Thesis/Dissertation Sheet

Surname/Family Name	:	**Shen**
Given Name/s	:	**Yanyan**
Abbreviation for degree as give in the University calendar	:	**PhD**
Faculty	:	**Faculty of Engineering**
School	:	**School of Computer Science and Engineering**
Thesis Title	:	**Microkernel Mechanisms for Improving the Trustworthiness of Commodity Hardware**

Abstract 350 words maximum: (PLEASE TYPE)

The thesis presents microkernel-based software-implemented mechanisms for improving the trustworthiness of computer systems based on commercial off-the-shelf (COTS) hardware that can malfunction when the hardware is impacted by transient hardware faults. The hardware anomalies, if undetected, can cause data corruptions, system crashes, and security vulnerabilities, significantly undermining system dependability. Specifically, we adopt the single event upset (SEU) fault model and address transient CPU or memory faults.

We take advantage of the functional correctness and isolation guarantee provided by the formally verified seL4 microkernel and hardware redundancy provided by multicore processors, design the redundant co-execution (RCoE) architecture that replicates a whole software system (including the microkernel) onto different CPU cores, and implement two variants, loosely-coupled redundant co-execution (LC-RCoE) and closely-coupled redundant co-execution (CC-RCoE), for the ARM and x86 architectures. RCoE treats each replica of the software system as a state machine and ensures that the replicas start from the same initial state, observe consistent inputs, perform equivalent state transitions, and thus produce consistent outputs during error-free executions. Compared with other software-based error detection approaches, the distinguishing feature of RCoE is that the microkernel and device drivers are also included in redundant co-execution, significantly extending the sphere of replication (SoR).

Based on RCoE, we introduce two kernel mechanisms, fingerprint validation and kernel barrier timeout, detecting fault-induced execution divergences between the replicated systems, with the flexibility of tuning the error detection latency and coverage. The kernel error-masking mechanisms built on RCoE enable downgrading from triple modular redundancy (TMR) to dual modular redundancy (DMR) without service interruption. We run synthetic benchmarks and system benchmarks to evaluate the performance overhead of the approach, observe that the overhead varies based on the characteristics of workloads and the variants (LC-RCoE or CC-RCoE), and conclude that the approach is applicable for real-world applications. The effectiveness of the error detection mechanisms is assessed by conducting fault injection campaigns on real hardware, and the results demonstrate compelling improvement.

Declaration relating to disposition of project thesis/dissertation

…………………………………………………………… ………………………………………..……………… ………..……………………...………

Signature Witness Signature Date

FOR OFFICE USE ONLY Date of completion of requirements for Award:

Microkernel Mechanisms for Improving the Trustworthiness of Commodity Hardware

Yanyan Shen

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computer Science and Engineering

Faculty of Engineering

March 2019

ORIGINALITY STATEMENT

‘I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.’

Signed ……………………………………………………..............

Date ……………………………………………………..............

COPYRIGHT STATEMENT

Signed ……………………………………………..........................

Date ……………………………………………..........................

AUTHENTICITY STATEMENT

Signed ……………………………………………..........................

Date ……………………………………………..........................

Publications

The thesis is partially based on the following publications:

- Yanyan Shen, Gernot Heiser, and Kevin Elphinstone. Fault tolerance through redundant execution on COTS multicores: Performance, functionality, and dependability trade-offs. In *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN)*, June 2019

- Yanyan Shen and Kevin Elphinstone. Microkernel mechanisms for improving the trustworthiness of commodity hardware. In *European Dependable Computing Conference*, page 12, Paris, France, September 2015

- Kevin Elphinstone and Yanyan Shen. Improving the trustworthiness of commodity hardware with software. In *Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology (DCDV)*, page 6, Budapest, Hungary, June 2013

Abstract

The thesis presents microkernel-based software-implemented mechanisms for improving the trustworthiness of computer systems based on commercial off-the-shelf (COTS) hardware that can malfunction when the hardware is impacted by transient hardware faults. The hardware anomalies, if undetected, can cause data corruptions, system crashes, and security vulnerabilities, significantly undermining system dependability. Specifically, we adopt the single event upset (SEU) fault model and address transient CPU or memory faults.

We take advantage of the functional correctness and isolation guarantee provided by the formally verified seL4 microkernel and hardware redundancy provided by multicore processors, design the redundant co-execution (RCoE) architecture that replicates a whole software system (including the microkernel) onto different CPU cores, and implement two variants, loosely-coupled redundant co-execution (LC-RCoE) and closely-coupled redundant co-execution (CC-RCoE), for the ARM and x86 architectures. RCoE treats each replica of the software system as a state machine and ensures that the replicas start from the same initial state, observe consistent inputs, perform equivalent state transitions, and thus produce consistent outputs during error-free executions. Compared with other software-based error detection approaches, the distinguishing feature of RCoE is that the microkernel and device drivers are also included in redundant co-execution, significantly extending the sphere of replication (SoR).

Based on RCoE, we introduce two kernel mechanisms, fingerprint validation and kernel barrier timeout, detecting fault-induced execution divergences between the replicated systems, with the flexibility of tuning the error detection latency and coverage. The kernel error-masking mechanisms built on RCoE enable downgrading from triple modular redundancy (TMR) to dual modular redundancy (DMR) without service interruption. We run synthetic benchmarks and system benchmarks to evaluate the performance overhead of the approach, observe that the overhead varies based on the characteristics of workloads and the variants (LC-RCoE or CC-RCoE), and conclude that the approach is applicable for real-world applications. The effectiveness of the error detection mechanisms is assessed by conducting fault injection campaigns on real hardware, and the results demonstrate compelling improvement.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

> In computing, the mean time to failure keeps getting shorter.
>
> Alan J. Perlis

1.1 Motivation

Computer-controlled safety-critical and security-critical systems require rigorous design, implementation, and evaluation to guarantee functional correctness and to provide a high level of assurance. The vast adoption of computers in such systems is a double-edged sword: on the one side, the computation power and storage provided by computers ensure responsive, autonomous, and accurate operations of the systems, accomplishing tasks that otherwise cannot be achieved; on the other side, computer failures, caused by hardware errors or software bugs, can render the systems malfunctioning, potentially causing catastrophic results.

Conventionally, software bugs are deemed to be inevitable in complex systems. One strategy that system designers may choose to defend against the negative effect of software bugs is to partition a complex system into trusted and untrusted components. The trusted components, such as a flight-control program in an unmanned aerial vehicle (UAV), can be kept small and straightforward so that formal verification can be applied to eliminate software bugs. The untrusted components, for instance, image-capturing and -processing software in the UAV, can be checked by traditional testing-based approaches which have limited correctness guarantee; thus, these untrusted components may contain bugs that can be exploited by attackers to take control of the trusted components indirectly. Hence, trustworthy isolation between trusted and untrusted components is crucial, so the malfunctioning or even compromised untrusted components are unable to affect the trusted components. The lack of trust in software-enforced isolation leads to architectures (e.g., an air gap) using redundant or separated hardware for different components to achieve the isolation. However, the issue of increased size, weight, and power consumption caused by employing redundant hardware cannot be tolerated in certain scenarios, such as in UAVs or military vehicles, even if the increased cost is not a concern.

Various software-based approaches for consolidating software components with different security levels have been proposed. *Separation kernels* create “an environment which is indistinguishable from that provided by a physically distributed system", confine software components in their regimes, and only allow information to flow through explicitly provided channels [Rushby, 1981]. Based on a separation kernel, the *multiple independent levels of security* (MILS) architecture consolidates partitioned systems onto a single hardware platform while guaranteeing that security policies are "non-bypassable, evaluatable, always-invoked, and tamperproof" [Alves-Foss et al., 2006]. Virtualization technology also provides platforms for system consolidation at the operating system level [Garfinkel et al., 2003; Meushaw and Simard, 2000; Popek and Goldberg, 1974]. All these approaches share a common property: the lowest level system software, be it a separation kernel or a virtual machine monitor (VMM), must function correctly according to specifications. This property implies a bug-free kernel or VMM that can be trusted to uphold safety and security policies.

Recent advances in formal verification and microkernel design make a bug-free microkernel possible [Klein et al., 2009]; the seL4 microkernel can be trusted to follow the design specifications. Additional proofs demonstrate that the microkernel can provide strong guarantees concerning data integrity and isolation of applications [Elkaduwe et al., 2006; Murray et al., 2013; Sewell et al., 2011]. Moreover, binary verification connects the proofs with the microkernel binary directly, thus removing compilers from the trusted tool-chain [Sewell et al., 2013]. Based on the verified kernel, we can construct a reliable platform or framework for integrating trusted and untrusted components on a single hardware platform without compromising performance [Fisher, 2014]; however, barriers to wide adoption of software-enforced isolation in safety- or security-critical situations still exist since the microkernel assumes that the hardware also functions correctly according to its specifications. Any deviation in hardware behaviour, even a single-bit flip, will invalidate the assumptions and thus undermine the security and functional properties derived from formal verification. For example, from a very high-level perspective, an application assumes that its private data stored in memory does not change if the application does not modify it explicitly; but the assumption can be invalidated by a single-bit flip in the memory region containing the data.

Another motivation for our work is to replace the expensive, power-hungry, and slow hardware components (e.g., radiation-harden processors) used in fault-tolerant systems with COTS hardware. We briefly compare the Cortex-A9 processor with the widely-used radiation-hardened RAD750 processor [Berger et al., 2011] to estimate the potential advantages of using software-implemented fault-tolerant systems based on COTS for short-term missions. The RAD750 processor running at 133 MHz achieves around 240 Dhrystone 2.1 MIPS and requires less than 6 watts of power [Berger et al., 2001]. Each Cortex-A9 core can achieve 2.50 DMIPS per MHz [ARM, 2009]. Thus, we can get around 2,000 DMIPS from the quad-core Cortex-A9 processor running at 800 MHz if we use three cores for triple modular redundancy (TMR) and leave the remaining core idle. Even if we assume the performance overhead of TMR is 50%, still, the system can achieve 1000 DMIPS with a total power consumption of 5 watts for the whole SABRE Lite board [Boundary Devices, 2011], which uses a quad-core Cortex-A9 processor. Each RAD750 processor costs around US$ 200,000 [Ginosar, 2012], but the price tag for a SABRE Lite board is around US$ 200 [Boundary Devices]. Admittedly, the RAD750 processors can tolerate a more stringent operating environment in terms of temperature ranges and radiation intensities. However, software-based approaches have the benefits of avoiding hardware vendor lock-in and adopting new technology quickly with significantly improved performance and reduced costs.

COTS (commercial-off-the-shelf) hardware refers to the general-purpose systems or devices that are manufactured in large quantity and ready to purchase from various vendors or distributors. The main advantages of COTS hardware over custom-made, in-house, or commissioned counterparts are substantial cost savings, reduced development time, improved performance, and current technology node. In the areas previously dominated by customised hardware components, COTS-based systems are gaining acceptance and applications [Engel, 2001; Esposito et al., 2015; Gansler and Lucyshyn, 2008; Hiergeist and Holzapfel, 2016]. Nevertheless, COTS hardware faults can have direct impacts on system reliability. Various field studies showed the significance of the COTS hardware reliability issue [Li et al., 2007; Nightingale et al., 2011; Schroeder et al., 2009; Shazli et al., 2008; Sridharan et al., 2013, 2015]. Hardware faults can be categorised as *permanent faults*, *transient faults*, and *intermittent faults* [Koren and Krishna, 2007]. A permanent fault completely damages a hardware component, and a replacement is required to restore functioning. An intermittent fault comes and goes repeatedly but never disappears entirely, causing a component to swing between correct and incorrect operating modes. A transient fault temporally deviates a hardware component from its specification; however, the component is not permanently damaged.

Transient faults, specifically single-event-effect-induced transient faults, are the threats we aim to deal with in the thesis. Correctly designed and manufactured hardware can be affected by environmental factors, such as power fluctuation, temperature variations, and radiation. A significant fraction of transient faults originates from single event effects (SEEs) which are triggered by alpha particle strikes or cosmic-ray-induced neutron showers [Baumann, 2005a; Michalak et al., 2005; Saggese et al., 2005] (details in Section 2.1). Recently studies show that seemingly benign transient faults can introduce security vulnerabilities into Linux kernel, network services, and virtual machines; some of the vulnerabilities can be removed only by a reboot, leaving the systems vulnerable for an extended period [Chen et al., 2004; Govindavajhala and Appel, 2003; Xu et al., 2001]. Furthermore, commodity DRAM is susceptible to disturbance errors [Kim et al., 2014] so that malicious applications can corrupt data stored in a row by repeatedly reading nearby rows and thus create an exploit, as demonstrated by the *row hammer* attack [Rowhammer]. Section 2.1 will give a thorough discussion about transient faults; related existing software and hardware solutions aiming to detect and tolerate hardware faults will be introduced in Chapter 3.

The thesis adopts single event upset (SEU) (Section 4.1.4) as the fault model, which implies faults are random and independent, other than malicious and coordinated. The Byzantine fault model [Lamport et al., 1982] describes a much stronger model in which faulty participants of a distributed system can produce arbitrary results or even cooperate intentionally to prevent non-faulty participants from reaching a consensus. Tolerating Byzantine faults usually requires more replication effort ($3 \times n + 1$ replicas in which n is the number of faulty replicas) and performance overhead because of the multi-stage voting scheme. Essentially, the safety analysis of the SEU model assumes multiple faults to be independent, while the Byzantine model assumes collusion. Therefore, tolerating Byzantine faults is out the scope of the thesis.

1.2 Contributions

The thesis presents a microkernel-based software solution addressing the COTS hardware reliability issue and reducing the window of vulnerability induced by transient hardware faults. Specifically, we apply redundant co-execution and the concept of sphere of replication (SoR) on a modern microkernel by instantiating multiple replicas of the kernel and applications. Furthermore, each replica is deployed on a core of a multicore processor to exploit the spatial redundancy provided by modern hardware; and thus a DMR (dual modular redundancy) or TMR (triple modular redundancy) system can be structured by running two or three replicas. We manage the replicas as replicated state machines (RSM), start them in the same state, and feed them with same inputs; so the replicas should proceed with consistent state transitions and produce comparable output data. Otherwise, if the state machines diverge or produce inconsistent output data, we consider them faulty; and we fail-stop the system or initiate recovery procedure if certain conditions are satisfied. Targeting COTS hardware, our error detection mechanisms cannot rely on special hardware features, nor do they assume a small piece of hardware is always reliable. However, our solution benefits from various improvements of fast-evolving COTS hardware from generation to generation, compensating for the performance overheads of redundant co-execution and error detection mechanisms. Designing and building for COTS hardware significantly lowers the barriers to adopting our solution when system designers are planning a new system or porting an existing one for improved security and integrity. We make the following contributions in the thesis:

- We identify the risks of building highly secure systems using commodity hardware since the states of processors or memory can be corrupted by transient hardware faults; undetected bit flips can be benign at best because of various masking effects or catastrophic if security- or safety-critical data is corrupted or control flow is altered.

- We design and implement a set of microkernel mechanisms to support whole-system redundant co-execution on multicore processors. We apply redundant co-execution to the lowest-level system software (the seL4 microkernel) as well as user-mode applications, and redundantly validate the status of the replicas for error detection. Essentially, the modified microkernel can conduct self-checking without relying on lower-layer software or specialised hardware (i.e., we do not shift the problem to another layer beneath the kernel or hardware).

- We identify and treat sources of non-determinism in microkernel-based systems. The non-deterministic events can lead to the divergence of the replicas even when the system is not affected by faults, introducing false positives.

- We design and apply the I/O access patterns to replicated device drivers so that the redundantly executed drivers can interact with non-redundant I/O devices correctly. The drivers replicate input data so that other components of the replicated system can observe consistent inputs; the drivers also check output data for error detection.

- We investigate and analyse the performance overhead of the resulting system on x86 and ARM platforms, demonstrating that the whole-system redundant co-execution and error detection mechanisms result in moderate performance degradation and are applicable to real-world systems.

- We build a fault injection framework and conduct fault injection campaigns on real x86 and ARM machines, showing that our prototype is effective in detecting errors and preserving data integrity.

In summary, an seL4-based software system protected by the whole-system redundant co-execution possesses significantly improved capability of defending against errors induced by transient hardware faults, and thus the guarantee of preserving security properties provided by the formally verified microkernel can be better protected when the system is deployed on COTS hardware. In this thesis, we focus on the case that a software system (including the microkernel, device drivers, and applications) runs on a single core; replicating a multicore system is left as future work and discussed in Section 8.2.1. Although the work is based on the seL4 microkernel, we believe that the obstacles we identified and solved when applying state machine replication to a whole software system, the set of kernel mechanisms for supporting redundant co-execution, and the error detection and masking methods are potentially applicable to other microkernels. The microkernel mechanisms introduced in the thesis have the potential of being used to build fault-tolerant systems to address dependability issues in a broader context beyond just a self-checking microkernel. Note that the new code added in our prototype is not formally verified; the verification of the changes is out the scope of the thesis. Additionally, our approach prevents vulnerabilities introduced by transient hardware faults from being created or being dormant too long in a system, eliminating or shortening the window of opportunity for attacking the system by exploiting the vulnerabilities, as in the attack-vulnerability-intrusion (AVI) fault model [Veríssimo et al., 2006]. But we do not claim that the approach can directly improve system security; therefore, evaluating the resulting systems in terms of security is considered as future work.

1.3 Thesis Organisation

Chapter 2 introduces why correctly designed and manufactured COTS hardware fails and provides necessary background about the formally verified seL4 microkernel. Chapter 3 describes related hardware or software approaches to detecting and tolerating hardware faults. Chapter 4 focuses on discussing microkernel mechanisms to replicate a whole software system onto different cores, to co-execute the replicas of the system redundantly, and to tolerate various non-deterministic events that can cause the replicas to diverge. Two redundant co-execution modes, *loosely-coupled redundant co-execution* (LC-RCoE) and *closely-coupled redundant co-execution* (CC-RCoE), are presented; and the applicability and restrictions of the modes are examined. Chapter 5 is dedicated to describing the microkernel mechanisms for supporting device driver replication. The device drivers serve as input data duplicators and output data comparators, ensuring consistent observations of I/O inputs by the replicas and checking outputs produced by the replicas. Chapter 6 discusses error-detection and error-masking mechanisms based on redundant co-execution (RCoE) approach, illustrating that these mechanisms are flexible enough so that system designers are able to experiment with various configurations that possess different error coverages, error detection latencies, and runtime overheads. In Chapter 7, we use microbenchmarks and system benchmarks to evaluate the performance of the resulting systems and to demonstrate the effectiveness of our error detection framework with fault injection campaigns. Finally, we conclude the thesis and propose future research directions in Chapter 8.

Chapter 2

Background

> The cheapest, fastest and most reliable components of a computer system are those that aren't there.
>
> Gordon Bell

This chapter lays the foundation for the following chapters, introducing why correctly designed and rigorously tested hardware can still fail, revealing the consequences of transient hardware faults, describing related concepts in fault-tolerant computing, and presenting the verified seL4 microkernel with sufficient details to facilitate understanding the error detection mechanisms and design decisions presented in the thesis. The precise meanings of various terms used in the thesis are listed in Appendix A for reference.

2.1 Transient Faults in CMOS Circuits

We first describe the physical phenomena behind single event effects (SEEs) and how the SEEs induce single event upsets (SEUs) in digital circuits and cause hardware components to deviate from their design specifications. This section serves as a gentle introduction that is sufficient for readers to understand the problem, not a complete review. CMOS (complementary metal oxide semiconductor) technology is the fundamental building block for most of the digital components used in general-purpose processors, micro-controllers, graphic processing units, DRAM (dynamic random access memory), SRAM (static random access memory), etc. Two essential characteristics of CMOS, large noise margin and low static power consumption, make CMOS the dominating technology for constructing ICs (integrated circuits). As the name suggested, CMOS implements logic functions by using complementary pairs of n-type and p-type MOSFETs (metal-oxide-semiconductor field-effect transistors). We will briefly describe how MOSFETs work, and then introduce how alpha particles and neutrons can cause transient hardware faults.

2.1.1 MOSFET and Basic Physical Mechanisms

Figure 2.1 illustrates the simplified structure of an n-type MOSFET which relies on electrons as the carriers. The *source* and *drain* are n-doped terminals, and the *bulk* (body) is p-typed. The "metal" in the name refers to the *gate* material, but nowadays the gate is usually a polysilicon layer. Similarly, the isolation layer that separates the gate from the bulk can be dielectric materials other than silicon dioxide. When no voltage is applied to the gate, there is no channel between the source and drain terminals; so the transistor is *OFF*. When a positive voltage is applied to the gate, holes are pushed downward into the body. In the meanwhile, electrons are drawn towards the gate area by the positive voltage. If the applied voltage is greater than the threshold voltage of the MOSFET, the electrons amassed under the insulation layer are sufficient to create a conductive channel between the drain and source, turning the transistor *ON*. P-type MOSFETs work in a similar way, except that the source and drain are p-type material, the bulk is n-type material, and the carriers forming the conductive channel are holes.

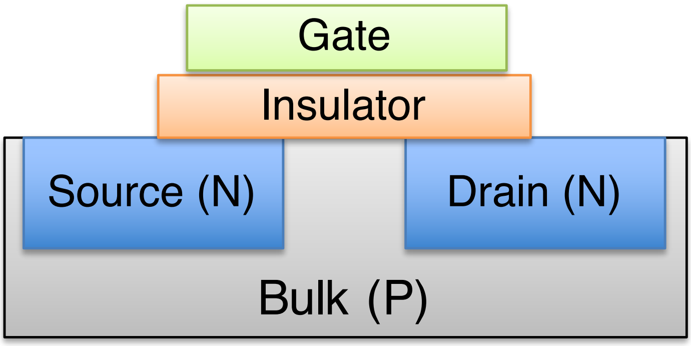

Figure 2.1: N-type MOSFET

Alpha particle and neutron strikes are two dominating causes that can cause transient hardware faults, although other factors (temperature variations, wire crosstalk, voltage fluctuations, etc.) may also contribute. Alpha particles, emitted from radioactive matters of impure packaging materials, consist of two protons and two neutrons. Charged alpha particles interact directly with silicon crystals, creating tracks of electron-hole pairs [May and Woods, 1979]. Purifying packaging materials can reduce alpha particle emissions, but it is difficult to eradicate alpha particles completely. Less susceptible circuit designs and shielding chips with thick polyimide layers can also improve the reliability of the chips significantly, yet the chips are still not free from alpha-particle-induced faults. High-energy neutrons, produced by primary cosmic rays colliding with atoms and molecules in the Earth's atmosphere, are ubiquitous around the world. Unlike alpha particles, neutrons do not produce electron-hole pairs directly. Instead, neutrons collide with nuclei in transistors and create secondary particles consisting of protons, neutrons, alpha particles, etc. Then, these secondary particles can produce ionization tracks that create electron-hole pairs [Ziegler and Lanford, 1979]. The density of neutron flux is highly correlated to the altitude and the location on the Earth. The JEDEC standard JESD89A [JEDEC, 2016] defines how neutron flux is measured and lists neutron flux at selected cities or locations relative to the reference flux of New York City.

Single event effects (SEEs) are caused by high energy particles striking sensitive regions of CMOS-based devices. A single event upset (SEU) is a non-destructive change of state in a storage element, and it may affect a single bit or multiple bits. The electron-hole pairs inside or near the depletion region of the p-n junction are prevented from recombination by the electric field that also collects the carriers efficiently in a short time (*drift collection*), resulting in a transient current at the impacted node. The following phase is called *diffusion collection* during which more charge is collected as excess carriers diffuse toward the depletion region [Baumann, 2005a; Dodd and Massengill, 2003]. If a sufficient amount of charge is collected, exceeding the *critical charge* (Q_{crit}), a single-event upset (SEU) occurs. The Q_{crit} is the minimal amount of charge that must be collected to cause an upset.

