *IEEE Transactions on Electron Devices* 16.4 (1969), pp. 338–347.

[Cai+13] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. "Robust Architectural Support for Transactional Memory in the Power Architecture". In: *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 2013, pp. 225–236. DOI: 10.1145/2485922.2485942.

[Cas+93] Guy Castagnoli, Stefan Brauer, and Martin Herrmann. "Optimization of Cyclic Redundancy-check Codes with 24 and 32 Parity Bits". In: *IEEE Transactions on Communications* 41.6 (1993), pp. 883–892.

[Chr+10] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. "Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack". In: *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*. 2010, pp. 27–40. DOI: 10.1145/1755913.1755918.

[Dic+09] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. "Early Experience with a Commercial Hardware Transactional Memory Implementation". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2009, pp. 157–168. DOI: 10.1145/1508244.1508263.

[Döb+14] Björn Döbel and Hermann Härtig. "Can We Put Concurrency Back into Redundant Multithreading?" In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*. 2014, pp. 1–10. DOI: 10.1145/2656045.2656050.

[Fet+11] Christof Fetzer and Pascal Felber. "Transactional Memory for Dependable Embedded Systems". In: *Proceedings of the International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2011, pp. 223–227. DOI: 10.1109/DSNW.2011.5958817.

[Haa+17] 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: *Proceedings of the International Conference on Architecture of Computing Systems (ARCS)*. 2017, pp. 16–30. DOI: 10.1007/978-3-319-54999-6_2.

[Ham+14] Per Hammarlund, Alberto J Martinez, Atiq A Bajwa, David L Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. "Haswell: The Fourth-Generation Intel Core Processor". In: *IEEE Micro* 34.2 (2014), pp. 6–20. DOI: 10.1109/MM.2014.10.

[Har+10] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. 2nd. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010. ISBN: 978-1608452354.

[Hen+17] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. Morgan Kaufmann Publishers Inc., 2017. ISBN: 978-0128119051.

[Her+08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. ISBN: 978-0123705914.

[Her+93] Maurice Herlihy and J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures". In: *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 1993, pp. 289–300.

[Int13a] Intel Corporation. *Intel 64 and IA-42 Architectures Optimization Reference Manual*. 2013. Chap. 12: Intel TSX Recommendations.

[Int13b] Intel Corporation. *Intel 64 and IA-42 Architectures Optimization Reference Manual*. 2013. Chap. 10: SSE4.2 and SIMD Programming for Text-Processing/Lexing/Parsing.

[Int13c] Intel Corporation. *Intel 64 and IA-42 Architectures Software Developer's Manual*. 2013. Chap. 3: Instruction Set Reference, A-M.

[Int13d] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*. 2013. Chap. 8: Intel Transactional Synchronization Extensions.

[Kor+07] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., 2007. ISBN: 978-0120885251.

[Kuv+16] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "HAFT: Hardware-assisted Fault Tolerance". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. 2016, 25:1–25:17. DOI: 10.1145/2901318.2901339.

[LaF+07] Christopher LaFrieda, Engin Ipek, Jose F. Martinez, and Rajit Manohar. "Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor". In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 2007, pp. 317–326. DOI: 10.1109/DSN.2007.100.

[Lat+04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 2004, p. 75.

[Lau+10] Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. "PEBIL: Efficient Static Binary Instrumentation for Linux". In: *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2010, pp. 175–183.

[Lom77] David Bruce Lomet. "Process Structuring, Synchronization, and Recovery using Atomic Actions". In: *ACM SIGOPS Operating Systems Review* 11.2 (1977), pp. 128–137.

[Lu+18] H.J. Lu, Milind Girkar, Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface*. 2018. URL: https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf (visited on 03/09/2019).

[May+79] Timothy C. May and Murray H. Woods. "Alpha-Particle-induced Soft Errors in Dynamic Memories". In: *IEEE Transactions on Electron Devices* 26.1 (1979), pp. 2–9. DOI: 10.1109/T-ED.1979.19370.

[Met+13] Stefan Metzlaff, Sebastian Weis, and Theo Ungerer. "Towards Transactional Memory for Safety-Critical Embedded Systems". In: *Euro-TM Workshop on Transactional Memory (WTM)*. 2013.

[Mon+17] Diego Montezanti, A. De Giusti, M. Naiouf, Jorge Villamayor, Dolores Rexachs, and Emilio Luque. "A Methodology for Soft Errors Detection and Automatic Recovery". In: *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*. 2017, pp. 434–441.

[Muk08] Shubu Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., 2008. ISBN: 978-0123695291.

[Nor96] Eugene Normand. "Single Event Upset at Ground Level". In: *IEEE Transactions on Nuclear Science* 43.6 (1996), pp. 2742–2750. DOI: 10.1109/23.556861.

[NVI19] NVIDIA Corporation. *F18 Fortran compiler*. 2019. URL: https://github.com/flang-compiler/f18 (visited on 05/03/2019).

[Oh+02] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. "Error Detection by Duplicated Instructions in Super-scalar Processors". In: *IEEE Transactions on Reliability* 51.1 (2002), pp. 63–75.