2.1.2 Random Access Memory (RAM)

We briefly introduce how a DRAM cell works in Section C.1. If the storage capacitor or the source of the access transistor of a DRAM cell is affected by a particle strike, the charge stored in the capacitor may be disturbed by the event so that the stored bit is flipped [May and Woods, 1979]. Further, a single particle strike is able to trigger multiple upsets (a multi-cell upset) if the ion track penetrates two proximal junctions or the charge is collected by multiple sensitive nodes [Massengill, 1996]. If the error bits are in one data word (a multi-bit upset), the SEC-DED ECC is unable to correct the errors. During a read operation, the differential voltage of the two bit lines can be disturbed if transistors directly connected to the bit lines experience a particle strike, causing the sense amplifier fail to interpret the stored bit correctly [Gulati et al., 1994]. DRAM needs various supporting logic circuits to decode addresses (decoders), to buffer the read results (latches), to pre-charge bit lines (pre-chargers), etc. Logic upsets in these components can also lead to erroneous data [Slayman, 2011]. Moreover, as the manufacturing process moves to smaller feature sizes, the closely packed cells can interfere each others. As shown by Kim et al. [2014], frequently activating and deactivating rows can cause some of adjacent cells to leak charges.

Static random access memory (SRAM) is widely used as various levels of caches in processors to fill the gap between very fast processors and slow DRAM. SRAM does not use capacitors for keeping data; a bit is stored in a pair of cross-coupled inverters [Pavlov and Sanchdev, 2008]. Thus, SRAM also suffers from transient faults [Cannon et al., 2004; Seifert et al., 2006]. Section C.2 describes the basic mechanism of an SRAM cell and how a transient fault changes the value stored in the cell.

2.1.3 Logic Circuits

Logic circuits are not immune to transient faults since the underlying physical phenomena are the same. The mechanism of particles striking latches, register files, flip-flops, etc. is similar to the one impacting SRAM [Karnik et al., 2004]. A transient voltage caused by a particle strike at a sensitive node of a combinational logic component is called a single event transient (SET), and SETs are becoming an urgent issue for complex circuits [Dodd et al., 2004; Ferlet-Cavrois et al., 2013]. If an SET propagates through logic gates and causes an incorrect value being captured by a storage element, the SET then leads to an SEU. Logic components benefit from the following masking effects [Shivakumar et al., 2002]:

- Electrical masking: A transient pulse is attenuated by subsequent logic gates so that it does not affect the result of a digital component. This masking effect is influenced by the electrical properties of the gates of the component.

- Logical masking: A transient fault is prevented from propagating from inputs of a component to its outputs because the outputs are completely determined by the input values that are not affected by the fault. Take an *AND* gate for example; if a zero is one of the two inputs, the output stays at zero no matter what the other input value is.
- Temporal masking: A transient fault reaches the output of the current latch, but the time when the fault arrives at the input of the next latch is outside the latching window of the next latch; thus, the erroneous value is not captured.

Unlike storage elements that are relatively straightforward to protect with error-correcting codes, approaches (hardening, DMR, TMR, or self-checking designs) to protecting logic components usually accompany significant overheads in terms of die areas and power consumption.

2.1.4 Trends of Transient Faults

The feature size of CMOS transistors is steadily and continuously decreasing, so is the supply voltage. An empirical model was developed to estimate the trend of SER for CMOS SRAM circuits [Hazucha and Svensson, 2000; Shivakumar et al., 2002]: $SER \propto F \times A \times \exp(-\frac{Q_{CRIT}}{Q_S})$. In the equation, F is the neutrons with energies greater than 1MeV, A is the sensitive area in cm^2, Q_{CRIT} is the critical charge in fC, and Q_S is the charge collection efficiency of the device. Q_{CRIT} mainly depends on the supply voltage V_{dd}, and the type of the (P or N) drain struck by a particle also affects Q_{CRIT}. Q_S represents the charge generated by a particle strike and is determined by the characteristics (e.g., doping and V_{dd}) of the device; Q_{CRIT} and Q_S are independent. The model states that the SER of SRAM circuits increases exponentially when Q_{CRIT} declines and decreases linearly when the sensitive area shrinks. The scaling feature size reduces the sensitive area, lowering the SER; but the accompanied reduction in supply voltage for improved power efficiency also decreases Q_{CRIT} and thus increases the SER. The single-bit SRAM per-bit SER tread from 180 nm to 65 nm nodes published by Intel peaked at 130 nm node and decreased since then [Seifert et al., 2006]. A sharp increase of the SER when scaling from 250 nm to 160 nm was depicted by Dodd et al. [2010], peaking at 140 nm node and decreasing ever since. Baumann [2005b] described and explained the saturation of the SER as the feature size scales. Slayman [2011] reported that the trend of SRAM per-bit SER from multiple vendors is almost flat from 250 nm to 50 nm design rules. Dixit and Wood [2011] observed a decline of the SRAM SER from from 250 nm to 65 nm, but the trend was reversed at the 40 nm feature size: the SER for individual 40 nm SRAM cells was greater than the SER of the 65 nm cells. They also expected that the reduction of Q_{CRIT} will eventually cause the SER of SRAM to rise. However, the general agreement is that the overall system-level SER of SRAM is increasing since the scaling feature size also translates to higher cell density and that the capacity of SRAM in microprocessors keeps growing rapidly. Furthermore, multi-cell upsets are increasing because a particle can more easily penetrate several densely packed cells [Dixit and Wood, 2011; Dixit et al., 2009], challenging the effectiveness of SEC-DED ECC.

The SER performance of DRAM cells is improving as technology advances [Baumann, 2003; Slayman, 2011]: over seven generations, the SER of DRAM has been reduced by more than 1000 times [Baumann, 2005b]. The significant reduction can be attributed to the facts that the charge collection efficiency decreases as the sensitive area shrinks and that the cell capacitance (and thus Q_{CRIT}) is relatively constant over generations. Nevertheless, the system SER of DRAM remains constant or even increases due to the greatly increased memory density and capacity. Another trend is that DRAM logic errors need more attention: the DRAM logic errors tend to be multi-bit errors and require advanced ECC (e.g., chipkill) to correct [Slayman, 2011].

Shivakumar et al. [2002] predict how technology scaling will affect the SER of combinational logic by building a model that takes masking effects into consideration. The results reveal that both memory and combinational logic elements are becoming more susceptible because of feature size scaling, and the SER of combinational logic is increasing faster than the memory's since the scaling also impacts the masking effects that reduce the SER of combinational logic. In terms of single event transients, increasing clock frequencies, along with CMOS technology scaling, has negative impact on the electrical and temporal masking: the higher frequencies lead to a higher percentage of transients that fulfil the requirements for propagation and the elevated probability of transient signals being captured by storage elements [Buchner et al., 1997; Ferlet-Cavrois et al., 2013]. The threshold frequency at which logic errors overtake flip-flop errors, based on test results on 40 nm bulk CMOS circuits, has been estimated in the range of 1.5 to 5 GHz [Mahatme et al., 2011]. The SET (single event transient) pulse width plays an important role in determining the vulnerability caused by SETs: the wider the transient pulse width, the higher the probability of the transient signal causing an error. Benedetto et al. [2006] observed significant increase in pulse width as a result of feature size scaling and lower nominal voltages. The increase of pulse width was also reported when the technology scaled from 130 nm node to 90 nm node [Narasimham et al., 2007], " The event cross section is the highest for SET pulses between 400 ps to 700 ps in the 130 nm process, while it is dominated by SET pulses in the range of 500 ps to 900 ps in the 90 nm process." However, Nakamura et al. [2012] reported that most of the pulse widths observed for 20-stage NAND AND inverter chains manufactured by 90 nm and 40 nm bulk CMOS process are less than 150 ps. The reduction of the pulse widths when moving to 65 nm node from 130 nm and 90 nm nodes is also described by Gadlage et al. [2010], and the authors concluded that the trend of transient pulse widths is difficult to determine due to the fact that the several combined factors [Ahlbin et al., 2010; Jagannathan et al., 2010; Maharrey et al., 2013] affect the SET pulse width distributions. Modeling the SETs in advanced ICs using advanced technologies, for example, FinFET (Fin Field Effect Transistor) and UTSOI (ultrathin silicon on insulator), is an active and challenging research area [Artola et al., 2015].

In summary, as the feature sizes shrink and technology advances, the volume of sensitive regions keeps shrinking so that the charge collection efficiency reduces. The move from planar bulk to SOI [Cannon et al., 2004; Oldiges et al., 2009] and multi-gate [Fang and Oates, 2011; Wang et al., 2006] technologies also reduces cross sections and sensitive volumes. However, feature size scaling is also accompanied by reductions in the nominal supply voltage (thus, the critical charge Q_{crit}), noise margins, and node capacitance. The reductions raise the sensitivity to soft errors and expand the spectrum of "problematic" particles (e.g., protons [Rodbell et al., 2007], muons [Sierawski et al., 2010], and electrons [King et al., 2013]). Also, the significantly increased device density raises the possibility of multiple components being affected by a single particle strike so that the effectiveness of SEC-DED ECC coding and other redundancy-base fault-tolerant mechanisms needs further investigation. Succinctly, despite advanced technologies improving soft error rates of ICs, the reliability issues caused by transient faults persist [Massengill et al., 2012].

2.1.5 COTS Hardware Faults in the Wild

Having introduced the physical mechanisms and the trends of transient faults, we try to get some insights about how transient faults affect the reliability of COTS hardware by examining real-world data—are the faults so rare that they can be ignored, or are they real threats that must be dealt with properly to ensure a certain level of reliability? There are a plethora of field studies collecting and analysing COTS hardware reliability data; we only excerpt the data related to transient faults.

The SEC-DED ECC-protected memory (approximately 3.92 GB each server) used by 212 servers was monitored over a period of three months, and 8288 memory errors were collected [Li et al., 2007]. The authors believed that most of the errors were permanent errors and that only two were soft errors. The error data was collected directly from memory controllers which recorded corrected errors. The authors also conducted measurement by using a software-based error collection approach on 20 desktops and 70 geographically distributed *PlanetLab* machines, and all these machines were not protected by ECC. The software approach reported no errors. We speculate that the zero-error result is because of the limited memory sizes monitored by the software method: averagely 104.23 MiB out of 512 MiB for each desktop and 1.54 MiB for each *PlanetLab* machine. Schroeder et al. [2009] gathered memory error data in a large server fleet over the period of 2.5 years. Two out of the six platforms studied employed SEC-DED ECC, and the rest platforms used more powerful Chipkill [IBM, 2016] ECC. The DIMMs came from multiple manufacturers and covered common DRAM technologies: double data rate (DDR1), double data rate 2 (DDR2), and fully-buffered DDR2 (FBDIMM). Some of the key findings include: (1) The error rates are much higher than previously reported. "Across the entire fleet, 8.2% of all DIMMs are affected by correctable errors and an average DIMM experiences nearly 4000 correctable errors per year." Even with SEC-DED ECC or Chipkill-protected memory, "the annual incidence of uncorrectable errors was 1.3% per machine and 0.22% per DIMM." (2) "Memory errors are strongly correlated. A DIMM that sees a correctable error is 13-228 times more likely to see another correctable error in the same month." (3) "Error rates are unlikely to be dominated by soft errors." From the two studies, we can observe that memory errors are common and frequent, that SEC-DED ECC or even Chipkill is unable to correct all errors, and that transient faults are rare but not negligible.

Sridharan et al. [2013] summarised DRAM and SRAM faults in supercomputers (Cielo and Jaguar). The authors made the following observations: (1) "the composition of DRAM faults shifts markedly during the first two years of lift time, changing from primarily permanent faults to primarily transient faults." (2) "a significant inter-vendor effect on DRAM fault rates, with fault rates varying by up to 4× among vendors." (3) "SRAM faults in the field are primarily transient." In a following-up study, DRAM and SRAM errors of two supercomputer clusters (Cielo and Hopper) were analysed [Sridharan et al., 2015]; the collected data comprised over 314 million CPU socket-hours and 45 billion DRAM device-hours. Cielo is equipped with Chipkill-correct ECC memory [IBM, 2016], and Hopper is protected by Chipkill-detect ECC memory. Both Chipkill ECCs are more resilient to faults than SDC-DED ECC memory: Chipkill-detect ECC is able to detect *any error* in a single memory chip, and Chipkill-correct ECC can correct *any error* in a single memory chip. The authors find that the SRAM used as CPU L1/L2 caches mainly suffers from transient faults caused by particle strikes, so extending the parity-protected SRAM to the SDC-DED ECC protected SRAM should be able to correct most of single-bit faults in SRAM. However, the level of protection provided by SDC-DED ECC memory is not sufficient for modern memory systems and may result in *undetected* errors up to 20 FIT per DRAM device. For the Hopper system, 78.9% of total DRAM faults are single-bit faults, and the rests are various multi-bit faults; transient faults represent 44.5% of total DRAM faults. It is worth noticing that the DRAM of these supercomputer systems uses advanced ECC technology and that the quality grade of the memory chips are higher than consumer-grade chips used in desktops, laptops, and mobile devices. Both studies confirm that the DRAM failure rates are vendor-dependant and that the percentages of transient faults are higher than the results of previous studies [Li et al., 2007; Schroeder et al., 2009]. The trend of increasing multi-bit faults is also supported by the data, calling for advance ECC variants to reduce uncorrectable errors and silent data corruptions.

Bautista-Gomez et al. [2016] collected raw memory errors on a supercomputer consisting of 1080 nodes from February 2015 to February 2016; each node is an ARM SoC including two cores and 4 GiB low-power DRAM without ECC protection. 923 nodes were continuously monitored by a memory scanning program which wrote and checked data patterns when a node was idle to detect bit flips. The authors intentionally chose DRAM modules without ECC protection and implemented software-based memory error monitoring to detect the errors escaping ECC and leading to silent data corruptions. In total, the study analysed the memory monitoring logs of 12,135 terabyte-hours, and 55,000 independent memory errors were logged. Among the logged errors, 85 corrupted multiple bits of a memory word: 76 were double-bit errors, and 9 errors corrupted more than 2 bits so that SEC-DED ECC would fail to catch the errors. Given the logs included the timestamps when errors were detected, an important fact revealed by the study is that "over 26,000 corruptions occurred simultaneously to other corruptions in the same node. Over 99.9% of those were multiple single-bit corruptions that occurred simultaneously in different parts of the memory of the same node." This finding demonstrates that the single-bit-error threat model, which is adopted by several fault-tolerant approaches, is not suitable for modern unprotected DRAM.

Nightingale et al. [2011] analysed hardware failure rates of over one million PCs that lack of error detection features, such as ECC protected memory. The data was collected by the Windows Error Reporting (WER) system which generated logs when the system crashed, so the errors that did not crash a system were excluded. For CPU-subsystem failures, a machined-check exception (MCE) was issued to notify the operating system. For DRAM failures, the bit flips in the Windows kernel code pages were logged. The read-only kernel code pages contain the OS kernel code and device drivers. Although the exact causes of such bit-flips are unknown, the authors confirmed that DRAM actually contained the erroneous values and that the bit-flips were less likely to be caused by buggy DMA. The failure rates of CPU and DRAM are not trivial: 1 in 190 for CPU subsystem (MCE) and 1 in 1700 for DRAM (one bit-flip) during a period of 30-day total accumulated CPU time (TACT). Furthermore, recurrent failures are common; the likelihood of hardware crashing again after the first crash increases by up to two orders of magnitude. Due to the lack of hardware support for diagnosing errors, the authors did not trace back to the root causes of the errors. Neither silent data corruptions nor application crashes were included in the study. However, the study does provide the evidence of bit flips in kernel code pages in the real world. We also infer that the total hardware fault rates are higher than the reported failure rates since not all faults trigger failures.

The study of failure rates for COTS server-grade processors is less prolific than that of DRAM or SRAM. Shazli et al. [2008] presented a case study for 32-bit server-grade processors used in information systems. Two anonymous system types, A and B, were included in the study. Parity was used to protect L1 cache and tags of L2 cache, and L2 cache data was guarded by ECC. The field data was collected from thousands of systems in various locations over a period of 3 years. The authors gave detailed examples of a probable SEU in the instruction pointer, a bit flip in the address of a function, and a bit flip in the instruction cache. Since the FIT (failure in time) data was considered as sensitive information, the absolute FIT rates were not revealed.

2.1.6 Summary

Transient hardware faults have significant impacts on the reliability of CMOS devices. We first introduced the physical process of alpha particles or neutrons striking silicon crystals and then described how such strikes can impact DRAM, SRAM, and logic circuits. Scaling feature sizes, reducing supply voltages, lowering noise margins, and increasing clock rates make CMOS devices more susceptible to transient faults. Several field studies demonstrate that the reliability issue of COTS hardware is real. Considering the trends of transient faults and the unlikelihood of adopting advanced error detection and correction techniques in high-volume COTS hardware due to the increased costs and energy consumption, transient hardware faults are not going to vanish in the near future.

2.2 How Transient Hardware Faults Impact Software

In this section, we first clarify the meanings of the terms faults, errors, and failures. We also introduce the classification of errors and point out that silent data corruptions can be harmful. Then, the security impacts of hardware faults are revealed as the main motivation of our work.

2.2.1 Classification of Errors

Obviously, not all transient faults will lead to errors. Some faults are removed by the software or hardware masking effects. When faults propagate to the architecture interface between software and hardware (e.g., faults in registers, caches, and memory), they become *errors* and may corrupt results of computations. Subsequently, the incorrect results may lead to system *failures*. Errors can be further categorised according to their severity and consequences, as shown in Figure 2.2 [Weaver et al., 2004]. The errors marked as benign or corrected do not affect the correctness of output data. For example, a memory error is masked if the memory location is overwritten by a new value before the previous erroneous data is used. Another example would be a single-bit flip detected and rectified by ECC circuits.

Detected unrecoverable errors (DUEs), for instance, multi-bit errors beyond the reach of SDC-DED ECC, can cause terminations of applications or kernel panics. As DUEs manifest as observable abnormal events, we can take actions to prevent them from propagating and causing severe consequences. Silent data corruptions (SDCs) are much more difficult to tackle—we are not aware

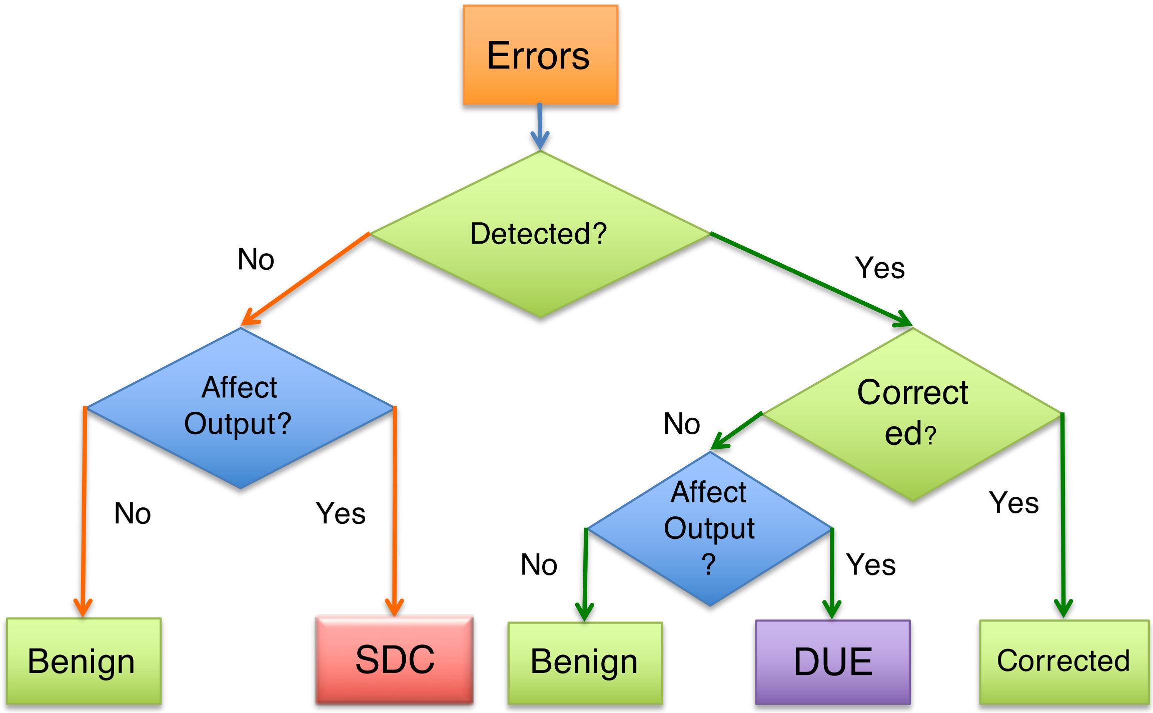

Figure 2.2: Classification of errors

that the data in registers, caches, or memory has been corrupted and that the computation process has produced incorrect results. Fault-tolerant mechanisms aim to convert DUEs to corrected errors and SDCs to DUEs or corrected errors, improving overall system dependability.

2.2.2 Security Impacts

Conducting software-based fault injection are often used to investigate how errors impact software systems (see [Hsueh et al., 1997; Ziade et al., 2004] for surveys on fault injection tools). We only include the studies that put emphasis on security implications of the errors induced by transient faults.

Xu et al. [2001] injected single-bit flips to code segments of FTP and SSH servers by using NFTAPE to evaluate how errors impact security of the servers. The fault injection approach is called *selective exhaustive injections*: only branch instructions in the code segments that are critical to system security are selected for injections (selective), and every bit of selected branch instructions is injected (exhaustive; one single-bit flip for each run). 7,432 runs were executed for the FTP server, and approximately 88% of the injected errors were never activated. The low activation rate was attributed to the fact that many of the code blocks were not reachable for a particular client request. Among the activated errors, about 38.5% had no impact; around 52% caused server crashes; and approximately 9% introduced fail silence violations. Most importantly, the security of the FTP server was compromised in 7 runs (1.07% of activated errors) in which unauthorised accesses with an invalid password were allowed. 2,664 runs were conducted for the SSH server. About 40% of injected errors were never activated, and the authors attributed the higher activation rate to the fact that the implementation of the SSH server was more compact than that of the FTP server. Out of the activated errors, the percentages for no impact, system crashes, and fail silence violations were 40%, 52%, and 7.5%. In 19 cases (1.5% of activated errors), a user with incorrect password was allowed to access the system. In a following-up study, Chen et al. [2004] examined how errors affected Linux kernel packet filters/firewalls: IPChains and Netfilter. Similarly. single-bit flips were injected exhaustively to the instructions of four critical firewall functions. For IPChains, 78% of the injected errors were not activated, and 20% of the errors caused crashes or hangs. 2% and 0.5% of the errors led to temporary and permanent security vulnerabilities that allowed unapproved packets to pass. For Netfilter, the numbers were 57%, 42%, 1.9%, and 0.02%.

Govindavajhala and Appel [2003] demonstrated that a single-bit flip induced by heating PC (personal computer) memory with a light bulb can be exploited to gain full control over a Java VM at a high success rate. The idea is to fill the heap with many objects of type B and a single object of type A. All the fields of the type B objects are pointers to the single type A object that has the following fields: a pointer to a type B object, an integer, and pointers to itself. By carefully arranging the fields and sizes of both types, a single-bit flip in a field of a type B object has a good probability of changing the field to reference a B type object. Thus, the static type of the field, type A, mismatches the actual referenced object type—type B. Subsequently, the runtime mismatch can be exploited to read and write arbitrary virtual memory addresses of the Java VM.

The *RowHammer* DRAM bug, which exits in many DRAM modules manufactured by all three major DRAM suppliers, provides a software-controlled and repeatable method to introduce bit flips in DRAM [Kim et al., 2014]. Since DRAM cells are getting smaller and closely packed together, frequently activating a row can introduce disturbance errors to adjacent rows. Software can simply use memory read and cache flushing (`CLFLUSH` on x86) instructions with row-conflict addresses in a loop to activate the bug. The bug has been exploited to break out of the Native Client sandbox in Chrome and to gain access to all physical memory on Linux machines [Rowhammer]. The latter attack is achieved by populating memory with page table entries and exploiting a bit flip in an entry. The consequence of a successful attack is complete control over the targeting machine. Although the bit flips are not directly related to particle strikes, the bug does corroborate the severity of the hardware reliability issue.

2.2.3 Summary

Although a significant fraction of the hardware faults is masked by various hardware/software masking effects, the remaining faults propagated to architectural interface can cause errors that corrupt data or alter control flows. The affected software may exhibit observable anomalies: page faults, invalid instructions, early terminations, or system crashes. Furthermore, silent data corruptions can induce security vulnerabilities or affect data integrity. Thus, the need for protecting software components from hardware errors is evident and immediate.

2.3 Relevant Concepts in Fault Tolerance

In this section, we introduce concepts and principles guiding our design and implementation. The *sphere of replication* (SoR) helps us identify components protected by redundant co-execution and vulnerable parts that should be guarded by other measures. *State machine replication* is a simple yet powerful approach to achieving fault tolerance, and it is mostly used to protect applications for improved availability in the case of server crashes.

2.3.1 The Sphere of Replication

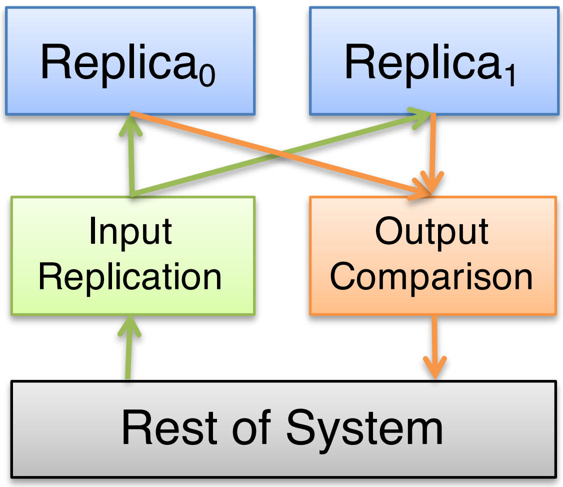

Figure 2.3: The sphere of replication

The sphere of replication (Figure 2.3) is introduced by Reinhardt and Mukherjee [2000] to describe the physical or logical redundancy in fault-tolerant systems. The root of this concept dates back to the early years of replication-based fault-tolerant systems, before the term (SoR) was coined. *Replica*$_0$ and *Replica*$_1$ are the parts of the system that are executed redundantly. *Input replication* ensures that the replicas observe consistent input data from the rest of the system so that they will not diverge, and *output comparison* checks for inconsistency of output data from the replicas for error detection. Although the concept originally was introduced in the context of hardware-based fault tolerance, it can also be applied to software; and we will use this concept to analyse each component in our proposed architecture in Section 4.1.3. The components carrying out input replication and output comparison are vulnerable if they are not self-checked or redundantly executed. Thus, one of our design decisions is to conduct output comparisons independently on each replica so that the comparisons are also protected by redundant co-execution. Input replication is not protected in the current framework. If an error occurs during the input replication step, there are two possible outcomes: (1) Inconsistent input data is provided to the replicas so that the replicas diverge and/or produce different output data. We design error detection mechanisms to capture the divergence or output irregularity to detect the errors. (2) The same but incorrect input data is provided to the replicas so that the replicas do not diverge and/or produce the same erroneous output data. As to the second case, we suggest adopting information redundancy (e.g., parity and CRC) to protect input data integrity so that corrupted data can be detected.

2.3.2 State Machine Replication

State machine replication (SMR) [Schneider, 1990] is an extensively studied and well-understood approach for building fault-tolerant services. The principles (or requirements) of SMR can be generalised as the following items.

1. Duplicate a service to one or more machines or processors and ensure that all replicas have consistent initial states.

2. Guarantee that all replicas observe the same input data or requests, in the same order.

3. Build state transition functions in a way that they produce output data or responses only based on current state and input data or requests.

4. Given a state and input data or a request, the new state and the output data or response of any correct replica are always the same (*determinism*).

Studies apply SMR on networked servers [Cui et al., 2015; Guo et al., 2014; Kapritsos et al., 2012]. Consensus protocols, for example, Paxos [Lamport, 1998] and Raft [Ongaro and Ousterhout, 2014], are used to ensure that the replicated services agree on the order and type of incoming requests to be handled. Replicating user-mode applications, as shown in Section 3.2.2, is straightforward and mature. To satisfy the requirement of consistent initial state, properly initialising variables and disabling address space randomisation (only if the control flow relies on virtual addresses) are sufficient. The consensus protocols are designed to fulfil the requirement of observing the same input data in an orderly manner. Various approaches described in Section 3.3 help to tame non-determinisms. Consistent state transitions and output data follow naturally if the first three items above are satisfied.

One common way of applying SMR to an application is to build an *active-standby* pair on physically separated machines so that the standby machine can keep operating should the active machine crash, improving the system availability. The *active-active* pattern is also prevailing since two active machines both serving incoming requests improve hardware efficiency. Our aim of implementing SMR is significantly different from the common active-standby and active-active patterns: our goal of applying SMR to a whole software system including the lowest-level system software is to prevent the errors induced by transient hardware faults from becoming silent data corruptions that impact system integrity and plant security vulnerabilities that can be exploited by attackers to compromise a system.

2.4 seL4 Microkernel

The formally verified seL4 microkernel is a member of the L4 family [Elphinstone and Heiser, 2013]. The formal verification of the kernel proves that the behaviours and security properties defined in the high-level design specification are implemented correctly in the C code, eliminating software bugs [Klein et al., 2009]. Furthermore, the microkernel can provide strong isolation guarantee [Elkaduwe et al., 2006; Murray et al., 2013; Sewell et al., 2011] that is indispensable for building dependable computer systems on which trusted and untrusted components can be integrated. This section, partially based on "seL4 Reference Manual" [seL4], introduces important concepts of the microkernel.

2.4.1 Capability Space and Resource Management

The access control model of seL4 is based on capabilities [Dennis and Van Horn, 1966]. System resources, for example, memory and I/O devices, are governed by capabilities that are unforgeable tokens with access rights. During initialisation, seL4 creates capabilities according to detected hardware configuration and stores them in a table called *CSpace* (Capability Space). Each capability is addressed by an integer called *CPtr* (Capability Pointer); and capabilities can be copied, moved, or revoked if the owners of the capabilities have authority to do so. The initial CSpace is passed to a user-mode bootstrapping thread (i.e., the *root task*) that is responsible for further initialising the whole system according to predefined policies that specify how the bootstrapping thread creates other threads and distributes the capabilities to other threads. Once the system initialisation is done, the bootstrapping thread typically halts itself; and thus each thread can only manipulate the capabilities assigned to it. Without explicit communication channels (*endpoints*) established between two threads, the threads cannot copy or move capabilities; so the authority to resource represented by capabilities is confined within the threads respectively.

Kernel objects, for instance, threads, endpoints, virtual memory objects, and even capability spaces, are managed through capabilities. The kernel objects are created through *retyping* memory regions which are represented by *untyped capabilities*. These untyped capabilities contain physical addresses and lengths of memory regions. A thread is allowed to create a kernel object only if it has an untyped capability which satisfies the required memory size of the kernel object, so the number of kernel objects that can be created by the thread is determined by the untyped capabilities owned by the thread. The advantage of this model is that all kernel object creations become explicit, preventing kernel memory exhaustion attacks mounted by malicious threads creating kernel objects repeatedly. The names and descriptions of the kernel objects are listed in Table 2.1.

After the kernel initialises the root task, the kernel creates untyped capabilities that cover all remaining unused physical memory regions and inserts them to the CSpace of the root task, delegating all physical memory management to user-mode applications. This design avoids the complexity of memory management in the microkernel, but it also exposes the physical memory addresses to applications. The kernel also creates various *device frame capabilities* and *I/O port capabilities* (x86 only) according to I/O devices available on a system, and these capabilities are given to the root task as well. Section 5.1 discusses in detail how user-mode device drivers use these capabilities to communicate with I/O devices. For now, we only need to know that the root task distributes these capabilities to device drivers so that the drivers are able to access I/O devices.

One important fact we need to emphasise is that kernel objects can be treated as extensions of kernel data structures although they are created at the discretion of user-mode threads. For example, the virtual address space objects resemble the multiple-level in-kernel page tables used in other kernels. Admittedly, the seL4 microkernel consumes an insignificant amount of memory after it finishes initialisation; but the memory usage of the kernel grows as more kernel objects are created.

Type	Description
CNodes	Capabilities are stored in CNodes, and each CNode has a fixed number of slots determined when the CNode is created.
Thread Control Blocks	TCBs represent kernel scheduling entities, called threads, in seL4; architecture-dependant registers files are saved in TCBs when threads are blocked.
IPC Endpoints	Endpoints are the main inter-process communication mechanism provided by seL4. Threads send, reply, and wait for messages through endpoints. Furthermore, capabilities can also be transferred through endpoints.
Notification Objects	Notification objects are provided as a signalling mechanism. A notification object contains a word-size array of flags, and each flag works like a binary semaphore. Hardware interrupts are delivered to user-mode drivers through notification objects. See Section 2.4.4
Virtual Address Space Objects	Virtual address space objects are used to construct virtual memory address spaces, which are consulted by processors to translate virtual memory addresses to physical memory addresses. See Section 2.4.3.
Interrupt Objects	Interrupt objects allow user-mode drivers to receive and acknowledging interrupts. A capability to the `IRQControl` object controls the creation of `IRQHandler` capabilities. An `IRQHandler` capability associated with an interrupt source can be delegated to a driver so that the driver is allowed to wait for and acknowledge the interrupt source (Section 2.4.4).
Untyped Memory	Untyped memory are kernel objects representing unused physical memory regions; each untyped object contains the start kernel virtual address and size of the region described by the object. The *retype* operation applied on untyped objects can create other kernel objects. Also, an untyped object representing a large memory region can be divided into several untyped objects describing smaller memory regions.
I/O Ports	I/O port objects only exist on x86 machine, representing the authority to access ranges of hardware I/O ports.

Table 2.1: seL4 kernel objects

2.4.2 Threads, Scheduling, and Inter-Process Communication (IPC)

The seL4 adopts a single-kernel-stack and non-preemptible kernel execution model to tackle the limitations of formal verification. Kernel system calls or interrupt handlers run to completion without being disturbed by interrupts, thus limiting concurrency inside the kernel to reduce complexity of verification. Some kernel operations may execute for a significantly extended period with interrupts disabled, so several long-running functions feature preemption points to reduce the interrupt latency. The preemption points poll for pending interrupts; and if there is one, the kernel suspends and backs out of the current system call, and then invokes the interrupt handling code. The suspended system call will be restarted once the interrupt handling is done.

The kernel scheduling entities of seL4 are threads. Threads are created through retyping untyped capabilities to thread control blocks (TCBs); a TCB contains the following fields:

The priority of a thread is used by the kernel scheduler to determine the next thread to run.

The state of a thread represents the current state; the states are: `Inactive`, `Running`, `Idle`, Restart, BlockedOnSend, BlockedOnReply, BlockedOnReceive, BlockedOnFault, and BlockedOnNotification.

User-mode registers are used to save user-mode register contents when a running thread traps to kernel mode due to interrupts, system calls, or exceptions. The contents are architecture-specific and are restored to hardware registers when the thread is resumed. The kernel can examine the saved values of the user-mode registers.

The remaining time slices record how long a thread can keep executing on the current core before the kernel preempts the thread and invokes the scheduler.

The type of fault indicates the type of fault that a thread triggers. The exact type is architecture-dependant. The common ones are: invalid addresses (data and instructions), invalid instructions, debug exceptions, etc.

The capability pointer to a fault handler tells whether there is a handler (another user-mode thread) for handling faults triggered by a thread. In the case that a valid handler exists, the faulting thread is suspended; the kernel resumes the handler thread and passes fault information to the handler.

The initial capability table of a thread is allocated together with a TCB, and the table contains several frequently used capabilities pre-populated by the creator of the thread.

The physical address of a thread's IPC buffer is used together with the IPC buffer capability to obtain the kernel virtual address of the IPC buffer so the kernel is able to pass messages between threads.

The pointer to a notification object usually is used to notify a thread of incoming hardware interrupts if the thread is a driver.

seL4 has 256 priorities and schedules ready-to-run threads with the same priority in a round-robin fashion. A platform-specific hardware timer is configured to generate periodic interrupts that are called kernel ticks. The length of a tick is configurable, so is the number of ticks allocated to running threads. When a thread uses up all its time slices, the kernel iterates the runnable thread queue from high priority to low priority and chooses the first runnable thread that has higher or equal priority (compared with the current thread) as the next thread to schedule. If such next-to-run thread exists, the current thread running out of slices is put back to the run queue; otherwise, the current thread is scheduled to run again. In both cases, the time slices of the thread are replenished. Among the valid state transitions, only the transition from BlockedOnNotification to Running is triggered by non-deterministic device interrupts. All other transitions are results of system calls or exceptions which are deterministic. Therefore, the scheduler of seL4 is deterministic provided that we ensure the deterministic observations of device interrupts.

Systems based on the seL4 kernel depend on fast IPC (Inter-Process Communication). seL4 implements IPC with endpoint kernel objects and adopts the rendezvous model. The capabilities to endpoints are used as parameters of seL4_Send, seL4_Call, seL4_Reply, seL4_Recv and seL4_ReplyRecv system calls. An endpoint blocks a sender until a receiver is ready. Likewise, a receiver is queued until a message arrives.

2.4.3 Virtual Memory Space

Virtual memory spaces are vital for isolation. Modern processors are equipped with memory management units (MMUs) to translate virtual addresses to physical memory addresses, enforcing isolation between threads by separating physical memory regions used by the threads. An MMU walks through multiple-level page tables, using different fields of a virtual address as indexes into various levels of the page tables. The last-level page tables contain the final corresponding physical address. Different architectures implement the table formats and lookup mechanisms differently [ARM, 2014; Int, 2016b], but the principle of address translation stays the same. When the kernel preempts the current thread and switches to another one, the control register holding the start address of the active translation tables is also updated with the address of the next thread's page tables; so the processor can start to use the new mappings. Because all threads share the kernel, the higher part of the virtual address space (usually the top 1 GiB for a 32-bit kernel, but configurable) is reserved; kernel code and data sections are mapped to the reserved regions of all threads.

The virtual memory space of the root task is set up by the kernel during system bootstrapping; all other VM spaces are created by retyping untyped objects into various VM kernel objects: page directory objects, page table objects, and frame objects with different sizes. System calls are provided to map frame objects and intermediate page table objects (if the tables do not exist yet) into a page directory at specified virtual addresses. As we will see later, these system calls are augmented with new options to support device driver replication (Section 5.4.1) and error masking (Section 5.7.3).

2.4.4 User-Mode Drivers and Interrupt Delivery

Device drivers for seL4-based systems run in user mode; they are just normal applications, except that they have capabilities representing I/O ports and device memory-mapped I/O regions. Also, drivers can receive device interrupts delivered by the kernel through notification object capabilities.

Different architectures have different approaches in terms of organising how I/O device registers are accessed by software. For example, ARM SoCs (system on chip) usually define the layout of physical addresses statically, specifying which regions are mapped to physical memory and which regions are backed up by I/O device registers. Intel platforms keep I/O ports for backward compatibility of legacy devices, for instance, programmable interrupt controller (PIC), programmable interval timer (PIT), and communication port (COM); and modern PCI devices are discovered dynamically by PCI bus scanning [Shanley and Anderson, 1999]—physical addresses for PCI devices are stored in the base address registers (BAR) of the PCI configuration spaces that are located during scanning. During the initialisation stage, the kernel creates capabilities that cover I/O ports and device register regions. These capabilities are passed to the root task that is responsible for further distributing the capabilities to corresponding device driver threads.

The kernel also creates an IRQ (interrupt request) control capability that is used by the root task to create an `IRQHandler` capability for each interrupt source in the system; the resulting capabilities are delegated to driver threads. We introduce how interrupts are delivered to user-mode device drivers since understanding the delivery procedure helps readers to perceive how we include the drivers in redundant co-execution (Chapter 5). The pseudo code in Listing 2.1 is the usual pattern used by the drivers to receive and handle interrupts. We first create a notification object (ntfn) by retyping an untyped memory object, and then get an IRQHandler capability (irq_handler) that can be used to bind an IRQ number to the notification object (line 3) and to unmask the IRQ source by acknowledging the handler (lines 4 and 10). The main body of the driver is the while loop (lines 5 to 11). The driver waits for an interrupt by calling seL4_Wait with the capability of the notification object, and the kernel sends a message to the notification object if an interrupt associated with the IRQ specified at line 2 is received by the kernel. If the driver has sufficient priority, the kernel will schedule the driver to run; so the driver starts to execute the device-specific interrupt handling procedure. The driver checks if the device issues an interrupt and the type of the interrupt, and then handles the interrupt accordingly. Finally, the interrupt source is unmasked at line 10 so that further interrupts from the device controlled by the driver can be delivered.

```
1  ntfn = retype_to_ntfn_object(untyped);
2  irq_handler = seL4_IRQControl_Get(irq_num);
3  seL4_IRQHandler_SetNotification(irq_handler, ntfn);
4  seL4_IRQHandler_Ack(irq_handler);
5  while (true) {
6    seL4_Wait(ntfn, &badge);
7    if (dev_irq_active(dev_regs)) {
8      handle_irq();
9    }
10   seL4_IRQHandler_Ack(irq_handler);
11 }
```

Listing 2.1: Handling I/O device interrupts

The user-mode device driver architecture employed by various microkernels is significantly different from monolithic kernels that include a considerable amount of driver code in kernel mode and execute the drivers with the highest privilege.

2.4.5 Non-determinism Analysis of seL4-Based Systems

Since our approach aims to apply SMR (state machine replication Section 2.3.2) to an seL4-based system, we need to understand the sources of non-deterministic events in the system so that the replicas of the system do not diverge during error-free runs. As a start, we discuss the determinism of user-mode applications. Following that, for a whole-system replication approach, we also consider the seL4 kernel and device drivers—components that are usually ignored in other SMR approaches, which only replicate selected applications. It is worth repeating that we focus on a seL4-based software system running on a single core.

Deterministic and non-deterministic events

Before diving into the details, we informally introduce the meanings of *deterministic* and *non-deterministic* events in the context of seL4-based systems. From the kernel's point of view, we call the events that are intrinsic to applications and can be observed by the kernel in program order as deterministic events. For seL4-based systems, system calls issued and exceptions triggered by user-mode threads are deterministic events. If we replicate a single-threaded process onto different cores with the same initial state and ensure that the system calls interacting with external environments return the same results for the replicas, the sequences of deterministic events and output results on different cores will be the same. In other words, deterministic events do not cause replicas of a system to diverge. Non-deterministic events come from two sources: (1) external environments, specifically interrupts and input data from I/O devices; (2) concurrent accesses to shared memory regions. Non-deterministic events can cause replicas of a system to diverge so that we need to tame them.

Non-deterministic events in applications

We start from single-threaded applications which only interact with other parts of the system through system calls (reading/writing files, sending/receiving data to/from sockets, etc.). For such an application, feeding the replicas of the application with consistent system call return values is sufficient to ensure the deterministic execution of the replicas. On ARM and x86 architectures, applications may try to read CPU timestamp counters and performance counters directly by using inline assembly, and the counters can return different values to the replicas. Such operations must be trapped so that the supporting library or kernel can have the opportunity of providing the same values to the replicas. In summary, controlling system calls and accesses to CPU performance and timestamp counters is sufficient to guarantee the single-threaded replicas not to diverge.

Multi-threaded applications inherently possess another source of non-determinism—concurrent accesses to shared variables from multiple threads. A thread and its replica can diverge if they observe different values from a shared variable and the shared variable's replica, depending on if the updates (made by other threads) to the shared variables have been performed or not. If all shared variables of a multi-threaded application are protected by synchronisation primitives, the application is *data-race-free*. Ensuring consistent locking/unlocking order across replicas is adequate for data-race-free multi-threaded applications to eliminate the non-determinism from concurrent accesses [Olszewski et al., 2009]. For applications with data races, in addition to ensuring consistent locking order, updates to shared variables are controlled by runtime libraries [Bergan et al., 2010; Liu et al., 2011] so that the updates are committed deterministically. Section 3.3 will describe approaches to deterministic execution of multi-threaded applications. We will also discuss support for replicating multi-threaded applications on seL4-based systems in Section 4.5.1.

Non-deterministic events in device drivers

I/O devices are primary sources of non-deterministic events since they generate unpredictable interrupts and that I/O operations have side-effects. In our approach to redundant co-execution, a device driver is also replicated onto multiple cores so that multiple replicas of the driver can compete for a physical device. As a straightforward example, if reading the status register of a device has the side-effect of clearing the register, the return values of reading operations issued by the driver replicas on different cores depend on the order of the read operations. Another example would be accessing DMA (direct memory access) buffers: the device is unaware of driver replication, so only one of the driver replicas gets input data from the DMA buffers. As to interrupts, they are usually routed to a designated core; the driver replica running on the designated core receives the interrupts through the notification object. However, other driver replicas are not notified. On platforms (e.g., x86 with IO-APIC and ARM with GICD) supporting broadcasting interrupts, all cores can get interrupts for an interrupt source, but the timing can be significantly different, leading to divergence of the replicas. The support for device driver replication and coordinating the driver replicas to observe input data and interrupts consistently are crucial for replicating the the whole system, so we dedicate Chapter 5 to discuss the issues in-depth and to describe the mechanisms for driver replication.

Non-deterministic Events in the seL4 kernel

The seL4 kernel is invoked in three circumstances: (1) executing system calls explicitly issued by applications, (2) handling interrupts from I/O devices, and (3) processing various exceptions triggered by applications. The category (2) is related to I/O devices; device interrupts non-deterministically interrupt user-mode applications, causing switches to kernel mode (interrupts are disabled in kernel mode). Further, the interrupts can be classified as kernel preemption interrupts that trigger rescheduling (the interrupts from the kernel preemption timer) and device interrupts that are forwarded to user-mode device drivers. The steps for handling the kernel preemption timer interrupts include decreasing the time slice available for the currently running thread and context switching to a new thread if the current thread uses up its time slices. Similarly, forwarding the device interrupts to user-mode drivers also involves preempting the current-running thread. If the preemptions happen on different cores without coordination, the current threads running on different cores can be preempted at different instructions; or even worse, the preempted current threads can be replicas of different threads. For instance, thread A is preempted on core 0, but thread B is preempted on core 1. The uncoordianted preemptions can lead to divergence.

The system calls provided by the seL4 kernel are quite different from the ones supplied by traditional monolithic kernels. Only functions or mechanisms enabling safe resource sharing are kept in the microkernel, and the kernel objects described above (Table 2.1) are manipulated by respective owners through capabilities to achieve various system configurations. The seL4 kernel only provides `Send`, `NBSend`, `Call`, `Wait`, `Reply`, `ReplyWait`, `Poll`, and `Yield`. Detail descriptions of the system calls can be found in [seL4]. The system calls, except `Yield`, take a capability (a handle to a kernel object) as the first parameters and modify the state of the kernel objects according to additional system call parameters provided. Therefore, we need to examine the kernel objects to determine the determinism of the system calls. The criterion is simple: the state of a kernel object is affected by non-deterministic events or not. In kernel mode, we only need to consider the non-deterministic events from I/O devices; thus, the following kernel objects are non-deterministic:

Notification objects are used for interrupt delivery so that they are directly affected by non-deterministic interrupts. For this reason, the system call `seL4_Wait` is non-deterministic when it is used with a notification object capability that is bound to an interrupt.

IO port objects (x86 only) are used for controlling accesses to hardware I/O ports. Although the objects are not changed by the hardware directly, the results returned by hardware are non-deterministic. Consequently, the following system calls, `seL4_IA32_IOPort_In8`, `seL4_IA32_IOPort_In16`, and `seL4_IA32_IOPort_In32` are non-deterministic. (Note that these I/O port calls are just wrappers of the `seL4_Call` system call.)

Thread control blocks (TCBs) contain the user-mode registers of threads when the threads are preempted. Preemptions are triggered by kernel preemption timer interrupts or I/O device interrupts that are not deterministic. Therefore, the system call, `TCBReadRegisters`, returns register values non-deterministically.

In short, for systems based on seL4, we consider system calls and application triggered exceptions as deterministic events; we also identify interrupts, unguarded memory accesses to shared memory regions by multiple threads, and input data from I/O devices as non-deterministic events. We also identify the non-deterministic system calls that need special attentions. In Chapter 4 and Chapter 5, we describe how our redundant co-execution approach deals with these non-deterministic events to ensure the divergence-free execution of the system replicas during fault-free runs.