[Pan+18] Reena Panda, Shuang Song, Joseph Dean, and Lizy K. John. "Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?" In: *International Symposium on High Performance Computer Architecture (HPCA)*. 2018, pp. 271–282. DOI: 10.1109/HPCA.2018.00032.

[Pap+17] Dimitra Papagiannopoulou, Andrea Marongiu, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar. "Edge-TM: Exploiting Transactional Memory for Error Tolerance and Energy Efficiency". In: *Transactions on Embedded Computing Systems (TECS)* 16.5s (2017), 153:1–153:18. DOI: 10.1145/3126556.

[Rei+00] Steven K. Reinhardt and Shubhendu S. Mukherjee. "Transient Fault Detection via Simultaneous Multithreading". In: *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 2000, pp. 25–36.

[Rei+05] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. "SWIFT: Software Implemented Fault Tolerance". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 2005, pp. 243–254.

[Rot99] Eric Rotenberg. "AR-SMT: a Microarchitectural Approach to Fault Tolerance in Microprocessors". In: *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*. 1999, pp. 84–91.

[Sán+14] Daniel Sánchez, Juan M. Cebrián, José M. García, and Juan L. Aragón. "Soft-error Mitigation by Means of Decoupled Transactional Memory Threads". English. In: *Distributed Computing* (2014), pp. 1–16. DOI: 10.1007/s00446-014-0215-6.

[Sca19] Steve Scalpone. *Flang Fortran front-end for LLVM*. 2019. URL: https://github.com/flang-compiler/flang (visited on 05/03/2019).

[Sei+12] Norbert Seifert, Balkaran Gill, Shah Jahinuzzaman, Joseph Basile, Vinod Ambrose, Quan Shi, Randy Allmon, and Arkady Bramnik. "Soft Error Susceptibilities of 22 nm Tri-Gate Devices". In: *IEEE Transactions on Nuclear Science* 59.6 (2012), pp. 2666–2673. DOI: 10.1109/TNS.2012.2218128.

[Shi+02] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic". In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 2002, pp. 389–398. DOI: 10.1109/DSN.2002.1028924.

[Shy+09] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A. Connors. "PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures". In: *Transactions on Dependable and Secure Computing (TDSC)* 6.2 (2009), pp. 135–148. DOI: 10.1109/TDSC.2008.62.

[Sor09] Daniel J. Sorin. *Fault Tolerant Computer Architecture*. Morgan and Claypool Publishers Inc., 2009. ISBN: 978-1598299533.

[Sta17] Standard Performance Evaluation Corporation. *SPEC CPU® 2017 Benchmark Package*. 2017. URL: https://www.spec.org/cpu2017/ (visited on 04/23/2019).

[Ste+08] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 2008.

[Uem+16] Taiki Uemura, Soonyoung Lee, Jongwoo Park, Ssangwoo Pae, and Haebum Lee. "Investigation of logic circuit soft error rate (SER) in 14nm FinFET technology". In: *Proceedings of the International Reliability Physics Symposium (IRPS)*. 2016, 3B-4-1 –3B-4-4. DOI: 10.1109/IRPS.2016.7574519.

[Val95] John D. Valois. "Lock-free Linked Lists Using Compare-and-swap". In: *Proceedings of the Annual Symposium on Principles of Distributed Computing (PODC)*. 1995, pp. 214–222. DOI: 10.1145/224964.224988.

[Vij+02] T.N. Vijaykumar, Irith Pomeranz, and Karl Cheng. "Transient-Fault Recovery Using Simultaneous Multithreading". In: *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 2002, pp. 87–98.

[Wan+07] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. "Compiler-Managed Software-Based Redundant Multi-threading for Transient Fault Detection". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 2007, pp. 244–258.

[Wan+12] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. "Evaluation of Blue Gene/Q Hardware Support for Transactional Memories". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012, pp. 127–136.

[Yal+10] Gulay Yalcin, Osman Unsal, Ibrahim Hur, Adrian Cristal, and Mateo Valero. "FaulTM: Fault-Tolerance Using Hardware Transactional Memory". In: *Workshop on Parallel Execution of Sequential Programs on Multi-core Architecture (PESPMA)*. 2010.

[Yal+13] Gulay Yalcin, Osman Unsal, and Adrian Cristal. "Fault Tolerance for Multi-Threaded Applications by Leveraging Hardware Transactional Memory". In: *Proceedings of the International Conference on Computing Frontiers (CF)*. 2013, 4:1–4:9.

[Yen+07] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. "LogTM-SE: Decoupling Hardware Transactional Memory from Caches". In: *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 2007, pp. 261–272. DOI: 10.1109/HPCA.2007.346204.

[Yoo+13] 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 for High Performance Computing, Networking, Storage and Analysis (SC)*. 2013. DOI: 10.1145/2503210.2503232.

[Zie+79] J. F. Ziegler and W. A. Lanford. "Effect of Cosmic Rays on Computer Memories". In: *Science* 206.4420 (1979), pp. 776–788. DOI: 10.1126/science.206.4420.776.

List of Figures

List of Tables

List of Code Listings

List of Algorithms