2.4.6 Summary

Being a formally verified microkernel, seL4 provides strong isolation guarantee by adopting the capability model for resource management and upholding the user-mode policies that prescribe the capability distribution. Delegating resource management to user-mode servers and implementing user-mode device drivers effectively reduce the complexity of kernel implementation, but the approach also implies that user-mode components involving in managing resources (e.g., a memory pager) or providing services (device drivers) must behave according to their specifications. Especially, a root task has all the initial capabilities and is responsible for setting up a system. The correct execution of the root task is vital to the security of the system since it controls capability distribution and system initialisation. From the security point of view, we cannot simply protect a high security level application by executing it redundantly and checking the results, especially when the application co-locates with other applications with lower security levels. This is because the system software (i.e., the kernel and other user-mode servers managing hardware resources) is unprotected so that any deviation of the system software from specifications may cause unexpected security violations.

Chapter 3

Related Work

Opposites are complementary.

Niels Bohr

As hardware faults are common and can cause serious consequences, system designers acknowledge the existence of hardware faults and develop various techniques to tolerate hardware faults for improved system reliability. Hardware solutions implement redundancy at different levels, ranging from replicated processor functional units inside a processor to redundant physical machines connected with high-speed interconnection fabrics. These solutions have the advantages of improved fault coverage, short fault detection latency, fast recovery, strong fault isolation, and low performance overhead; they can tolerate both transient and permanent faults, preventing costly system downtime and preserving service availability even when a faulty component is being replaced. However, replicated hardware components also imply increased size, weight, cost, and power consumption. The thesis focuses on building systems using commodity hardware; still, several representative hardware solutions are introduced in Section 3.1 for completeness.

Software-based fault-tolerant or error-detection-only approaches are flexible and inexpensive, but, as far as we know, all of them assume that the underlying system software layer is not affected by hardware faults, or that some hardware components are always correct. Unfortunately, these assumptions limit the applicability of the software solutions, which will be discussed in detail in Section 3.2. In contrast, our approach explicitly includes the microkernel into the sphere of replication (SoR) and protects the kernel with redundant co-execution, aiming for a self-checking kernel running on COTS hardware. Our SMR-based approach requires deterministic execution of the replicas when multithreaded applications are deployed, so the related work for building deterministic systems is also covered in Section 3.3.

3.1 Hardware Solutions

We first introduce error detection and correction methods based on *circuit-level redundancy* to protect processors and memory. The increased die area, energy consumption, and latency need to be considered when choosing a protection method for a particular circuit element. *System-level redundancy* systematically duplicates all hardware components (processors, memory modules, power supplies, I/O devices, buses, etc.) used by a system to construct multiple running replicas of the hardware system, aiming for very high level of reliability. The functionality of the system is unaffected except for potential performance degradation if one or more (in some approaches) replicas crash.

3.1.1 Circuit-Level Redundancy

Processors

Processors themselves can be enhanced with error checking features for early detection of faults. Storage elements of the processors are normally protected by coding techniques: ECC or parity is commonly used to detect bit flips in caches, registers, latches, TLB, etc. As to the pipeline protection, the techniques can be classified into two categories: spatial redundancy including lockstepping, redundant multithreading (RMT), heterogeneous cores, etc.; and information redundancy consisting of AN codes [Peterson and Weldon, 1972], residue codes [Garner, 1966; Lipetz and Schwarz, 2011], parity predication [Nicolaidis et al., 1997], and so on. We focus on several representative spatial redundancy approaches to protecting logic components and then briefly introduce commercial processors embracing information redundancy.

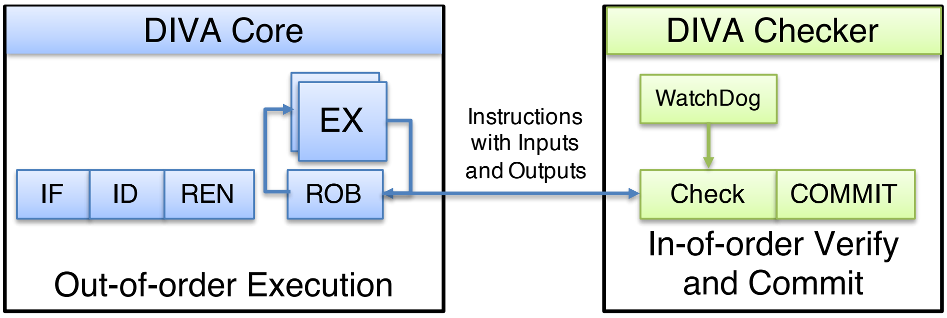

Figure 3.1: Dynamic implementation verification architecture (DIVA)

As shown in Figure 3.1, DIVA [Austin, 1999] proposes using a simple, electrically robust, reusable, and latency-insensitive checking unit, which is suitable for formal verification, to verify the correctness of a high-performance complex core. Instructions completed by the complex core are shipped to the checker with inputs and outputs, and the checker re-executes the instructions with the supplied inputs and compares the re-computed outputs with the given outputs. If the results differ, the checker throws an exception to indicate an error. The exception is handled at the *commit* stage of the checker, and the result produced by the checker is used to fix the errant instruction. The register and memory input operands for each instruction are also verified, but DIVA requires all architectural register files and memory be protected by coding schemes. The author claims that the performance impact is very limited based on detailed timing-simulation results. The reliability of the DIVA checker is vital to the correct operation of the processor since the results produced by the checker are used to rectify errors. Thus, if the checker fails, erroneous results will be committed to architectural storage. The author informally argues that formal verification can guarantee the correctness of the simple checker and that using slow and large transistors makes the checker less susceptible to transient faults. The main advantages of DIVA are that the checker consumes less die area than duplicating a fully functional pipeline and that the checker can be reused with new versions of the complex core to leverage the correctness guarantee.

Redundant multi-threading (RMT) runs two identical instances of an application thread independently and compares the outputs of the threads for error detection. Based on a simultaneous multi-threaded (SMT) processor, an RMT implementation called simultaneous and redundantly threaded (SRT) processor is designed and evaluated by Reinhardt and Mukherjee [2000]. The concept of sphere of replication (SoR) is introduced in this paper, and the authors analyse the SRT processor with the concept to identify that inputs, instructions, cached load data, uncached load data, and external interrupts must be replicated and that outputs, stores and uncached load addresses, must be compared. Mukherjee et al. [2002] apply the RMT on a dual-processor device and coin the term chip-level redundant threading (CRT). Loosely-synchronised redundant threads are generated transparently by the processor, and the leading and trailing threads are executed on different cores of a chip multiprocessor (CMP) to leverage the spatial redundancy and improved fault coverage. Both SRT and CRT use similar techniques for input replication and output comparison. We summarise the techniques so that we can compare them with our software-implemented replication and comparison methods in Section 4.1.3. In the case of SRT, memory stores are compared through a store buffer shared between the redundant threads: the leading thread puts the retired store addresses and values in the buffer, and the trailing threads compares its retired stores with the ones in the buffer. For CRT, a separate structure called *store comparator* monitors the store queues of the cores and compares the stores retired by the trailing thread with the corresponding ones retired by the leading thread. The consistent view of cached data is maintained by a *load value queue* that contains source addresses and values of load instructions issued by the leading thread; the trailing thread directly receives values from the queue for corresponding load instructions instead of probing caches. As to comparing uncached load addresses, the leading thread stalls the execution of an uncached load instruction until the trailing thread executes the same instruction so that the addresses can be checked. Thus, the threads synchronise for the uncached load instruction and keep synchronised until the load data is returned and replicated. An external interrupt is delivered to both threads precisely either by synchronising the threads before the delivery or by recording the point when the interrupt is delivered to the leading thread and redelivering the interrupt when the trailing thread arrives at the same point.

IBM G5 processors, used in IBM S/390 mainframes, are designed with replicated pipelines and ECC protected on-chip storage (register files, cache, etc.) [Slegel et al., 1999; Spainhower and Gregg, 1999]. S/390 mainframes are used where data corruptions must be prevented, so extensive error checking is integrated into all function components of the processors, even including combinatorial logic components that are considered difficult and time-consuming to check. To avoid prohibitive checking overhead, the designers duplicate the I-unit (fetching and decoding instructions) and E-unit (executing instructions). The R-unit (a register file) is protected by ECC and also stores the architecture checkpoint used for recovery. Caches are guarded by parity. The total area overhead for the fault-tolerant features is 35%. The processor executes an instruction independently twice with duplicated units, and results are compared before being committed to the R-unit. Should a comparison fail, the faulty instruction will be retried automatically. All the self-correcting activities of the G5 processor are transparent to software.

ARM implemented [ARM] a lockstep mode for the Cortex-R series processors that are designed for electronic systems that should provide uninterrupted services, in addition to ECC-protected storage components (caches, tightly coupled memory, TLB, etc.) and parity-protected branch prediction RAMs. The lockstep mode employs the second processor as a copy of the first processor to protect logic components, without duplicating on-chip RAM to cut down the area overhead; the input pins, the caches, and the TCM (tightly-coupled memory) of the first processor are shared by the two processors. The output signals of the duplicated components are checked by the comparison logic that can be customised during implementation.

Recent processors adopt coding-based techniques to detect errors in logic units instead of replicating a whole pipeline for reduced die-area overhead and power consumption, but at the risk of reduced fault coverage. The fifth generation Fujitsu SPARC64 processor [Ando et al., 2003] covers 80% of the 200,000 latches with parity. The ALUs and shifters of the processor are protected with parity predication, and the multiply and divide units are guarded by residue check and parity predication. Furthermore, the parity bits are carried along the data paths and checked again by the receiving components to detect errors introduced during transfers. Errors in pipelines of Itanium processors (9500 series) [Bostian, 2012] are detected by using residues that are produced by arithmetic units and by using parity bits for data and instructions moving through the pipelines. Once an error is detected, the *instruction replay* technology, implemented at pipeline level, re-executes the affected instruction and corrects the error automatically without software intervention. IBM's POWER processors employ comprehensive built-in fault detection features [Henderson; Henderson et al.; Reick et al., 2008; Sanda et al., 2008]. Starting from POWER6, most of data and control-flow units are protected by error-detection logic such as parity (for latches) and residue checking (for floating-point units). A recovery unit that is functionally similar to the R-unit in the G5 processor also exists, and the ECC-protected register file also serves as the architectural checkpoint so that IRR (instruction retry recovery), the primary method for handling transient faults, can try to correct a detected error based on the previous checkpointed state. If a permanent fault is detected, APR (alternate processor recovery) moves the checkpointed state from a failing core to a spare core and restarts execution.

Memory

The commonly-used error-correcting code for memory modules is SEC-DED (single error correction, double error detection) code [Moon, 2005]. DDR3 or DDR4 memory transfers 64-bit data at a time; for ECC DIMMs (dual in-line memory module), the width is 72-bit including a 8-bit error-correcting checksum to protect the 64-bit data. The 8-bit checksum is calculated based on the 64-bit data when the data is written to memory. When the data is read back, a new 8-bit checksum will be computed again based on the 64-data and compared with the original checksum for error detection. In a DIMM consisting of 4-bit memory chips, a 72-bit ECC word is stored in 18 chips; so a failure of a chip results in 4-bit data error that cannot be dealt by SEC-DED.

In order to tolerate a whole-chip failure, IBM developed *Chipkill Memory* [IBM, 2016] that distributes an ECC word across multiple DIMMs so that each memory chip only contains one bit of the ECC word. Take a 72-bit ECC word for example; 72 memory chips spanning 4 DIMMs are required (assuming 4-bit memory chips). In this configuration, a chip failure causes 4 single-bit errors in 4 different ECC words, and such errors can be corrected by SEC-DED. Other major vendors also implement similar technologies, such as *Advanced ECC* from HP [HP, 2016], *Extended ECC* from Oracle, and *SDCC* (enhanced DRAM single device data correction) and *DDDC* (enhanced DRAM double device data correction) from Intel [Intel, 2016a].

ECC and its advanced versions provide error correction for memory, but they do not implement failover capability that is required for uninterrupted service if an entire DIMM fails. High-end servers provide *memory mirroring* mode in which identical data is written to two memory channels simultaneously [Dell, 2016; Fujitsu, 2016; HP, 2016; Intel, 2016a]. If a read from the first DIMM fails because of an uncorrectable error, the memory controller automatically retries the read from the second DIMM. Memory mirroring increases reliability and availability of the memory subsystem significantly at the cost of halving the size of memory available.

Sridharan et al. [2015] conclude that SEC-DED ECC is insufficient to protect DRAM since that multi-bit upsets are becoming more frequent and that the memory capacity keeps growing. Advanced ECCs are capable of correcting MBUs, but they are not widely adopted mainly because of the increased cost. Additionally, more memory chips are required to achieve the same data bus width (72 chips for chipkill vs. 18 chips for SEC-DED ECC, assuming 72-bit words and 4-bit chips), making the advanced ECCs less suitable for embedded systems. In this thesis, we do not assume the memory is protected by SEC-DED ECC, Chipkill, or memory mirroring. Thus, our approach duplicates and maintains two or three copies of data of a system in memory for redundancy and error detection.

3.1.2 System-Level Redundancy

Tolerating hardware faults by using isolated and redundant processors, buses, memory modules, and I/O devices is a well-accepted approach in safety-critical computer systems [Brière and Traverse, 1993; Hopkins Jr. et al., 1978; Wensley et al., 1978; Yeh, 1996]. Usually, TMR (triple modular redundancy) is employed by such systems, so a faulty unit can be identified, recovered, and reintegrated without service interruptions. Redundant outcomes are voted onto form a final output, and the voting units can be made self-checking to further improve reliability.

Flight-control computers for airplanes are representative safety-critical systems that require ultimate reliability and integrity, so massive system-level redundancy and dissimilarity are the techniques used to achieve extreme levels of reliability and availability. Take the Airbus A320 fly-by-wire system for example [Brière et al., 1995]; 5 computers are simultaneously active, and the performance and safety of an A320 plane is unaffected if one computer fails. Furthermore, flying the plane is still possible when only one computer is active. Each computer has two separated channels: a control channel and a monitor channel. Each channel is a fully functional "subcomputer" with dedicated I/O units, power supply, memory, processors, etc. The monitor channel constantly compares the outputs of the control channel with its own outputs and stops the computer if the results of the two channels diverge significantly. The five computers are built with two different processor models to avoid common hardware failures. Two out of five are ELAC (elevator and aileron computers) that are built by Thomson-CSF with 68010 processors, and the rest three computers are SEC (spoiler and elevator computers) manufactured by SFENA/aerospatiale based on 80186 processors. The Boeing 777 embraces triple-triple redundant primary flight computers (PFC) [Yeh, 1996]. The system consists of a left PFC, a centre PFC, and a right PFC. Each PFC further comprises three lanes connected to an ARNIC 629 data bus with dedicated interfaces; each lane also has its own processor, power supply, and other peripherals. Three processor models (AMD 29050, MOTOROLA 68040, and INTEL 80486) are used by the lanes in one PFC to avoid the case that a single common-mode hardware fault brings down the whole PFC. Dissimilar ADA compilers compile the common software for the processors. One of three lanes in a PFC is the *command* lane, and the other two lanes are the *monitor* lanes. The outputs computed by the three lanes are compared against each other. Likewise, each PFC also checks its outputs with the outputs of the other two PFCs. From the two examples above, we can learn that dependable computer systems can be built on COTS processors by designing the systems with enormous redundancy and systematic engineering. In the case of A320, at least 10 processors are engineered to finish the task of one processor; and the number is 9 for Boeing 777. Additionally, hardware diversity is a must to avoid common-mode failures that can disrupt all processors.

For the enterprise market, NEC manages redundant Intel Xeon processors and chipsets in lockstep mode by using specially designed LSI chips called GeminiEngine™ that monitor all system transactions and error signals from the chipsets [NEC, 2011]. All primary components, CPU, memory, motherboards, I/O devices, cooling fans, and power supply units, are fortified by hardware redundancy to achieve uninterrupted service. The Xeon processors, chipsets, and GeminiEngine™ chips embed specifics lockstep functions that are developed jointly by Intel and NEC: (1) The XEON processors are updated with fault-tolerant specific firmware. (2) The chipsets are enhanced with technology to maintain determinism. (3) A special initialisation sequence is required for PCI Express. Thus, the processors in lockstep execute the same instructions cycle by cycle synchronously. Both processors are active so that a faulty component does not upset normal operation. The *DM* (data mover) in the GeminiEngine™ chips is also responsible for copying memory contents from the *primary* node to the *secondary* node when the system starts up or after a board is replaced. The operating systems observe the processors running in lockstep as a single processor. I/O devices adopt traditional failover approach: pairs of active-standby disks and network cards are visible to the processors so that device drivers can detect I/O failures of the active devices and control the switch over to the standby devices. This example shows that lockstepping two high-performance multicore processors is a challenging task that requires close collaboration with the processor manufacturer. Furthermore, development and validation costs to support new generations of processors could be significant.

The designers of NSAA (nonstop advanced architecture) [Bernick et al., 2005] recognise that running commodity multicore processors in lockstep is becoming more challenging because of dynamic core frequency scaling, increasing CPU frequency, and the fact that multicore processors may not expose individual cores through the sockets. As shown in Figure 3.2, NSAA loosely couples multicore processors to allow redundant instruction streams execute on different cores at different speeds. Each 4-way SMP server is called a slice; and in a TMR configuration, three SMP servers are employed. Each physical core of a 4-way multicore processor represents a processor element (PE). Three PEs belonging to different servers are grouped together as a logical processor. In the figure, the system has four logical processors. The physical memory of each slice is split and assigned to PEs for isolation. Sharing memory is prohibited so that the processors can only communicate through the redundant *ServerNet* SAN (system area networks) in a message-passing style. The *ServerNet* SAN is made up two independent fabrics, ensuring that a failure can at most disrupt one fabric.

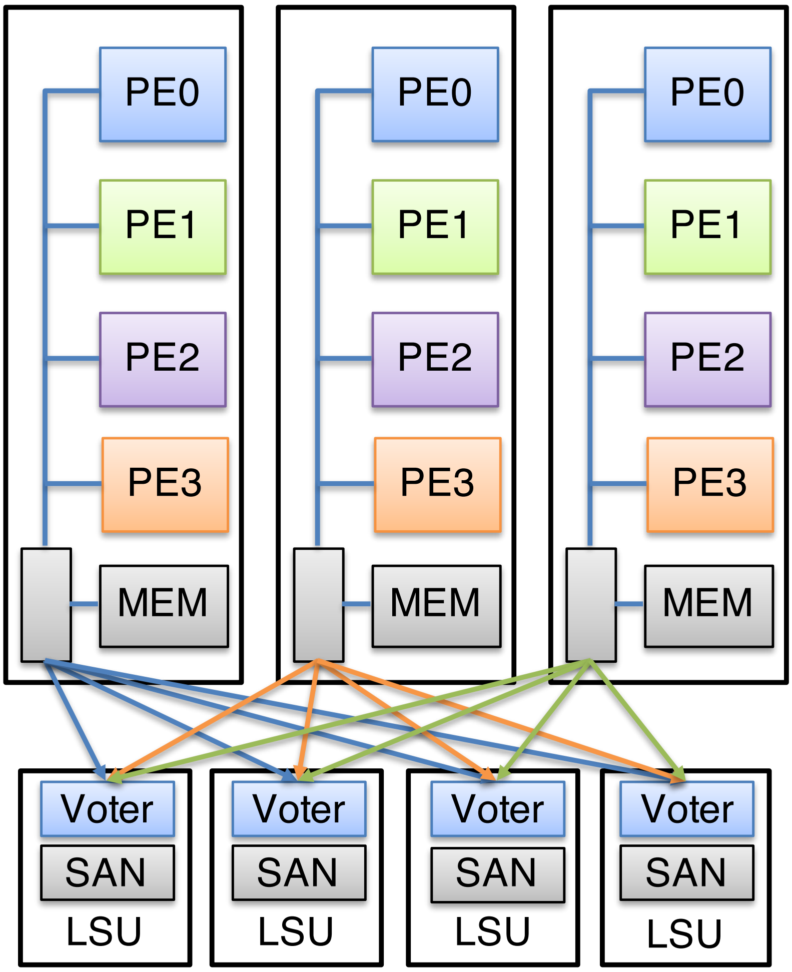

Figure 3.2: Non-stop advanced architecture

Each logical processor is connected with one or two self-checking logical synchronisation units (LSU), and each LSU includes a voter and a SAN interface. The voter compares IPC (inter-processor communications) and I/O from all PEs inside a logical processor for error detection and masking. The SAN interface is responsible for attaching the logical processor to the SAN so that the processor can issue I/O requests and receive I/O responses from redundant I/O devices (e.g., mirrored disks). The packets transmitted through SAN are protected by CRC (cyclic redundancy check), and disk data integrity is also protected by end-to-end checksums. Because the PEs inside a logical processor are not strictly lock-stepped, they can observe an interrupt when they execute different instructions and thus drift away, producing inconsistent output data that causes comparison failures at the LSUs. The PEs must be synchronised so that the interrupt handlers are invoked exactly at the same instruction on all PEs. A rendezvous protocol and the voters in the LSUs work together to achieve the goal: (1) When an interrupt is received by the PEs, each PE initiates a synchronisation and proposes its VRO (voluntary rendezvous opportunity) number to handle the interrupt by writing to the special rendezvous register in its LRU. The VRO is a small code section embedded in the OS and called implicitly by applications. (2) The proposed VRO numbers are reflected to all PEs, and the highest VRO number is chosen. (3) When each PE arrives the agreed VRO number, it calls the interrupt handler. The VRO embeds a small portion of code that increaes the VRO number and checks for pending interrupts, and it is inserted into the code streams and meant to be executed periodically. To cope with uncooperative processes that do not execute VRO for an extended period, two algorithms, the UNCP-Store and UNCP-Trace, are developed to bring the uncooperative processes running on different PEs to the same instruction before suspending the processes and executing VRO. The UNCP-Store chooses one PE as the target and copies memory stores and registers of the target PE to other PEs. The UNCP-Trace figures out which PE is leading and determines how many instructions the trailing PEs need to catch up. The sophisticated operating system, NonStop Kernel, is capable of recovering from various software or hardware failures. For instance, critical system services are implemented in primary-backup process pairs.

This approach demonstrates how processors can be loosely lockstepped (i.e., the signals the processors are not checked cycle by cycle) by the kernel with help from LSUs and invocations of VRO. We also aim to execute a whole system redundantly on different CPU cores, but all the cores belong to the same processor. The issue of cores diverging because of imprecise interrupt handling needs to be tackled in our approach as well. The synchronisation protocol Section 4.3 we developed for *redundant co-execution* largely resembles functionality of the rendezvous protocol and the UNCP-Trace algorithm in NSAA.

3.1.3 Summary

The approaches employing hardware or system redundancy are able to tolerate most transient hardware faults without service interruptions. However, the initial purchase and subsequent operating costs restrict them to several specialised segments requiring extreme uptime, such as banks, stock exchanges, or telecommunication providers. In addition to the costs, the increased size, weight, and power consumption of these approaches also make them unsuitable for embedded systems (e.g., flight computers for satellites and geographic information systems for vehicles) that have limited budgets in physical volume and are usually powered by batteries. Compared with software-implemented approaches, hardware redundancy lacks the flexibility of allowing end users to trade the fault coverage for better performance or vice versa. Furthermore, software approaches usually do not rely on special hardware features so that they may be ported to different architectures or platforms. On the other hand, relying on hardware fault-tolerant features usually means vendor lock-in.

3.2 Software Solutions

Since we build systems on COTS (commercial of the shelf) hardware that usually lacks advanced error-detection or fault-tolerant features (for instance, the caches of some desktop processors are not protected by ECC), we shift our focus to software-implemented error-detection approaches. Replication and redundant execution are the common techniques used by software-implemented fault-tolerant approaches. The granularity of replication can be generally categorised into instruction level, thread level, process level, and whole-system level; furthermore, the replication mechanisms can be implemented in compilers, libraries, operating system services, OS kernels, or hypervisors. The mechanisms implemented in different levels have contrasting assumptions, error coverage, error detection latencies, run-time overhead, and applicability. We can also observe a trend that the replication granularity is becoming coarser to utilise the hardware redundancy provided by multicore processors.

3.2.1 Instruction-Level Replication by Compiler Techniques

SWIFT is a compiler-based solution targeted for Itanium 2 processors; the modified compiler duplicates instructions to compute results with different registers and inserts instructions that compare the results to detect transient hardware faults [Reis et al., 2005]. It assumes memory and caches are protected by ECC so that store instructions are not replicated. In terms of SoR, general computation and flow-control instructions are within the SoR; but the inserted checking instructions, store instructions, caches, and memory are excluded from the SoR. Input replication is achieved by duplicating load instructions, and output comparison is realised by checking store addresses and output register values. There is a gap between the validation of the address and value of a store instruction and the actual execution of the instruction, so an error occurred between the validation and execution can cause an undetected data corruption. A limitation of SWIFT is that the compiler cannot enforce an order of memory accesses for a multi-threaded application or applications communicating through shared memory regions, so only single-threaded applications are supported. For a multithreaded application, a replicated load instruction pair may observe different values and cause comparison failures in absence of transient faults.

Wang et al. [2007] propose to exploit multiple cores provided by chip multi-processors (CMPs). This technique implemented in a research version of the Intel ICC 9.0 automatically generates a pair of threads, a leading thread and a trailing thread, for a thread in source code. The leading thread performs all operations of the original program, and additional instructions are added to communicate with the trailing thread. System calls for I/O operations and shared memory loads are excluded from the SoR because the values returned by the calls or loads are non-deterministic. Thus, the leading thread takes the responsibility of input replication by invoking the calls or performing memory loads and then sending the return values to the trailing thread through a shared memory buffer. The trailing thread, running on a separate core, re-executes the computations of the leading thread and compares its outputs with the outputs received from the leading thread. The comparisons conducted by the trailing thread verify the following output data: the addresses of shared memory load or store operations, the values to be stored into shared memory, and the parameters passed to system calls. The inserted instructions that implement the communication channel between the leading and trailing threads and that conduct output comparisons are not redundantly executed, so they are vulnerable to transient hardware faults.

HAFT (hardware-assisted fault tolerance) [Kuvaiskii et al., 2016] takes a hybrid approach, which combines compiler-based instruction-level replication for error detection and Intel TSX (transactional synchronisation extensions) instructions for error recovery. HAFT is implemented as two LLVM [LLVM, 2016] passes: the ILR (instruction level redundancy) pass replicates instructions and inserts checking instructions for error detection; and the Tx (transactification) pass covers an application with transactions (the number of transactions is determined by a *transactification* algorithm at compile-time) to provide hardware-assisted recovery. The ILR pass essentially creates shadow threads that re-execute instructions (except control-flow and memory-related ones) of original threads with different registers as source and destination operands, and the computation results are checked before being used to update memory locations. If an error is detected by the inserted checking instructions, the current transaction is aborted by issuing a TSX instruction explicitly, and the processor automatically rolls back the processor's architecturally-visible state to the point before the transaction started. By using hardware transactions, HAFT almost gets error recovery for free. If an aborted transaction fails to commit within a bounded number of retries, a non-transactional fallback will be executed instead. If an error hits the fallback, HAFT has no choice but terminates the application. PARSEC 3.0 [Bienia and Li, 2009] and Phoenix 2.0 [Ranger et al., 2007] are used to evaluate the performance overhead of HAFT. Although the averaged runtime overhead is 1.89 times, the overhead for each individual benchmark can be as low as 1.04 times or as high as 4.21 times. Furthermore, enabling hyper-threading can increase or decrease the overhead for each benchmark dramatically: `matrixmul` goes from 1.04 times to 377 times due to cache-unfriendly behaviour, but `vips` goes down from 4.21 times to 1.5 times due to increased instruction-level parallelism. A fault injection tool, based on the Intel SDE emulator [Intel, 2016b], selects an instruction randomly and injects a fault to one of the chosen instruction's output registers arbitrarily to complete one fault injection. The results show that HAFT can detect 98.9% of injected data corruptions and correct 91.2% of the detected corruptions. However, the total number of faults injected is not disclosed, neither is how random numbers are generated. HAFT does require that memory be protected by other measures such as SDC-DED ECC or Chipkill since application data and memory load/store instructions are not duplicated, and it cannot detect the faults happened in-between inserted checking instructions and non-replicated instructions that update memory or change control flow. HAFT does not cover Linux system calls that run in privileged kernel mode, leaving an important part of the system unprotected.

In summary, the three compiler-based approaches share the following commonalities: (1) Memory and caches are assumed to be protected by ECC so that application data is not duplicated; neither are memory store instructions. (2) Recompilation is required, so static or dynamic libraries are not protected unless the libraries are also recompiled. (3) The inserted checking instructions are vulnerable since they are executed only once. (4) There is a window of vulnerability between the successful validation of output data and the actual use of data. (5) System call instructions are not duplicated; instructions executed in kernel mode to serve system calls are completely unprotected. (6) Non-deterministic shared memory loads are not duplicated: a load is executed once, and subsequently the return value is duplicated.

3.2.2 Process-Level Replication by System Services

As shown in Figure 3.3, PLR (process-level redundancy) [Shye et al., 2009] exploits hardware redundancy provided by symmetric multiprocessing (SMP) systems or multicore processors, but it targets unmodified single-threaded binary applications by creating replicas at the process level with Pin (a dynamic binary instrumentation system) [Luk et al., 2005]. When a protected application starts, PLR gains control and starts a monitor process first; the monitor process forks the application twice or three times to create redundant processes that actually perform computations, and the original application process becomes a figurehead process that acts as a Unix

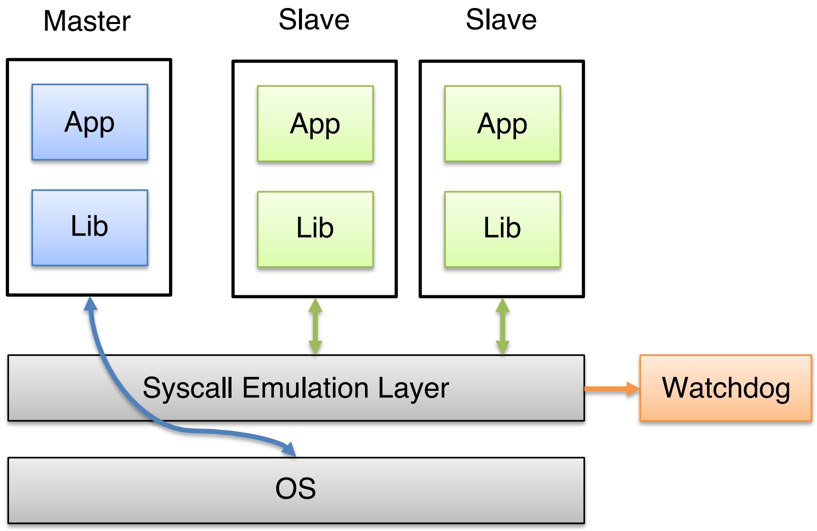

Figure 3.3: Process-level redundancy (PLR)

signal [Stevens and Rago, 2005] forwarder. One of the redundant processes is designated as master process that invokes system calls, others are slave processes that receive system call results from the system call emulation layer that is also responsible for input replication and output comparison. Shared memory accesses are handled by trap-and-emulate techniques. An asynchronous signal is only delivered to the figurehead process, and then the figurehead process stops all redundant processes, sets up the pending signal service epoch number as the highest epoch counter plus one, and resumes the processes. The redundant processes poll for a pending signal at the end of each epoch; if a signal is pending and the current epoch counter matches the epoch number set up by the figurehead process, the processes transfer control to signal handlers. The epoch boundaries can be placed at each system call, each function call, or each backward branch. Nevertheless, PLR assumes that operating systems are protected by other measures; neither the monitor process nor the system call emulation layer is protected by redundant execution. A subset of SPEC2000 is used in fault injection campaigns to evaluate PLR in terms of error detection. Each benchmark is executed 1000 times, and one fault is injected for each run. The victim instruction is randomly selected based on a dynamic instruction execution count profile of the application, and a random bit is chosen from the source or destination general-purpose registers of the selected instruction. A random bit flip is injected by the Pin instrumentation tool during runtime; thus, the fault model is a single-bit flip in general-purpose registers. The authors claim that PLR successfully eliminates all silent data corruptions, and aborts and terminations caused by detected unrecoverable errors. The averaged performance overheads measured using a subset of SPEC2000 compiled with -O2 optimisation level are 16.9% for two redundant processes and 41.1% for three redundant processes.

Romain [Döbel et al., 2012] is an operating system service based on a modern microkernel (Fiasco.OC). Romain consists a master process that initialises environments, creates process replicas, and handles CPU exceptions (e.g., page faults, invalid instructions, or system calls) triggered by the replicas; the exception handling code compares the replicas' states, which include architecture registers, kernel-level exception state, and UTCB (user-level thread control block, a per-thread memory region shared between the kernel and thread) contents, to detect errors caused by hardware faults. The master process also acts as a system call proxy to ensure that a system call is actually invoked only once although the replicas attempt to execute the system call multiple times. Each replica has its own virtual memory space to leverage memory isolation, reducing the possibility of fault propagation from one replica to others; the master process also functions as the memory manager of all the replicas, handling page faults and maintaining consistent memory layouts. The microkernel, OS runtime services, device drivers, and the Romain framework lie outside of the SoR. The term *reliable computing base* (RCB) is used to describe the software components that must function correctly to provide transparent process replication but are outside of the SoR. The RCB must be protected by other measures; for example, the authors propose to protect the RCB by deploying the system on heterogeneous multicore processors which consist of hardened resilient cores (ResCore) and non-resilient cores, and only execute the RCB on the resilient cores for reliable operations [Döbel and Härtig, 2012]. Romain is further improved to support multi-threaded applications by ensuring that the invocations of `pthread_mutex_lock/unlock` functions are executed deterministically [Döbel and Härtig, 2014].

Similar to instruction-replication-based approaches, process-level redundancy provides protections for selected user-mode applications. Once a system switches to kernel mode to handle a system call or an interrupt, redundant execution stops. Therefore, a significant part of the whole system is left unprotected. For example, if the kernel responds to a system call with incorrect data, the user-mode applications may not be able to detect such errors. A common trait shared by these approaches is that the runtime software layers responsible for input replication, output comparison, and replica management are not redundantly executed and verified. The trait implies the singe-bit-flip fault model: only one bit flip throughout the execution of an application [Reis et al., 2005]. This model assumes that a bit flip changes either the result of a normal instruction or the result of an error-checking instruction, but not both. If the normal instruction is impacted, the error-checking instruction should be able to detect the error; if the error-checking instruction is affected, a false positive triggers an unnecessary recovery, but the result is still correct. The input replication instructions can also be affected; in this case, the approaches rely on the output comparison to catch the divergence of the replicas. However, if corrupted input data is used by all replicas, the error cannot be detected by the output comparison—application-specific error-detection algorithms should be employed.

Device drivers are indispensable for building practical and useful systems, but the code of device drivers tends to have a higher error rate than other kernel subsystems [Chou et al., 2001]. Buggy device drivers impact the dependability of operating systems. Minix 3 aims to increase system availability by isolating device drivers as user-mode processes and introducing mechanisms that detect and repair failures of device drivers [Herder, 2010]. The system is designed to tolerate intermittent and transient driver failures, and the observation is that microrebooting can correct a large portion of driver failures. Running device drivers as user-mode processes employs the isolation provided by MMU to prevent a driver failure from crashing or corrupting the entire kernel. Static per-driver isolation policies restrict the resources that a user-mode driver can access, and I/O MMU ensures that each driver can only issue DMA (direct memory access) requests targeting the driver's own physical memory regions. A reincarnation server monitors the status of device drivers and detects driver defects by the following measures: process exit or panic, CPU exceptions, user-initiated termination, loss of heartbeat messages, complaints by other components, and dynamic update by users. The recovery procedure is initiated by the reincarnation server once a defect is detected, and the procedure can be a simple restart or a policy script that contains precise steps to repair the defect.

3.2.3 Virtual-Machine Replication by Hypervisors

The virtualization technologies extend the replication boundary to include operating systems, making system-level redundant execution possible. A hypervisor is software that creates and manages virtual machines (VMs) so that multiple operating systems are able to share the computing resources provided by a single physical host machine; the VMs are isolated from each other by the hypervisor. Since the hypervisor has full control over the VMs, it can pause a VM, and inspect or modify the VM's internal data at its will.

Bressoud and Schneider [1996] design the protocols that coordinate non-deterministic event delivery for a hypervisor based on HP's PA-RISC architecture so that the hypervisor is capable of managing a primary-backup virtual machine pair for fault tolerance. The protocols divide VM execution into epochs by programming the recovery register to generate an interrupt after a certain number of instructions have been executed, and the length of an epoch is configurable by varying the number written to the recovery register. A very important feature of the PA-RISC processor is the deterministic deliveries of the interrupts generated by the recovery register, so the primary and backup VMs start and end each epoch at the same instruction precisely. The hypervisor buffers interrupts on the primary VM and sends them to the backup VM, and deliveries of the interrupts on both VMs only happen on epoch boundaries. Therefore, the epoch length determines the synchronisation frequency as well as the interrupt delivery latency. For the instructions that interact with I/O devices, the hypervisor allows the primary VM to execute the instructions; and the return values of the instructions are transferred to the backup VM as well, ensuring consistent input data is observed by both virtual machines. The performance evaluation section reveals how the epoch length affects CPU-intensive and I/O-intensive applications. For CPU-intensive workload, Dhrystone 2.1 is used and executed 1 million iterations per experiment. The averaged execution time of 20 experiments running on the fault-tolerant VM pair is 6.5 times of the baseline execution time when the epoch length is 4000 instructions, but the predicated best case is 1.24 times if the length is 385,000 instructions, at the expense of increased interrupt latency. Our closely-coupled redundant co-execution in Section 4.4 also ensures that external device interrupts are observed consistently by replicas with assistance from hardware performance counters or a compiler plugin; but our approach does not divide the execution of VMs or applications into epochs, nor does it require a constant distance be maintained between the replicas. The way of this approach treating I/O instructions is also similar to our I/O device access patterns that will be described in Section 5.2.

Remus [Cully et al., 2008] aims for high availability by replicating *protected* and *backup* virtual machines on a pair of physical hosts. The approach enhances the live migration capability of the Xen virtual machine monitor to support fine-grained checkpoints. Instead of delivering non-deterministic events to the protected and backup virtual machines, Remus periodically checkpoints the state of the protected VM, transfers the checkpoints to the backup VM, and synchronises the backup VM state by applying the checkpoints. Network out-bound data generated by the protected VM is buffered and can only be released when state synchronisation finishes on the backup VM (i.e., the protected VM receives acknowledges of completed checkpoints from the backup VM). The error detection mechanism is simply based on detecting timeouts: either the backup VM fails to respond commit requests or the protected VM stops sending checkpoints. The states of protected and backup VMs are not checked for data integrity or control flow correctness: the backup VM is a copy of the protected VM so that it can be used for resuming operations if necessary. Performance overhead is proportional to the checkpoint frequency; the measured Linux kernel compilation overheads, in terms of increased wall-clock time, are 31%, 52%, 80%, and 103% for checkpoint rates at 10, 20, 30, and 40 times per second respectively.

Scales et al. [2010] bring fault-tolerant features to commercial enterprise-grade virtualization product—VMWare vSphere 4.0. Fault tolerance is provided by running the primary and backup virtual machines in *virtual lockstep* on different physical machines. The term virtual lockstep is coined to describe the fact that the hypervisors manage the virtual CPU of the backup VM to execute the same instructions committed by the primary VM. A logging channel is used to transmit input data and non-deterministic events captured by the primary VM to the backup VM, which applies the data and replays the events deterministically. VMWare vSphere 6.0 [VMware, 2015] adopts a new approach called *fast checkpointing* to support up to 4 virtual CPUs in a virtual machine. Fast checkpointing continuously captures the active memory and precise execution state of the primary VM and transfers the data to the backup VM over a high-speed, dedicated 10 Gbps network. Fast checkpointing is used to avoid recording non-deterministic events introduced by concurrent memory accesses from multiple CPUs; this type of non-determinism does not exist in a single-core system, and the overhead to record and replay such events is significant without hardware support.

Overall, the VM-replication-based fault-tolerant approaches focus on tolerating machine failures, aiming for uninterrupted service. Two general techniques, record-replay and checkpointing, are used to ensure that primary and backup VMs are in consistent states. Neither of the techniques considers the effects of transient faults. For example, a bit flip occurred in the memory of the primary VM can be checkpointed and transferred to the backup VM; so both VMs may suffer from the same error and fail in the same way. Another weakness is that the outputs of the primary and backup VMs are not compared for consistency, so the primary VM affected by an SEU may release erroneous outputs.

3.2.4 Discussion

Table 3.1 lists representative software-implemented fault-tolerant approaches. Although the implementation details, assumptions, and levels of replications vary greatly, they share a common prerequisite: the underlying system software, be it an OS kernel or a hypervisor, is either assumed to be fault-free or be protected by other measures. The system software is usually the critical component to enforce system security policies and ensure data integrity. Kernels, hypervisors, and device drivers represent a significant portion of a whole software system, providing runtime environments for applications. Leaving these components unprotected would undermine the effort of defending the applications from transient faults. Moreover, the system software is shared by all applications, so bit flips in kernels, hypervisors, or device drivers impact the whole-system reliability, security, and integrity: a transient-fault-induced kernel vulnerability can cause much more serious damages than a faulty application since the kernel has full control over the whole system and is running at the highest privilege level. Most importantly, the existing software approaches described above aim for fault tolerance, neglecting the security issues led by hardware faults. For instance, none of the approaches checks the integrity of page tables that are used to translate virtual memory addresses to physical memory addresses by processors; but these page tables are vital for security since the isolation between applications relies on the correctness of the page tables. Our proposed approach can be configured to include operations on the page tables into execution fingerprints (details in Section 6.2) so that the kernel replicas can compare the fingerprints to validate the correctness of the operations. Furthermore, the page tables of the replicas exist in different memory areas so that they can be validated entry by entry if necessary.

Name	Replication Level	Notes
SWIFT	Instruction	is based on a compiler, assumes ECC memory, and supports only single-threaded applications.
Redundant MT	Thread	is based on a compiler and hardware multi-threading, and assumes ECC memory.
HAFT	Instruction	is based on a compiler and hardware TSX, and assumes ECC memory.
PLR	Process	is based on a binary instrumentation tool (Pin).
ROMAIN	Process	is implemented as OS Services.
VM PA-RISC	VM	is implemented in the hypervisor with record-replay.
Remus	VM	is implemented in the hypervisor wich check-pointing.
vSphere 4	VM	is implemented in the hypervisor with record-replay.
vSphere 6	VM	is implemnted in the hypervisor with fast check-pointing.

Table 3.1: A summary of software-implemented fault-tolerant approaches

3.3 Deterministic Execution

Deterministic systems produce consistent outputs across multiple runs given that the inputs are the same for each run. This attribute makes these systems very useful for debugging and replication-based fault tolerance. In this section, we only focus on software-implemented approaches to deterministic execution since building systems on COTS hardware rules out customised hardware components, such as FDR (flight data recorder) which uses modified directory-based cache hardware to record data races [Xu et al., 2003].

The meaning of deterministic execution in the context of our SMR-based approach is different from the traditional definition of deterministic execution summarised by Bergan et al. [2011]. We aim to ensure that the concurrently running replicas of an application produce the same outputs in a run, given that the inputs are the same for all the replicas. However, the outputs produced by the replicas in different runs can be different even if the same inputs are used. Thus, we use the term *co-execution* to denote our intended execution mode.

Olszewski et al. [2009] roughly classify the systems into two categories based on the degree of determinism that the systems can provide: (1) *strong determinism* that ensures memory access ordering of a multi-threaded application is deterministic, and (2) *weak determinism* that enforces deterministic locking ordering for a multi-threaded application. In the case of strong determinism, data races (two or more threads in a process access the same memory location concurrently without using synchronisation primitives, and at least one of the accesses is a write) can be tolerated and made deterministic; however, weak determinism requires data-race-free applications (i.e., shared data is protected by synchronisation primitives).

Kendo [Olszewski et al., 2009] builds a logical clock by counting retired memory-store instructions with hardware performance counters of x86 processors since the authors find that the event is deterministic on the Intel Core 2 models. The performance counters are programmed to generate an interrupt after N (an adjustable chunk size) memory-store instructions retired, and the interrupt handler updates each thread's logical clock accordingly. Given that each thread's logical clock increases deterministically and is independent of physical time, the locking ordering can be enforced by only allowing the thread with the least logical clock value to grab a lock when multiple threads are competing. When several threads have the same logical clock value, the thread identifiers are used as a tie breaker. Kendo guarantees weak determinism by enforcing locking orders. Kendo inspired us to construct a logical clock to coordinate the running replicas of a multi-threaded application. In Section 4.4, we present how we employ hardware performance counters on x86 processors and compiler-implemented counters on ARM processors as logical clocks to achieve precise preemption that is a prerequisite (otherwise replicas may diverge because of the different memory access orders) for replicating multi-threaded applications.

DTHREADS [Liu et al., 2011], designed as a POSIX threads [IEEE and Group] drop-in replacement, provides strong determinism by isolating "threads" with private memory regions, updating shared data transactionally at synchronisation points, and enforcing locking orders. Creating and terminating threads, acquiring and releasing locks, waiting and signalling conditional variables, blocking on barriers, and several selected systems calls are defined as synchronisation points. When creating a thread, the library uses the Linux system call `clone` internally to initialise a process instead, and initially the global data and heap regions are marked as read-only. Each thread's memory updates to the read-only regions are trapped, and private pages are allocated to store the changes. The private updates from all threads are committed later at the synchronisation points in a deterministic order, so data races can be resolved. Between two synchronisation points, threads are executing in parallel and accessing private memory regions. At the synchronisation points, a global token pointing to the next thread in the run queue is used to serialise threads: each thread must wait for its turn before committing private memory copy to the global memory regions (the last writer wins when multiple threads updated the same memory address), performing synchronisation primitives, and passing the token to the next thread in the queue. The threads finished their turns wait on an internal barrier; when all threads finish their turns, they pass the barrier and start next parallel phase. The positions of the threads in the token queue determine the order of getting turns, and the positions can be arranged the same across multiple runs, achieving deterministic synchronisation primitives that do not rely on particular hardware features. DTHEADS still uses POSIX threads internally as building blocks. DTHREAD presents a straightforward approach to building a library that provides strong determinism without relying on hardware features. As we will describe in Section 4.5.1, the loosely-coupled redundant co-execution only supports multi-threaded applications that are data-race-free. The techniques described in DTHREAD can be retrofitted to the seL4 environment so that applications with data races can also be supported.

3.4 Summary

Various degrees and forms of redundancy are the key elements in hardware, software, or combined fault-tolerant approaches. Hardware solutions extensively adopt spatial redundancy and coding techniques: SEC-DED ECC and advanced variants are used to protect caches, register files, and DRAM. Parity, SEC-DED ECC, residue checking, redundant execution safeguard the logic components in processors. Hardware approaches require little intervention from software layers, and the performance degradation is less severe than software-based counterparts. However, such hardened hardware targets high-end server market so that the increased space, power consumption, and costs are less concerned. For instance, most of the everyday desktops, laptops, and mobile devices that count for the majority IT (information technology) equipment sales are not even equipped with SEC-DED ECC memory.

Software-implemented fault tolerance (SIFT) is flexible, and it is possible to apply an approach to different CPU architectures. As multicore processors have become ubiquitous, SIFT approaches also shift from fine-grained instruction-level duplication to coarse-grained thread, process, or virtual machine level replication, better exploiting the hardware redundancy of multicore processors with simultaneous multithreading. The compiler-based techniques duplicate instructions and insert additional checking instructions, delegating memory protection to ECC. The coarse-grained replication-based approaches, the unprotected system software layers (kernels, runtime libraries, hypervisors, etc.) perform replications and error detection and masking. These software-based approaches presume the lowest-level system software is correct and protected by alternative measures. The assumption may be justifiable for systems that availability is the only concern, but we believe that a self-protecting kernel, being the lowest-level system software, is vital for security-critical systems built based on COTS hardware, which suffers from the issues caused by transient hardware faults.

Chapter 4

Whole-System Redundant Co-Execution

> Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.
>
> Edsger W. Dijkstra

This chapter presents the framework for achieving *redundant co-execution (cooperative and concurrent execution)* on multicore platforms, providing a solid foundation and backbone for building error detection mechanisms. We use the term *co-execution* to highlight the fact that the execution of replicas of a whole software system is *coordinated* by a synchronisation protocol and *concurrent* on different cores. We begin with the architecture overview that reiterates the need to protect the lowest-level system software (specifically, the seL4 microkernel), present the synopsis of our approach to improving the microkernel with self-checking capabilities by applying state machine replication to all layers of the software stack, and examine the challenges. After the overview, we explain how we replicate a whole software system and ensure that the replicas have identical initial states. Then, we introduce a synchronisation protocol to coordinate redundant co-execution. There are two implementations of the protocol: the first variant, called *closely-coupled redundant co-execution* (CC-RCoE Section 4.4), takes advantages of hardware performance counters available on the modern x86 architecture or utilises a GCC compiler plugin on the ARM architecture to achieve precise preemption; the second variant, called *loosely-coupled redundant co-execution* (LC-RCoE Section 4.5), does not rely on hardware features or the GCC plugin but requires that multi-threaded applications must be free from data races.

The modifications and newly introduced mechanisms to the seL4 kernel are highlighted in each section. Currently, our implementation is based on the seL4 microkernel; but we believe that other microkernels may face similar challenges when applying SMR to a whole software system. Thus, we strive to generalise our design so that others can also benefit from our work.

4.1 Architecture Overview

First, we give an informal definition of redundant co-execution. Then, we delineate how SMR (state machine replication in Section 2.3.2) is applied to a microkernel-based system, aiming for redundant execution of the whole system. Following that, we employ the SoR (sphere of replication in Section 2.3.1) to analyse the resulting system and map the components in the system to the logic counterparts in the SoR. Based on the determinism analysis of an seL4-based system (Section 2.4.5), challenges of replicating a complete system that includes the microkernel, applications, and device drivers are examined at the end of this section.

4.1.1 Redundant Co-Execution

We use the term redundant co-execution (RCoE) to describe an execution model in which multiple replicas of a software system are running *concurrently* and *independently* on different CPU cores unless the replicas need to synchronise for consistently observing input data from I/O devices, comparing I/O outputs, and handling I/O device interrupts. Most components of the system, including the system software, are redundantly executed, and the synchronisation of the replicas is coordinated by a protocol.

We briefly compare our model with related redundant execution models in the literature. The leader-trailer model used in SRT [Reinhardt and Mukherjee, 2000], CRT [Mukherjee et al., 2002], and SRMT [Wang et al., 2007], designates a replica as the leader and maintains a distance between the leader and the trailer. Note that the Delta-4 XPA architecture pioneered the leader/follower model [Barret et al., 1990] in the context of building an open, fault-tolerant, and distributed computing architecture. The leader-trailer model applies the ideas of the leader/follower model in different contexts. The leading replica performs memory loads and feeds the load values to the trailer, since the leader and trailer share the same memory. In the RCoE model, uncoupling the replicas is made possible by the fact that each replica has private memory regions so that the corresponding load and store instructions to private memory regions can be executed independently by each replica. Also, the execution speed of each replica is not throttled unless a round of synchronisation is in progress. In Romain's transparent redundant multithreading model [Döbel et al., 2012], each replica of a user-mode application has its address space. However, the replicas are synchronised for each externalisation event (such as a system call or a page fault): the replicas are blocked on an externalisation event, and controls are transferred to the Romain master process that handles the event and compares the architectural states of the replicas. Note that the Romain master is assumed to run on reliable hardware so that the master is not affected by hardware faults. The model of PLR [Shye et al., 2009] also blocks replicas of a process at the system calls so that a designated master process is able to execute a system call on behalf of the replicated processes and thus returns the call results to the slave processes. In both models above, system calls are essentially synchronisation points. In the RCoE model, replicas do not synchronise at system calls (except for the system calls explicitly requiring synchronisations of the replicas or the systems configured to compare every system call), reducing the runtime overhead. Furthermore, system calls are handled locally by each replica due to the fact that the kernel and device drivers are also replicated; thus, in our model, there are no special, unchecked master processes, which can become a single point of failure.

4.1.2 Applying SMR to an seL4-Based Software System

The need for replicating a whole software system

As a microkernel, seL4 follows the minimality principle [Liedtke, 1995]:

> A concept is tolerated inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

The kernel designers strive to provide general microkernel mechanisms for building almost arbitrary systems. Being a policy-free kernel means that delegating policy-making decisions to user-mode applications is the favoured approach enforced by design. The approach alleviates the complexity of kernel design and implementation, but the implication is that the states of policy-making applications are also vital. For example, the distribution of capabilities (see Section 2.4.1) is entirely managed by user-mode applications so that an error affecting the applications may cause a violation of resource allocation policies. Another example is the so-called root task, which is the first user-mode application brought up by the kernel. The root task has all hardware resources that are not reserved by the kernel and is responsible for initialising device drivers, system services, and applications. Should an error affect the root task, the whole system state could be invalid. User-mode device drivers, of course, should be protected since they deal with incoming and outgoing data directly. Also, they are the last line of defence against silent data corruptions, so we should check the integrity of the outgoing data in the drivers.

As the policy enforcer, the seL4 microkernel must perform correct operations instructed by user-mode policy-making applications. For example, as we described in Section 2.4, the capability spaces and virtual memory spaces are in-kernel data structures that are vital for managing the authority distribution and maintaining the isolation between any two applications; the microkernel directly manipulates the spaces based on applications' directions. Obviously, if the kernel operations on these data structures are disturbed by transient hardware faults, the corrupted data in the spaces can cause serious violations of isolation and security properties. For example, the IA32 page table entry, as shown in Table 4.1, uses a single bit `R` to represent if a 4 KiB memory frame is read-only or read-write; so a single bit flip can change the access permission. Google Project Zero team demonstrated the attack that exploits DRAM rowhammer bug [Rowhammer] and triggers single-bit flips in page tables can be leveraged to gain unrestricted access to all physical memory [Seaborn and Dullien, 2015].

31-12	11-9									0
Physical Page Address	Avail.	G	0	D	A	C	W	U	R	P

Table 4.1: The format of an IA-32 page table entry

In summary, in order to protect an seL4-based software system with redundant co-execution, all components should be replicated, redundantly co-executed, and subsequently checked to ensure the proper initialisation of the system, to guarantee the correct distribution of capabilities, to uphold data integrity, and to scrutinise communications between applications. The approach is a natural decision based on the fact that many kernel-mode functionalities of monolithic kernels are moved to user mode in microkernel-based systems

An example of a replicated system

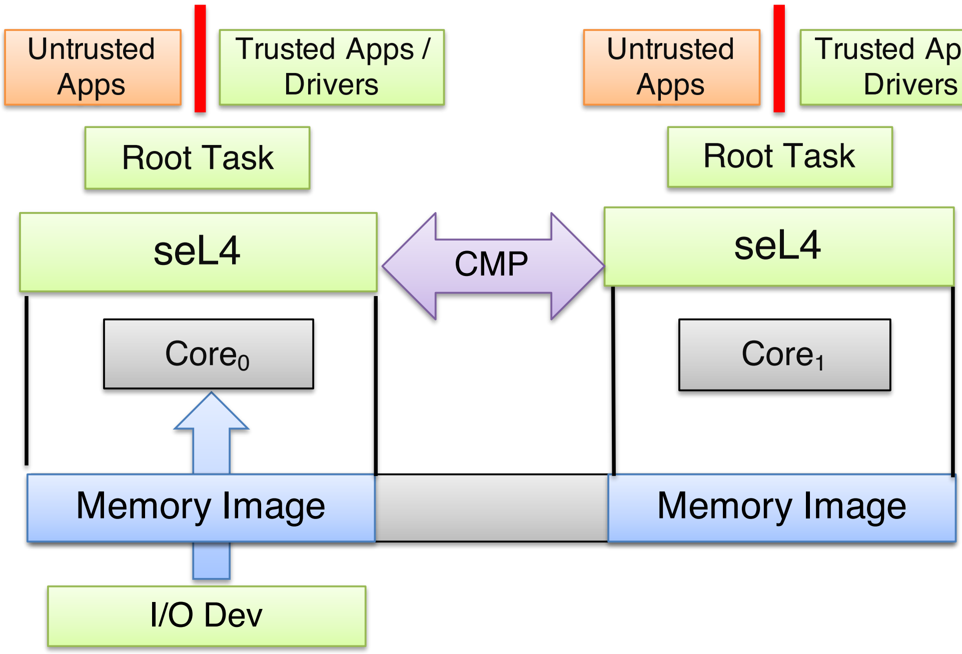

Figure 4.1: Replicate an seL4-based system

An exemplary replicated system is presented in Figure 4.1, depicting how physical memory and CPU cores are split and assigned to the replicas. Each replica includes an seL4 kernel instance and applications. In the figure, the green box labelled “Trusted Apps / Drivers” represents applications that can be trusted, and the orange “Untrusted Apps” box represents untrusted applications that can potentially affect the trusted applications. The vertical red bars represent the isolation that must be maintained by the microkernel. The figure also shows an important trait of our replication approach: the seL4 kernel, being the lowest level system software, is also instantiated on different cores; most of the kernel functions are executed redundantly on different cores, so are the applications. The purple double arrow indicates the redundant comparisons that validate the states of the replicas for error detection.

We treat each replica as a state machine, and thus we can observe that the inputs for the state machines (replicas) are from the I/O devices. Admittedly, the initial states of the replicas are not strictly identical. For instance, the physical memory regions allocated to the replicas are not the same. We need to ensure that the state transitions of the replicas are independent of the different initial states. In the figure, the replica running on $Core_0$ manages I/O devices and receives interrupts from the devices; so we call this replica as the *primary replica* (Section 5.2). The replica running on $Core_1$ does not have direct accesses to the I/O devices, nor can it receive the I/O device interrupts. Therefore, the observations of non-deterministic events from I/O devices must be coordinated so that the replicas observe and handle the events consistently (see Chapter 5).

4.1.3 SoR Analysis

In our approach, each replica includes an seL4 kernel instance, applications, libraries, and device drivers; and the hardware components for each replica comprise a CPU core and physical memory regions allocated to the replica. However, the I/O devices are not redundant, so they are the "rest of the system". Non-redundant I/O devices only communicate with the replica that is designated as the primary replica. Since user-mode device drivers and the kernel interact with I/O devices, the functions of input replication and output comparison are implemented in the drivers and the kernel. The output comparisons are conducted redundantly on each replica; the checked output data is finally committed to the I/O devices by the primary replica. Thus, a window of vulnerability exists between the successful comparisons and the actual uses of output data. The output comparison is effective to protect the integrity of output data, but it is inefficient to guard the system security because many important state updates of the replicas do not change the output data to I/O devices. For example, a bit flip in a page table may not be detected by output comparison if the bit flip does not affect output data. However, the bit flip presents a potential security vulnerability.

In the thesis, we mainly focus on detecting and tolerating transient CPU or memory faults, so we do not explore the model of using redundant I/O devices to tolerate I/O device errors. However, we estimate that supporting redundant I/O devices does not require significant changes to our approach since the drivers for the redundant I/O devices are replicated in the same way we replicate drivers for non-redundant I/O devices. Thus, we mainly examine the mechanisms of supporting whole system replication despite that I/O hardware is not replicated. Of course, properly adopting redundant I/O devices will improve system reliability, and we leave it as a future work (Section 8.2.8).

	WS-RCoE	CRT	NSAA	SWIFT	ROMAIN
In SoR	Cores, Mem, seL4, Apps, Drivers	Cores, Code	All	Selected Apps	Apps, Data Mem, Cores
Out of SoR	I/O	Mem, I/O	nil	OS, Cores, I/O, Mem	Kernel, Drivers, Code Mem, I/O
Input Rep	I/O Inputs	Mem Loads by LVQ	I/O Inputs	Mem Loads	Syscall Results
Output Cmp	I/O Outputs	Mem Stores	Outputs of PEs	Mem Stores	Syscall, User Registers
Checker	seL4, Drivers (redundant checking)	The trailing thread	Self-checked LSUs	Inserted comparison instructions	Romain master
Hardware Requirements	Multicore processors	Hardware implementation	LSUs, redundant hardware	ECC memory	Hardened resilient cores

Table 4.2: SoR comparison

In Table 4.2, we compare the SoR of our proposed WS-RCoE (whole-system redundant co-execution) with those of CRT (chip-level redundant threading) [Mukherjee et al., 2002] in Section 3.1.1, NSAA (non-stop advanced architecture) [Bernick et al., 2005] in Section 3.1.2, SWIFT (software implemented fault tolerance) [Reis et al., 2005] in Section 3.2.1, and Romain [Döbel et al., 2012] in Section 3.2.2. We recognise that the SoR of our approach is very close to the SoR of NSAA, which, not surprisingly, provides the highest level of reliability among the approaches. The "Code" in the CRT column indicates that instructions are automatically replicated by hardware and executed on different CPU cores.

The level of replication determines the input replication and output comparison. At the instruction level, memory load instructions provide data to be consumed by the cores, and memory store instructions output the results. At the process level, system calls are the main means used by processes to communicate with the environment. Thus, it is natural to compare system call parameters for error detection and to replicate results of system calls to all process replicas. At the system level, external events from I/O devices are the inputs to the systems, and outputs produced by the systems must be checked.

The checkers used for output comparison must be reliable so that they do not fail to catch errors. In the WS-RCoE, we design the checker in a ways that it is also replicated and redundantly executed so that an error will not be missed if as least one checker replica works. The NSAA uses self-checked LSUs to ensure the functional correctness of the checkers. The checkers used by the other three approaches are neither self-checked nor protected by redundancy. Lastly, we list the hardware requirements of the approaches. Our WS-RCoE requires multicore processors that are ubiquitous nowadays. CRT is a hardware implementation. NSAA employs customised LSUs, each of which includes an FPGA (field-programmable gate array) voter and an ASIC (application-specific integrated circuit) ServerNet SAN (system area network) interface, to connect redundant SMP processors belonging to different servers. SWIFT assumes ECC-protected memory, and ROMAIN specifically mentions that hardened resilient cores should be used to protect the software components that must always function correctly. Among the software approaches, our WS-RCoE includes more components in the sphere of replication and imposes little restriction on the selection of hardware components.

4.1.4 Fault Model

As described in Section 2.1, single-bit or multi-bit upsets can affect correct operation of digital components; and the results can be benign or catastrophic. Before we describe our approach to the issue, we introduce the failure model of transient hardware faults first. Specifically, we focus on the transient faults affecting processors and memory in the thesis; I/O peripherals (disks, network cards, video cards, etc.) should be protected by other measures.

A commonly used fault model is the *single event upset* (SEU) model. An SEU can be caused by of a single particle striking storage elements directly or a single event transient propagating to storage elements. Unlike the simplified version of the SEU fault model described by Reis et al. [2005], an SEU can be a single-bit upset (SBU) or a multi-cell upset (MCU). If an MCU affects several bits in one data word, a multi-bit upset (MBU) is registered. We also adopt the SEU as our fault model and acknowledge the fact that an SBU or an MCU can occur. The trends of transient faults described in Section 2.1.4 mandate us not to ignore MCUs that are becoming frequent as the feature size keeps shrinking. Thus, we omit various hardware masking effects (Section 2.1.3) and assume that an SEU manifests as a single-bit error, a multi-bit error, or several single-bit errors. An important consequence of including MCUs into the fault model is that some assumptions based on SBUs being the only fault model are not valid. For instance, a replication-based fault-tolerant approach cannot assume that only one replica is affected by a single error.

Note that we assume the errors occurred in one CPU core do not propagate to other cores, and we focus on the hardware components that can be directly observable by software—CPU registers and memory. The components hidden from software (branch prediction units, translation lookup buffers, instruction decoders, micro-operation cache [Solomon et al., 2003], etc.) can indirectly affect the execution of a replica if they are disturbed by an error; however, our software-based error detection mechanisms can only catch the error if (a) the replica diverges from other replicas (see Section 6.3.2 or (b) the execution fingerprint of the faulty replica is different from other replicas' fingerprints (see Section 6.3.1). Thus, the implication is that if an error or multiple errors can affect the replicas exactly the same way (i.e., the execution fingerprints are the same but incorrect), our mechanisms are unable to detect the error(s).

4.1.5 Challenges

Applying SMR to seL4-based systems requires the following additional steps:

(a) Hiding the differences of initial kernel states from user-mode applications.

(b) Coordinating the delivery of interrupts and observation of input data from I/O devices.

(c) Taming non-deterministic events introduced by applications.

Item (a) ensures that applications and their replicas managed by different kernel instances do not diverge by not allowing them to observe the differences. For instance, as shown in Section 4.2.3, the *relative memory addresses* are provided to the replicas instead of absolute physical memory addresses. In other words, we create identical "virtualised" environments on different cores to replicate the applications. Both items (b) and (c) guarantee the order and consistency of observing and handling non-deterministic events so that the replicas perform the same state transitions. From now on, the term applications represents user-mode programs other than the root task and device drivers. We use root task and device drivers directly when we mean them. Other complications, although not directly related to SMR, are also listed below.

- Modifications to applications should be minimal. Ideally, the system should be able to run unmodified binary applications directly. This property is mainly for reducing the efforts to adopt our approach and increasing the range of supported applications.

- Building systems based on COTS hardware means that we cannot rely on specialised hardware that is immune to transient faults. Making no assumptions about the hardware suggests that we must extend the sphere of replication as much possible.

- The performance of the resulting system is usable for real-world applications. To achieve good performance, we need to minimise the overheads of redundant co-execution by decoupling the replicas as much as possible: the unthrottled replicas run independently and only synchronise for non-deterministic events, input replications, and output comparisons.

- The synchronisation protocol described in Section 4.3 plays a vital role in coordinating the execution of the system replicas, so it should be lightweight, correct, and resilient to transient faults. As a performance consideration, we also need low-overhead methods of measuring the progress of the replicas of a system, since the synchronisation protocol frequently invokes these methods.

4.2 System Initialisation and Resource Allocation

The SMR approach requires that the replicas have consistent initial states. The replication process starts from the kernel booting stage, and the kernel needs to replicate itself and a root task onto different cores. The OS loaders, GRUB [Dubbs] on x86 and u-boot [UBoot] on ARM, load the seL4 kernel and the root task to memory from hard disk or network and then pass control to the non-replicated kernel image. It is the responsibility of the kernel to replicate itself and the root task, split and allocate hardware resources, and bring up the root task.

4.2.1 Kernel Replication

The original single-core seL4 has kernel-code and kernel-data regions. To replicate the kernel onto different cores, we split the kernel data region to two regions; so the modified kernel supporting replication has the following three regions.

- Per-core kernel code region (PC KCode): Each kernel replica has its own code region which is copied from the kernel image loaded by the system boot loader. The checksums of the original and copied kernel text regions are compared for correctness. Replicating the kernel code section ensures that a single-bit error does not affect the execution of all kernel replicas in the same way. Furthermore, keeping multiple copies of the code region provides the possibility to recover from an error even if the error affects the code section of a kernel replica.

- Per-core kernel data region (PC KData): The private kernel data regions belonging to different kernel replicas are disjoint; each kernel replica's private variables and stack are stored in its private region.

- Shared kernel data region (S KData): This region is accessible by all kernel replicas. The data structures used by the synchronisation protocol are allocated in this region, so are the variables and buffers for implementing the new system calls provided for driver replication (see Section 5.5). The variables in this region do not have additional copies.

Figure 4.2 shows the physical memory layout before and after the kernel replicates itself and the root task to construct a DMR (dual modular redundancy) system; the grey-shaded box indicates the physical memory region shared by all the kernel replicas. The bootstrapping kernel copies the kernel code section once or twice for DMR or TMR. Then, the bootstrapping kernel instance brings up other cores, and each core (the bootstrapping core included) starts to run kernel initialization code which sets up the per-kernel page tables. When the page tables are populated properly, each core switches to the new page table and then starts to fetch instructions from its own code and

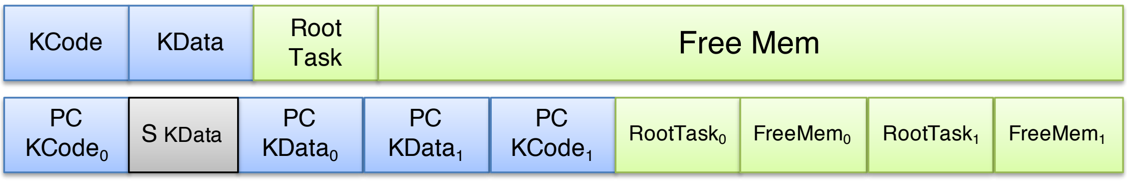

Figure 4.2: Physical memory layout before and after kernel replication for a dual modular redundancy (DMR) system

data sections. All the kernel replicas use a barrier (a synchronisation primitive) to coordinate the dropping to user mode after they finish the per-kernel initialization.

The seL4 kernel replicates itself during starting up so that each kernel replica has its private code, data, and stack. The shared kernel data region is accessible by all kernel replicas.

4.2.2 Root-Task Replication

The root task is copied by the bootstrapping (BSP) core twice or three times for DMR or TMR configurations. The kernel replica running on the BSP core splits physical memory according to the number of replicas before it brings up other cores, and we call the regions as *per-replica free memory regions*. For instance, the *RootTask*$_0$ and *FreeMem*$_0$ in Figure 4.2 occupy the per-replica free memory region assigned to the first replica. The root task is copied multiple times by the BSP kernel to the beginning areas of the per-replica free memory regions, and the information of the root task replicas and free memory regions is saved and passed to other kernel replicas by the BSP kernel. Thus, other kernel replicas are able to initialize the root task replicas and create untyped capabilities from the free memory regions.

Each kernel replica creates untyped kernel objects from the remaining free physical memory region after the kernel finishes allocating the physical memory it needs for initialisation, and the capabilities of these untyped kernel objects are passed to the root task through the *bootinfo* data structure. The physical memory management is completely controlled by user-mode system processes by distributing the capabilities. The bootinfo also contains the physical addresses and sizes of the untyped kernel objects. Although we trust the root task and its replicas will not intentionally abuse the address differences, we prefer to limit the exposure of the state dissimilarities as few as possible. Therefore, we supply all the replicas with the physical addresses and sizes of the primary replica in the bootinfo data structures. Applications usually do not care about absolute physical addresses, but DMA-enabled device drivers need the physical addresses that are passed to I/O devices. The kernel mechanisms to support DMA operations will be presented in Section 5.4 and Section 5.5.

Physical addresses exposed to replicas are "virtualised" so that all the replicas observe the same addresses and sizes as the primary replica's.

4.2.3 Relative Addresses

Although the actual physical addresses are hidden to user-mode, each untyped memory kernel object contains the absolute start kernel virtual memory address and size of the memory region described by the object. The kernel virtual memory address is derived from the start physical address of the region by adding a architecture-specific constant offset. Thus, the corresponding untyped objects belonging to the replicas cannot be compared directly because their start kernel virtual addresses are different. We ought to compare the offsets instead, and we call the offsets as *relative addresses* to avoid confusions. A relative address is calculated by following two steps: (1) If an address is a kernel virtual address, we convert it to a physical address by subtracting a platform-specific constant offset from the kernel virtual address. (2) We subtract the start physical address of the physical memory region assigned to the current replica from the physical address calculated in the previous step to get a relative address. The same rule applies when we check the page table entries of the replicas one by one—we compare the calculated offsets instead of the absolute physical addresses stored in the entries (entries mapping virtual addresses to device MMIO regions are exceptions).

> Relative addresses are compared when validating kernel objects or data structures containing physical addresses or values derived from physical addresses.

4.2.4 Application Replication

User-mode applications are brought up by a root task. The root task can bundle the applications in its data segment as payload, or it can load the applications from disk or network if the drivers are included in the root task. Since the root task replicas already run on different kernel replicas when they are setting up the rest of the system, no special support is required from the kernel to replicate the applications. Each root task retypes the untyped capabilities to create various kernel objects (e.g., TCBs, VSpaces, CSpaces, endpoints, etc). All of the operations are redundantly executed and checked on multiple cores. The root task is not aware of the fact that it is being replicated; and we only modify the part that brings up device drivers (details in Section 5.2). The redundant co-execution framework treats drivers more specially than the general seL4 model does, where the drivers are just applications that happen to have special device memory regions mapped. Actually, we need to modify the drivers so that they can fulfil the responsibilities of input replication and output comparison.

An application instantiated by the root task can create child threads provided that it owns sufficient untyped capabilities to construct various kernel objects required for thread creations. Consequently, the child threads are also replicated naturally on different cores.

4.3 A Synchronisation Protocol for Redundant Co-Execution

One of the meanings of co-execution is that the execution of replicas is coordinated by a synchronisation protocol. Whenever the replicas need to be synchronised to handle a non-deterministic interrupt, the protocol is triggered on all replicas. The protocol has 5 stages as described in Table 4.3, and we use *logical time* to measure the progress of each replica. The precision of the logical time determines the level of consistency we can expect when comparing the replicas in the H stage. Based on how the logical time is constructed (Section 4.4.2, Section 4.4.3, and Section 4.5), two implementations, closely-coupled redundant co-execution and loosely-coupled redundant co-execution, will be described in Section 4.4 and Section 4.5.

We call the replicas *synchronised* when the replicas are ready to observe a non-deterministic event without triggering divergence. As we will see in the following sections, the meanings of synchronised replicas are different for single-threaded applications, multi-threaded applications with or without data races, and applications communicating through shared memory. The closely-coupled version is able to precisely stop all replicas at the same position in the user-mode instruction streams, ensuring that the replicas have consistent states when they handle a non-deterministic event. The loosely-coupled variant brings the replicas close enough—they all have finished the same number of deterministic events that are observable by the kernel—so that the replicas will not diverge if certain conditions are satisfied.

Stage	Description
Initiating (I)	A round of synchronisation is initiated when the primary replica receives an interrupt. A notification is sent to all replicas to indicate the start of the synchronisation.
Proposing (P)	Having received the synchronisation notification, each replica proposes the logical time to handle the synchronisation request based on its current progress. The proposals are stored in a kernel shared memory region so that they can be read by all replicas.
Voting (V)	Each replica reads the proposals from other replicas and chooses the proposal with the highest logical time value. The voting phase runs independently on each replica.
Synchronising (S)	Each replica compares the chosen logical time in the stage V with its proposed time. If the values are the same, the replica waits for other replicas to catch up. Otherwise, it keeps executing until its logical time value is the same as the chosen value and joins the wait afterwards. The synchronising stage finishes when all the replicas join the wait.
Handling (H)	Now, all replicas' logical time values are the same as the chosen value; the replicas can handle the interrupt saved at the stage I and compare the states.

Table 4.3: Five stages of the synchronisation protocol

The five stages are coordinated by a class of kernel barriers, and we name them *S-barriers* (synchronisation barriers). The term *barrier* in this context represents a synchronisation primitive at which any kernel instance cannot proceed until all kernel instances reach the same barrier. For instance, in the S stage, a leading replica blocks at a barrier in order to wait for other replicas to catch up. Furthermore, there is another class of kernel barriers that are used to support device driver replication (Section 5.2) and error detection(Section 6.3.1); we call them *F-barriers* (functional barriers). The two classes of kernel-mode barriers can deadlock a system if the replicas block at different barriers, simply because none of the replica is able to make progress. (Actually, this is an error detection method that we adopt to uncover execution divergence of the replicas. See details in Section 6.3.2.) Since the replicas and barriers can interact in a number of ways that are infeasible for human to explore and verify, the SPIN model serves as a guider when we implement the redundant co-execution frameworks, ensuring that the system will not deadlock during error-free operation.

4.3.1 The SPIN Model of The Synchronisation Protocol

We first describe the abstract SPIN [Holzmann, 2003] model of the protocol, and then we introduce how we map the model to the concrete kernel implementation. The SPIN model is a distilled abstract model of the actual C code. We exclude implementation details for the following reasons: (1) Detailed modelling of system state will lead to the state explosion problem. (2) SPIN does not have direct support for the required operations (e.g., reading performance counters). (3) SPIN does not assume the execution speed of the processes, so the barrier timeout mechanism (Section 6.3.2) is not modelled. Essentially, we use the model to check assertion violations and deadlocks, but not liveness properties. For the purpose of not distracting from our main topic, in the following discussion, we do not explain details of the Promela code, but focus on what the model represents. The correctness of the model can be examined mechanically by the model checker. For the complete SPIN model, please consult Appendix B, which is commented and self-contained.

```
typedef data_t {
    bool        catchup;
    bool        entry_de;
    bool        entry_nde;
    bool        sync;
    bool        sync_req;
    bool        out_flag;
    unsigned    de_type      : DE_TYPE_BITS;
    unsigned    de_count     : DE_LIMIT_BITS;
    unsigned    sync_abort   : N_BITS;
    unsigned    sync_abort_notify : N_BITS;
    unsigned    lead_rep     : N_BITS;
};

/* The per-replica data stored in the shared kernel data region */
data_t data[NKERNEL];
```

Listing 4.1: Per-replica kernel data structure

Each replica's data used during the synchronisation stages is represented by the data_t data structure as in Listing 4.1, and the data[NKERNEL] is accessible by all kernel replicas and is indexed by the replica IDs. The field catchup indicates that the current replica is in the progress of catching up. The fields entry_de and entry_nde specify the replica is processing a deterministic or non-deterministic event. The field sync indicates that if a round of synchronisation is initiated or in progress, and sync_req is set by the primary replica to notify the non-primary replica(s), triggering synchronisations. The type of the deterministic event currently being handled is stored in de_type; de_count is the deterministic event counter that is used as the logical clock to measure the progress of each replica; it is incremented by system calls and application-triggered exceptions. The ID of the leading replica for the current round of synchronisation is in lead_rep. The fields sync_abort and sync_abort_notify are helpers to implement the conditional barrier. The field out_flag helps the implementation of the control flow due to the slightly more restricted control-flow constructs provided by the Promela language, and it does not exist in the C implementation.

The details of deterministic events are abstracted away to reduce the number of states that must be explored and checked by the model checker. As an example, we do not distinguish system calls by call numbers. We generalise the deterministic events into two types: the *normal events* and *synchronous events*. The normal events can be handled independently by each kernel replica. For instance, the system call mapping a physical memory frame to a virtual memory region is considered as a normal event. In contrast, the synchronous events need to synchronise the states of the replicas before they can be handled consistently by all the kernel replicas. For instance, the system call, seL4_FT_Add_Event (see Section 6.2.1), for validating the execution fingerprints (see Section 6.2) of the replicas triggers synchronous events, since the fingerprints of the replicas should only be compared when the replicas have finished the same number of deterministic events. When a kernel replica arrives at a synchronous event first, it blocks on an F-barrier to wait other kernel replicas.

We use a SPIN *process* to model a CPU core executing a system replica, as in Listing 4.2 and Listing 4.3. The system starts from a special process called init which generates a sequence of deterministic events and brings up other non-primary replica(s). Then, init executes the kernel_pri and becomes the primary replica. For more information about the primary and non-primary replicas, please see Section 5.2. For now, we only need to know that the primary replica is designated to receive non-deterministic events and to trigger synchronisations. The kid identifies a replica, and it is also used as an index into the data array in Listing 4.1. In kernel_pri, the do–od repetition construct forms the main body, representing an infinite sequence of input events. The first option, data[kid].entry_de = true, is an assignment so that it is always executable. The second option, data[kid].entry_nde = true, is also executable. For each repetition, one of the two options will be selected non-deterministically. The inline get_de returns the same sequences of deterministic event types (normal or synchronous) to all the replicas. de_handler increases the per-replica event counter (de_count); if the current event is synchronous, the de_handler blocks the replica on an F-barrier used by synchronous events. The inline nd_handler triggers a new round of synchronisation by setting the sync and sync_req to true.

```
inline kernel_pri(kid) {
    unsigned _for_i : N_BITS;
    do
    :: data[kid].entry_de = true -> {
        get_de(kid);
        de_handler(kid);
        data[kid].entry_de = false;
    }
    :: data[kid].entry_nde = true -> {
        nd_handler(kid);
        data[kid].entry_nde = false;
    }
    od
}
init {
    gen_des();
    run kernel_np(1);
#if NKERNEL == 3
    run kernel_np(2);
#endif
    kernel_pri(0);
}
```

Listing 4.2: The primary kernel replica

As to the non-primary replica(s), the main body constitutes two choices: deterministic events (`data[kid].entry_de = true`) and synchronisation requests (`data[kid].sync_req == true`). The first option is the same as the one in `kernel_pri`. The second option becomes executable when the primary replica receives a non-deterministic event and initiates a synchronisation. Again, when both the options are executable, one of them will be chosen non-deterministically. However, the model checker will explore all the possible combinations.

```
proctype kernel_np(unsigned kid : N_BITS) {
    unsigned _for_i : N_BITS;
    do
    :: data[kid].entry_de = true -> {
        get_de(kid);
        de_handler(kid);
        data[kid].entry_de = false;
    }
    :: data[kid].sync_req == true -> {
        handle_action(kid);
        data[kid].sync_req = false;
    }
    od
}
```

Listing 4.3: The non-primary kernel replica(s)

The inline `handle_action` in Listing 4.4 is called by both `de_handler` and `nd_handler` to examine if a synchronisation is requested or in progress by examining the `sync` variable. If the `sync` is true and and the leading replica has not been elected (the else part from line 30), a replica first saves a copy of the variable `sync_abort_notify`, which is used to implement the conditional barriers. Then, if other replicas have made more progress than the replica, it skips the subsequent steps and catches up by setting the `out_flag` variable to `true` (lines 32 to 38). Otherwise, the replica proceeds to the conditional barrier at line 41 and waits for others to arrive. It is worth pointing out that even if the replica passes the check (lines 32 to 38), the replica is not necessarily the leading replica: another replica could have started the next event, but that replica's de_count has not been increased.

When all replicas pass the conditional barrier at line 41 and arrive at line 44, they vote the leading replica by comparing the de_count and store the ID of the leading replica in the lead_rep variables (lines 45 to 51). Eventually, do_sync is called at line 52. The leading replica waits on the barrier sbar at line 6, and the other replicas, which need to catch up, set the catchup variables to true (line 20).

If a replica is catching up, it directly executes do_sync the next time when handle_action is called (line 29). The inline do_sync checks if the replica has finished the same number deterministic events as the leading replicas did (line 3). If the numbers are equal, the chasing replica waits on the barrier sbar at line 6. Eventually, when all the replicas reach the barrier sbar, they have executed the same number of deterministic events and are synchronised. We do not include the actions (e.g., comparing fingerprints, or propagating and injecting interrupts) to be executed when the replicas are synchronised in the model, so the replicas simply reset the states of the replicas and continue execution.

```
1  inline do_sync(kid) {
2    if
3    :: data[kid].de_count == data[data[kid].lead_rep].de_count -> {
4    /* the one with the highest event counter needs to wait others
        to catch up */
5      data[kid].catchup = false;
6      bar(kid, sbar);
7         assert(data[kid].de_count <= DE_LIMIT);
8         assert(data[kid].de_count ==
             data[data[kid].lead_rep].de_count);
9         bar(kid, sbar);
10        /* reset states */
11        data[kid].catchup = false;
12        data[kid].lead_rep = 0;
13        data[kid].sync = false;
14        data[kid].out_flag = false;
15        bar(kid, sbar);
16     }
17     :: else -> {
18        assert(data[kid].de_count <
             data[data[kid].lead_rep].de_count);
19        /* others enter catch-up mode */
20        data[kid].catchup = true;
21     }
22     fi
23 }
24
25 inline handle_action(kid) {
  if
  :: data[kid].sync == true -> {
    if
    :: data[kid].catchup == true -> do_sync(kid);
    :: else -> {
      data[kid].sync_abort = data[kid].sync_abort_notify;
      for (_for_i : 0 .. (NKERNEL - 1)) {
        if
        :: data[kid].de_count < data[_for_i].de_count ->
             data[kid].out_flag = true;
        :: else -> skip;
        fi
      }
      if
      :: data[kid].out_flag == false -> {
        bar_cond(kid, sbar);
        if
        :: sbar.backoff[kid] == true -> sbar.backoff[kid] = false;
        :: else -> {
          for (_for_i : 0 .. (NKERNEL - 1)) {
            if
            :: data[_for_i].de_count >
               data[data[kid].lead_rep].de_count ->
                            data[kid].lead_rep = _for_i;
            :: else -> skip;
            fi
          }
          do_sync(kid);
        }
        fi
      }
      :: else -> data[kid].out_flag = false;
      fi
    }
    fi
  }
  :: else -> skip;
  fi
}
```

Listing 4.4: The synchronisation procedure

In the model, non-deterministic events (e.g., interrupts) can happen any time between any two deterministic events; and each non-deterministic event triggers a synchronisation of the replicas before the replicas are allowed to handle the non-deterministic event. Note that the barrier (`sbar`, an S-barrier, employed in `handle_action` and `sync` is different from the F-barriers (not shown in Listing 4.4) used for synchronous events. Thus, it is possible that the replicas block on different barriers so that the none of them is able to make progress. We introduce the conditional barrier at line 41 above (see Section D.1 for details): a blocking replica is allowed to pass the conditional barrier if certain condition is satisfied. In our case, the condition is that another replica with a higher `de_count` blocks on a barrier for synchronous events. The variable `sbar.backoff[kid]` is set to `true` if the condition meets. We have performed full verification for the DMR (`NKERNEL = 2`) and TMR (`NKERNEL = 3`) versions of the SPIN model, and the results show that our approach of using the S-barriers and F-barriers to synchronise and coordinate the execution of the replicas does not introduce deadlocks.

4.4 Closely-Coupled Redundant Co-Execution (CC-RCoE)

The first implementation of the synchronisation protocol is called closely-coupled redundant co-execution, which measures the progress of each replica by using the branching-instruction counters and the instruction pointer together. CC-RCoE is able to support *precise preemption*: a thread and its replicas running on different cores are preempted only when they have executed the same number of *user-mode* instructions. Note that the instructions executed in kernel mode are not included, since the microkernel replicas are exposed to non-deterministic events and thus may have executed different numbers of kernel-mode instructions. As we will see in Section 5.2, the kernel replica designated as the *primary replica*, which is responsible for receiving hardware interrupts, triggering synchronisations, and performing I/O operations, inherently executes more instructions in kernel mode than the *non-primary replicas*. If kernel-mode instructions were counted, CC-RCoE would fail to synchronise the execution of replicated user-mode threads. This is because the primary replica could execute more kernel-mode instructions and less user-mode instructions than the non-primary replicas, despite that the total numbers of instructions executed by the replicas are the same. By doing so, multi-threaded applications, even with data races or ad-hoc synchronisations, are supported by CC-RCoE. We start from the algorithm for supporting precise preemption, and then, two implementations for the ARM and x86 architectures are examined in detail. Later, we show that the approach can be used to support replicating Linux virtual machines as a case study in Section 7.2; the ability to run unmodified Linux kernel significantly expands the range of applications ready to be deployed.

4.4.1 The Logical Time and Precise Preemption Algorithm

The flowchart for the precise preemption algorithm is shown in Figure 4.3, and the first step of the algorithm is to identify which replica has executed the most instructions so that we can stop it and let other replicas catch up. As shown by Mellor-Crummey and LeBlanc [1989], the number of backward branches taken and the current instruction pointer together are sufficient to identify a unique point in the instruction stream. We will describe how we count branches (including backward branches) by using a per-core performance measurement unit (PMU) on x86 in Section 4.4.2 and by using a GCC plugin on ARM in Section 4.4.3. Let us assume, for the moment, that we can count branches accurately so that we can introduce the algorithm without worrying about the details.

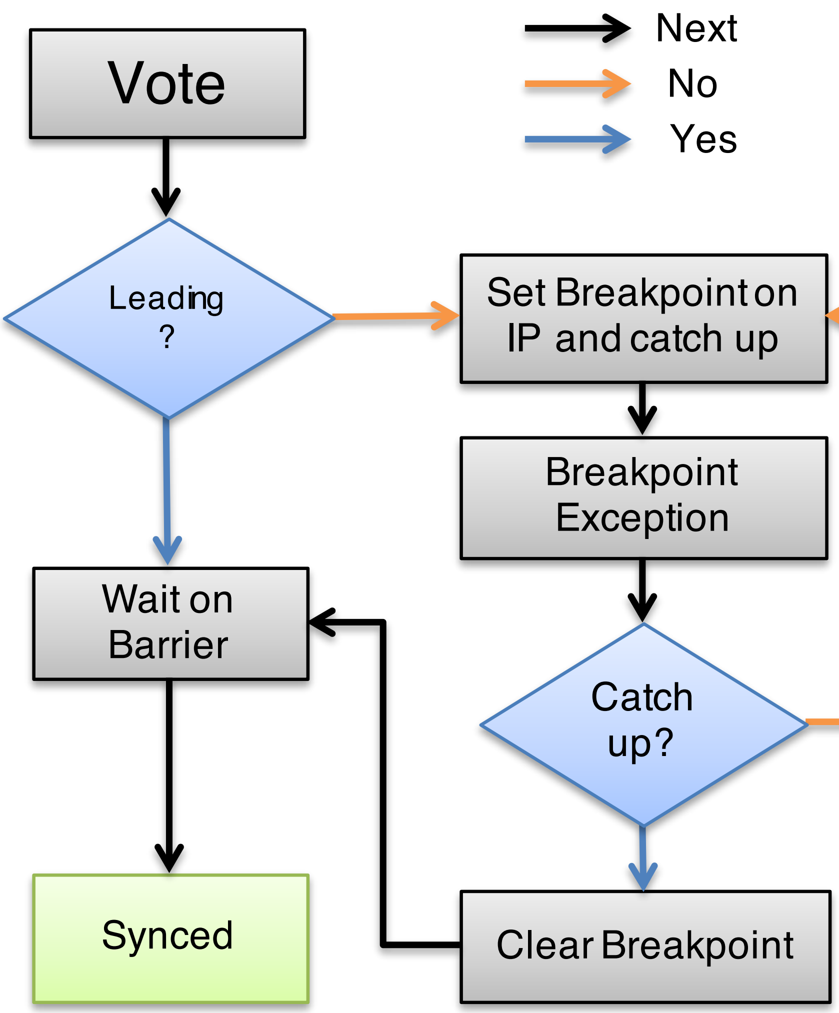

Figure 4.3: The flowchart for precise preemption

In the closely-coupled redundant co-execution approach, each kernel instance uses the triple, (deterministic events, user-mode branches, user-mode instruction pointer), to construct the logical time of each replica.

The triple (`events, branches, IP`) can be used to pinpoint an exact point in the instruction stream. The first item `events` is incremented each time when the kernel handles a deterministic event (system calls, exceptions, etc.). The second item `branches` is the number of branches executed in user mode so far, and the third item `IP` is the instruction pointer of the current thread. The replicas are only considered as consistent when the triples are the same. The item `events` is necessary since system calls and application-triggered exceptions are not included in `branches`. Therefore, an application and its replica can have the same `branches` and `IP` items, but they still can have different `events` items. For instance, a thread traps into kernel mode because of a page fault triggered by an instruction, but the replica of the thread is interrupted by a hardware interrupt at the same instruction (an interrupt has higher priority than a page fault if they are both pending at the instruction boundary). In this scenario, `events` of the former thread is greater than that of the later thread, but their values of `branches` and `IP` are the same. Should we compare `branches` and `IP` only, we would conclude that the replicas are consistent.

At the Vote stage, the leading replica is pinpointed by comparing the triplets. The event counters are compared first. If the counters are the same, we continue to compare branch counters. Again, if the branch counters are equal, we compare instruction pointers. Having identified the leading replica that will wait on a barrier, the algorithm instructs other replicas to program the hardware debug registers to generate exceptions when the values of their instruction pointers are the same as the leading replica's instruction pointer. In the breakpoint handler, a chasing replica checks if its triple is the same as the leading replica's. If they are the same, the replica has finished the same amount of progress as the leading replica did; so the replica clears the breakpoint and joins the barrier. Otherwise, the replica continues execution to make more progress and reprograms the debug registers if necessary. When all the replicas join the barrier, they arrive at the same position in the user-mode instruction streams so that we can preempt them precisely without triggering divergence. Essentially, we implement the synchronisation protocol with fine-grained event counting and stopping replicas at the same instruction by using hardware debug registers. Having given an overview of the algorithm, we will present how the algorithm is implemented for the x86 and ARM architectures.

4.4.2 Hardware-Assisted CC-RCoE

Modern processors provide performance counters that can be programmed to count the the numbers of certain architectural events that have occurred on a core (e.g., instruction retired, branching instruction retired, memory stored retired, etc.). Unfortunately, for the popular architectures, ARM and x86, most of the events are non-deterministic, exhibiting overcounting or undercounting issues [Weaver et al., 2013].

According to the Intel SDM [Int, 2016b], the `BR_INST_RETIRED.ALL_BRANCHES` represents all branch instructions retired and the `BR_INST_RETIRED.FAR_BRANCH` counts all far branches that are triggered by interrupts, exceptions, system calls, etc; the former event includes the latter one. The difference of the two events is the number of user-mode branches executed on each core. We experimentally confirm that the differences between `BR_INST_RETIRED.ALL_BRANCHES` and `BR_INST_RETIRED.FAR_BRANCH` on different cores are the same for Intel Haswell and Skylake microarchitectures. Thus, we can delegate the task of counting user-mode branches to hardware by programming two performance counters. The technical details are described in Section D.3 for the readers who are interested.

4.4.3 Compiler-Assisted CC-RCoE

ARMv7-A processors, specifically the Cortex-A9 processors, cannot provide accurate counters for branching instructions. Slye and Elnozahy [1996] demonstrated the feasibility of using compiler-based branch counting technique to build a record-replay fault-tolerant solution on a DEC Alpha processor with moderate performance overhead. Similarly, we developed a plug-in for GCC to count branches for the ARM architecture. An overview about GCC plugin based on GCC internals [FSF] can be found in Section D.4.

We need to insert an instruction (`insn`) before each call instruction (`call_insn`) and jump instruction (`jump_insn`) to count branches. We decide not to use memory as the storage for counting since we want to minimise the overheads of added instructions. Incrementing a register is probably one of the cheapest instructions since it does not access memory, nor does it significantly complicate pipeline scheduling. (However, there is still cost since reserving a register can affect instruction scheduling of the compiler.) To use a register for counting branches, we need to ensure that the chosen register is not used by any application code. For the GCC compiler, reserving a register can be achieved by the `--ffixed-register` GCC option. On ARM, we adopt register `r9` for this purpose, so the option is `--ffixed-r9`. With the `r9` register reserved, we implement our plugin to iterate lists of `insns`, which are the GCC's internal representation of instructions. If a `call_insn` (a function call instruction) or `jump_insn` (a jump instruction) is found, we create an `insn` that represents the increment operation and insert the `insn` before the `call_insn` or `jump_inst`. The plugin is called after various optimisation passes so that the inserted instructions will not be removed by the optimisations. In Listing 4.5 and Listing 4.6, the original assembly code and the assembly generated when our plugin is activated are listed; `add r9, r9, #1`, is added at line 6, line 10, and line 14 to count the function call instruction (`bl _IO_getc`) and function return instructions (`ldmeqfd sp!, {r3, pc}` and `ldmfd sp!, {r3, pc}`).

```
if_test:
stmfd sp!, {r3, lr}
movw  r3, #:lower16:stdin
movt  r3, #:upper16:stdin
ldr r0, [r3]
bl  _IO_getc
uxtb  r0, r0
cmp r0, #66
ldmeqfd sp!, {r3, pc}
cmp r0, #55
moveq r0, #56
ldmfd sp!, {r3, pc}
```

Listing 4.5: Original assembly code

```
1   if_test:
2   stmfd sp!, {r3, lr}
3   movw  r3, #:lower16:stdin
4   movt  r3, #:upper16:stdin
5   ldr r0, [r3]
6   add r9, r9, #1
7   bl  _IO_getc
8   uxtb  r0, r0
9   cmp r0, #66
10  add r9, r9, #1
11  ldmeqfd sp!, {r3, pc}
12  cmp r0, #55
13  moveq r0, #56
14  add r9, r9, #1
15  ldmfd sp!, {r3, pc}
```

Listing 4.6: Processed assembly code

The plugin is executed by the GCC compiler for each compilation unit, and working at the RTL (register transfer language) level means that all languages supported by GCC can be supported by the plugin. However, all the source code (including libraries) must be recompiled with the compiler with our plugin enabled so the reserved register (`r9`) is not accidentally used. Furthermore, inspections are required for the code in GCC inline assembly or in assembly files, and modifications are made accordingly to exclude the usages of the `r9` register if necessary. Another advantage of transforming code in RTL level is that the plugin can be re-targeted to other architectures easily, thanks to the architecture-independent nature of RTL. Should we need to build a similar plugin for the x86 architecture, we would only need to modify several lines of the plugin code and pick another reserved register for x86.

With the register `r9` as the branch counter, the kernel can read the register and use the preemption algorithm in Figure 4.3. The value in the register `r9` does not need to be passed between threads when context switching for the following reasons: (1) If a context switch is triggered by an interrupt, the kernel replicas need to synchronise the current running threads first so that the values in the `r9` registers are the same. Thus, the implication is that the replicas of the switch-to thread have the same `r9` register values as well. (2) If a context switching is a result of a system call, the system call increments the per-replica event counter so that the kernel replicas can still identify the leading replicas since the deterministic event counters are compared first. Therefore, each thread's branch counter, the `r9` register, stays private: a thread can only observe its branch counter, not other thread's counter values.

Furthermore, we must consider the possibility that the next instruction to be executed is a branching instruction when all the replicas have the same number of branches, as shown in Listing 4.7. In this case, if the next instruction of $replica_0$ is line 2, the next instruction of $replica_1$ is line 5, and the `r9` registers for them are the same, $replica_1$ would be chosen as the leader since its IP is higher than the IP of $replica_0$. However, the actual case is that $replica_0$ is the leader since the branching instruction at line 5 has not been executed by $replica_1$. This scenario can happen due to the fact that the branch counting instruction and the actual branching instruction are separated. Because of this, we have to check if the previous instruction is the branch counting instruction by comparing the content of `(IP - 4)` with the binary form of the instruction—`0xe2899001`. The comparison is only required when the replicas have the same branch counters, and the additional check ensures that the correct leading replica is chosen. Nevertheless, just like checking for repeated string operations described in Section D.3, reading the content of `(IP - 4)` can cause a kernel page fault; so paging out code sections is not allowed for reducing implementation complexity.

```
1  L1:
2  add r8, r8, #1
3  cmp r8, #100
4  add r9, r9, #1
5  bne L1
```

Listing 4.7: An example of assembly code with a branch-counting instruction

Special considerations must taken in order to support ARMv7 atomic primitives. The primitives (e.g., `compare_and_swap`, `fetch_and_add`, and `fetch_and_dec`) are implemented using the load exclusive (`ldrex`) and store exclusive (`strex`) instructions. Take the assembly code of `fetch_and_add` in Listing 4.8 for example; `r0` contains the memory address of a variable, and `r1` specifies the value to be added to the variable. The value of the variable is loaded into `r2` at line 3, and we save the result of addition in `r3` at line 4. At line 5, we try to store the value in `r3` in to memory exclusively. Roughly speaking, the `ldrex` instruction loads the value and tags the underlying physical address for exclusive access; and the following `strex` instruction performs a conditional store that only succeeds if the tagged physical address is still in exclusive state. A value of 0 is returned in `r12` for a successful store; otherwise, a value of 1 is returned for a failure store. At line 7, we branch to the `retry:` label if a store fails so that the whole exclusive load-store sequence is retried. The precise and complete conditions that can fail an `strex` instruction are explained in [ARM, 2014]. For now, what we need to know is that an exclusive load-store sequence and its replicas running on different cores can be repeated different times.

```
1     dmb
2   retry:
3     ldrex       r2, [r0]
4     add         r3, r2, r1
5     strex       r12, r3, [r0]
6     cmp         r12, #0
7     bne         retry
8     dmb
9     mov         r0, r2
```

Listing 4.8: Function fetch_and_add

```
1     dmb
2   retry:
3     ldrex       r2, [r0]
4     add         r3, r2, r1
5     strex       r12, r3, [r0]
6     cmp         r12, #0
7     add         r9, r9, #1
8     bne         retry
9     dmb
10    mov         r0, r2
```

Listing 4.9: Function fetch_and_add with branch counting

The `fetch_and_add` function with inserted branch-counting instruction is shown in Listing 4.9. Since the `ldrex-strex` sequence can be retried several times and the numbers of retries on different cores can be different, the inserted branch-counting instruction at line 7 can be repeated different times as well. Therefore, the replicas may observe different numbers of branches even if they do not diverge. To address this issue, we use system calls to replace the synchronisation primitives that use the `ldrex-strex` sequences and implement the required operations in kernel mode, avoiding non-deterministic retries of the sequences.

Overall, this compiler-assisted approach is rather complex in implementation and inconvenient for end users (everything must be recompiled), so this approach is reserved as the last resort when neither LC-RCoE nor PMU-assisted CC-RCoE is applicable. Some implementation details on ARM are presented in Section D.5.

4.5 Loosely-Coupled Redundant Co-Execution (LC-RCoE)

Although CC-RCoE supports precise preemption, it can exhibit significant performance overhead, as shown later in Chapter 7. We implement the second realisation of the synchronisation protocol, called loosely-coupled redundant co-execution, aiming to optimise performance and simplify the implementation. The performance overhead is reduced since this approach does not need to read and write performance counters, to program debug registers, and to handle debug exceptions. The implementation is simplified because the approach uses only the deterministic event counter (the `events` in the triple used by CC-RCoE) as the logical clock. This implies that neither hardware support nor a compiler plugin is required, making the approach independent of architectures; and we do not need to deal with architecture-specific issues of CC-RCoE (some of the issues are described in Section D.3 and Section D.5).

The term loosely-coupled in this context means that the protocol does not have to synchronise the replicas at the same instruction if the applications are single-threaded, or multi-threaded and data-race-free. We first consider how LC-RCoE works when all applications are single-threaded, and then we explain how data-race-free multi-threaded applications can be supported if the microkernel is able to observe synchronisation operations (e.g., acquiring or releasing a mutex).

> In the loosely-coupled redundant co-execution approach, each kernel instance counts deterministic events that are observable by the kernel to construct the logical time of each replica.

For an application, its progress can be coarsely and simply measured by counting the number of deterministic events (e.g., system calls, exceptions, etc.) executed. These events obey the program order, and the whole execution of a replica can be viewed as sequences of chunks of deterministic events divided by non-deterministic interrupts. The microkernel can count these events effortlessly by incrementing an integer counter without instrumenting user code. If one replica runs faster than any other replicas, it has the highest event counter value. The synchronisation protocol ensures that the fastest replica waits until other replicas catch up, ensuring that the event counters of the replicas have the same value when the replicas are allowed to handle non-deterministic events. If all the applications are single-threaded and do not communicate through shared memory, they can be preempted anywhere between two deterministic events (inclusion) for the following reasons: (1) The replicas of an application can diverge if they observe inconsistent input data from system calls. (2) We ensure that the input data from I/O devices is consistent to all the replicas at the device driver level (details in Chapter 5). (3) LC-RCoE guarantees that preemptions are allowed only when the replicas have finished the same number of deterministic events.

4.5.1 Support for Multi-Threaded Applications

In the context of LC-RCoE, we discuss software-only approaches to support multi-threaded applications that are data-race-free. Although an application we replicate onto different cores is multi-threaded, all the threads of a replica of the application run on a single core. Thus, we do not need to resolve concurrent locking requests from threads running on different cores deterministically. We begin with the restrictions caused by imprecise preemption and explain the problem with an example. Then, we describe the requirements that should be satisfied to avoid execution divergence and present three approaches to fulfil the requirements in the context of seL4-based systems.

4.5.2 Imprecise Preemption and Restrictions

The microkernel can only observe application state changes through system calls and exceptions, so the lack of fine-grained tracking of thread progress introduces a restriction on threads: All observable state shared among threads in a replica is consistent between deterministic events. This limitation affects threads communicating through shared memory. Consider the simple code in Listing 4.10. There are two threads T_1 and T_2 running on each core, and `shared_x` is a shared variable used by T_1 and T_2 with an initial value 0. Now if $Core_0$ preempts T_1 before T_1 sets `shared_x` to 1 and $Core_1$ preempts $T_1{}'$ after $T_1{}'$ sets `shared_x` to 1, threads T_2 and $T_2{}'$ will diverge because they observe different values of shared_x, as in Figure 4.4. If the kernel knew that the values of shared_x are different, it could postpone the preemptions and only preempt T_1 and T_1' threads either before or after the values of shared_x are changed.

```
Thread1 (T1):
  statement0;
  /* core0 preempts T1 here */
  shared_x = 1;
  /* core1 preempts T1' here */
  statement1;

Thread2 (T2):
  if (shared_x == 1) {
  /* T2' on core1 executes */
  } else {
  /* T2 on core0 executes */
  }
```

Listing 4.10: Divergence caused by a race condition

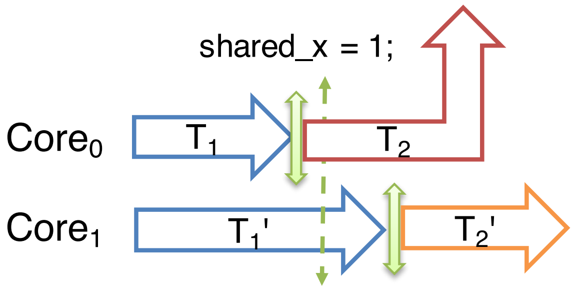

Figure 4.4: Timeline of the imprecise preemption example

Of course, this is a data race because the accesses to the shared variable shared_x from different threads are not protected by a lock; but merely putting locks around shared variables is not enough to avoid divergences. The requirements for supporting multi-threaded applications are very straightforward and intuitive: (1) Every locking or unlocking operation must be observed by the kernel so that the deterministic event counters also include such operations. (2) If the lock information data is accessible by all threads invoking the locking or unlocking primitives that use the data, the locking or unlocking operations must appear to be atomic. The need for the requirement (2) will become apparent later.We propose the approaches below to implement the locking and unlocking primitives that are applicable in our LC-RCoE environment.

A locking server

A microkernel-based system can be designed in a way that a lock server manages all locks in a multi-thread application, and the server runs as a separate thread. Locking and unlocking operations are translated to IPCs between the calling threads and the lock server. The calling thread provides the lock ID in the IPC and blocks for a reply by invoking seL4_Call. The lock server determines if the requested lock is free or not; for a free lock, the server resumes the execution of the calling thread by replying to it (seL4_Reply). Otherwise, the lock server waits (seL4_Wait) for another incoming request instead of replying the caller so that the caller is suspended until the lock is free. Note that the lock server needs bookkeeping data about the status of the locks, and we must ensure that the bookkeeping data is only accessed by the lock server. This approach is elegant and simple since the kernel already counts the three system calls; so in this case, the multi-threaded applications are supported naturally.

Notification objects as semaphores

The seL4 user-mode library provides synchronisation primitives including a mutex implementation based on notification objects [seL4, 2014]. When a mutex is being initialised, a notification object and a counter are allocated together. The counter represents the current status of the mutex and is initialised as one (unlocked). The pseudo code for locking and unlocking operations is shown in Listing 4.11.

```
1   int lock(seL4_CPtr notification, volatile int *value) {
2     int oldval;
3     int result = sync_atomic_decrement_safe(value, &oldval,
         __ATOMIC_ACQUIRE);
4     if (result != 0) return -1;
5     if (oldval <= 0) {
6       /* the lock is not free; wait on the notification object */
7       seL4_Wait(notification);
8       /* memory barrier */
9       __atomic_thread_fence(__ATOMIC_ACQUIRE);
10    }
11    return 0;
12  }
13
14  int unlock(seL4_CPtr notification, volatile int *value) {
15    int val = sync_atomic_increment(value, __ATOMIC_RELEASE);
16    if (val <= 0) {
17      /* wake up waiters */
18      seL4_Signal(notification);
19    }
20    return 0;
21  }
```

Listing 4.11: Locking/unlocking functions

In the code above, the pointer value references to a shared data; although the atomic instruction at line 3 is used to read and modify the data, preemptions can happen before and after the atomic instruction on different cores. Thus, a similar scenario as in Figure 4.4 can happen (replacing the shared_x with the pointer value). One straightforward remedy is to provide a variant of the seL4_Wait system call that takes an additional parameter, which is the address of the lock. The kernel can examine and modify the locking information, and block the calling thread on the notification object if the lock is held by another thread. All these steps appear atomic from the user-mode thread's point of view. Another way to address the issue is to disable preemption temporarily when a thread is in the lock function.

Bulk data transfer through shared memory

For seL4-based systems, processes communicating through shared memory for bulk data transfers usually are organised in a way that they use notification objects to interchange buffer-full or buffer-empty messages. When a producer process fills a buffer, a consumer process blocks on a notification object and only starts to read the buffer after it receives a “buffer-full” message. The producer process, on the other side, blocks for a “buffer-empty” message before it puts more data in the buffer. Since the producer and the consumer run on the same core, only one of them can access the shared memory at a time. Also, these messages sent through system calls are counted by the kernel as deterministic events so that the processes following this pattern to access shared memory are supported. However, other forms of accessing shared memory regions (e.g., polling a flag to check if data is ready) need case-by-case discussion.

4.6 Summary

In this chapter, we present our approach to replicating an seL4-based system on a multicore machine and executing the replicas redundantly. We examine the obstacles to apply SMR on a whole software system including the seL4 kernel, device drivers, and applications. This chapter focuses on the synchronisation protocol that coordinates the execution of the system replicas and ensures that non-deterministic events do not cause divergence. The protocol is modelled in SPIN and checked to guarantee not to introduce deadlocks. We presented two implementations of the synchronisation protocol: closely-coupled redundant co-execution (CC-RCoE) and loosely-coupled redundant co-execution (LC-RCoE).

CC-RCoE uses the triple `(events, branches, IP)` to pinpoint a position in an instruction stream, aiming to achieve precise preemption. On x86, we program performance counters to count the events `BR_INST_RETIRED.ALL_BRANCHES` and `BR_INST_RETIRED.FAR_BRANCH` in user mode and calculate the user-mode branches. For ARM Cortex-A9 series processors, we develop a GCC plugin that introduces one more pass during compilations. The new pass works at RTL level and scans for branching instructions. When such an instruction is found, the plugin inserts an instruction counting the branch before the branching instruction. The kernel replicas utilise the hardware instruction breakpoints and `(events, branches, IP)` together to stop the application replicas precisely at the same instruction and then handle non-deterministic events accordingly. Multi-threaded applications, even with data races, are supported by this variant. However, the compiler-based branch counting needs to recompile the whole user-mode software stack and GCC built-in functions; code written in assembly also needs to be processed.

LC-RCoE counts deterministic events that can be observed by the kernel; this variant does not depend on hardware features or compiler techniques so that it can be ported on different architectures straightforwardly (x86 and ARM are currently supported). However, multi-threaded applications can be supported only if they are data-race-free and if the lock and unlock operations meet the requirements described in Section 4.5.1. LC-RCoE does not preempt a thread and its replicas precisely at exact the same instruction to reduce performance overhead and implementation complexity.

The realisations of the synchronisation protocol work together with the techniques to support device driver replication (see Chapter 5), enabling a software system running in DMR or TMR on commodity hardware and thus establishing a solid foundation for the error detection mechanisms described in Chapter 6.

Chapter 5

Support for Device Driver Replication

> We do not know where we are "stupid" until we "stick our neck out," and so the whole idea is to put our neck out.
>
> Richard Feynman

Device drivers play important roles in our whole-system replication framework: they are the boundary between replicated software components and non-replicated I/O devices, and they function as input duplicators and output comparators. As the input duplicators, the driver replicas provide consistent input data to other replicated components (e.g., file systems, network stacks, etc.) of the system replicas, avoiding execution divergence caused by observing inconsistent input data. As the output comparators, the driver replicas check the final output data produced by other parts of the system, detecting inconsistent computation results.

We begin with describing how device drivers work in general and then introduce the access patterns used to coordinate I/O operations: we designate a replica as the *primary replica* to perform I/O operations and propagate the results to other replicas. The drivers modified to work with LC-RCoE and CC-RCoE require different types of support from the microkernel. For LC-RCoE, an existing system call is augmented to create *cross-replica shared memory regions* so that the driver replicas are able to conduct input data replication in user mode. For CC-RCoE, the replicas of a device driver must behave exactly the same (remember we count user-mode branches for precise preemption); so we provide two new system calls to conduct the input replication procedure in kernel mode (the branch instruction counters are stopped in kernel mode). Applying the access patterns needs to modify the source code of the drivers. The second part of the chapter focuses on the microkernel infrastructure to support primary replica migration, which is motivated by error masking in the TMR configuration. Should the primary replica be voted faulty, the functionality of the primary replica is transferred to another error-free replica that becomes the new primary replica. This chapter includes the work published in [Shen and Elphinstone, 2015; Shen et al., 2019].

5.1 Device Drivers

Device drivers communicate with I/O devices through I/O ports (x86) or memory-mapped registers (x86 and ARM). Reading I/O ports is supported by special instructions: outX and inX. The X specifies the width (8-bit, 16-bit, or 32-bit) of the data. The dx register contains the I/O port number to access, and the ax register has the input or output data. The following Listing 5.1 and Listing 5.2 show how to transfer 8-bit data from/to the first serial port on 32-bit x86 machines. The I/O address space has 2^{16} 8-bit ports, so the value range for dx is 0 to 0xffff, with 0xf8 to 0xff reserved. Accesses to I/O ports are restricted to the microkernel for seL4-based systems.

```
xor    %eax, %eax
movw   0x3f8, %dx
inb    %dx, %al
```

Listing 5.1: Read from I/O ports

```
movw   0x3f8, %dx
movw   0x38,  %ax
outb   %al, %dx
```

Listing 5.2: Write to I/O ports

For memory-mapped I/O, device registers can be accessed by normal load or store instructions, just like accessing physical memory. Usually, designers of ARM platforms SoCs (system on chip) arrange the allocations of device addresses statically since the peripherals integrated on a chip are known. For x86 machines, PCI (peripheral component interconnect) [Shanley and Anderson, 1999] devices can be added and removed dynamically; so the platform firmware (BIOS or UEFI) detects the available PCI devices, allocates bus addresses for the devices, and ensures that the addresses for the devices do not conflict with the addresses assigned to physical memory. The microkernel creates a list of frame capabilities covering the regions of the devices by scanning the PCI bus on x86 or reading the list known devices on ARM. The frame capabilities can be mapped into virtual memory space by their owners. Once the mapping is established, the drivers can access the device registers without restrictions; so the kernel cannot monitor such accesses without incurring overhead (e.g., using debugging hardware to monitor memory accesses).

5.1.1 Devices Reserved by the Microkernel

The microkernel reserves several devices for its own use. On x86, the devices are the IO-APIC (I/O advanced programmable interrupt controller), per-core local-APIC, and APIC timer. The IO-APIC is used to control interrupt routing. The per-core local-APIC is used to enable, disable, and acknowledge the interrupts that sent to the core. The APIC timer is used to generate periodic preemption ticks. On the ARM Cortex-A9 based platforms, seL4 uses the GICD (generic interrupt controller distributor), per-core GICC (generic interrupt controller CPU interface), L2 cache controller, and per-core private timer. The GICD and GICC are functionally similar to the IO-APIC and local-APIC: they manage hardware interrupts. The private ARM timer provides kernel preemption interrupts. The L2 cache controller is necessary for the i.MX6 platform, which uses a separate controller for L2 cache, and the kernel must have full control over cache cleaning, invalidating, and flushing operations. Among these devices, the IO-APIC, GICD, and L2 cache controller are per-machine devices that can be accessed by all cores.

5.1.2 Simple Devices

We call the devices that only use I/O ports or memory-mapped device registers for data transfers *simple devices*. In other words, simple devices do not use direct memory access (DMA) for data transfers. For example, UART (universal asynchronous receiver/transmitter), PIT (programmable interval timer), and PIC (programmable interrupt controller) are such devices. Some UART chips are capable of DMA, but we do not use the function. The following requirements are sufficient to replicate such drivers.

1. For each read, one of the driver replicas is allowed to carry out the read. All the replicas need to wait until the read is finished so that they all observe the read result.
2. For each write, one of the driver replicas is allowed to carry out the write. The values from the replicas can be optionally compared to detect an potential error.

The output comparison and input replication need a mechanism that enables communications among the replicas of a driver, but, not surprisingly, the unmodified microkernel does not have support for the purpose.

5.1.3 DMA-Enabled Devices

High-speed I/O devices, such as network cards, hard disk controllers, or video cards, use DMA for bulk data transfers. The DMA engines of the devices use physical memory addresses if IOMMU (input-output memory management unit) is not present or not programmed to perform address translations for I/O devices. As an example, we show the replicated network interface card (NIC) drivers and the hardware NIC managed by the drivers in Figure 5.1 Two memory-mapped I/O (MMIO) regions in the figure are mapped to the NIC registers so that both the drivers are able access the registers. When the driver is not replicated, the NIC stores incoming network packets in the physical memory regions (DMA buffers) specified by its driver and generates interrupts to notify the driver that the data is ready. However, in the replicated case, the NIC is not aware of the situation that the driver is replicated—each driver instance has its own DMA buffers so that only one replica's DMA buffers contain the incoming data.

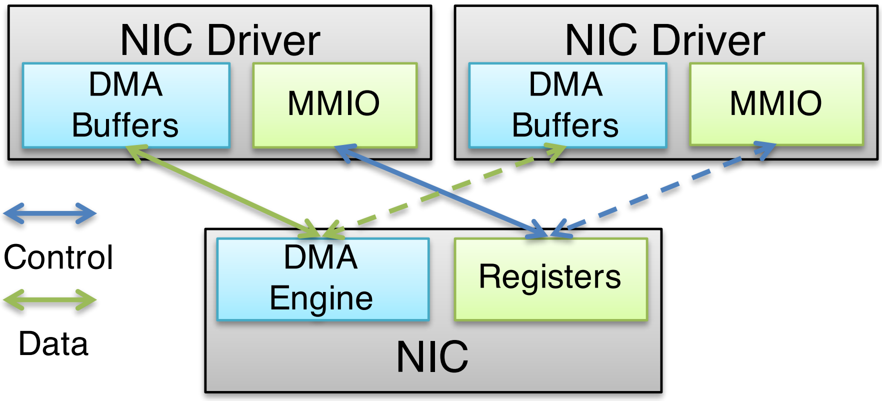

Figure 5.1: Replicated NIC drivers

In Figure 5.1, the solid green line represents data transfers performed by the DMA engine, and the dashed green line stands for the data transfers that ideally should be automatically duplicated, based on the solid-green-line transfers by the DMA engine; so the input data is copied into multiple DMA buffers that belong to different driver replicas. However, COTS network cards do not support such functionality yet. The IOMMU address translation cannot help on this issue either since it is not designed to translate one device address to multiple physical addresses and to duplicate the input data. Therefore, the other driver replica on the right in the figure does not observe the incoming data and diverges from the replica on the left. We name the DMA buffers actually used by I/O devices as the *real DMA buffers* and the DMA buffers not being used by devices as the *shadow DMA buffers*. Therefore, we need a mechanism to ensure that either all replicas observe input data from the real DMA buffers or the input data in the real DMA buffers is replicated to the shadow DMA buffers.

5.2 The Primary Replica and Access Patterns

We designate one system replica in a DMR or TMR system as the *primary replica*, and the device drivers of the primary replica are responsible for actually performing I/O accesses. Also, we program the interrupt controllers (the GICD and IO-APIC) to reroute device interrupts to the primary replica. The way we appoint a primary replica is simple: the replica with ID 0 is the primary replica; so by default, the bootstrapping replica is the primary replica. However, the primary replica is not permanent. As we will introduce in Chapter 6, if the primary replica is voted as the faulty replica in a TMR configuration, the remaining replicas designate another fault-free instance as the primary replica. But for now, let us assume that the primary is the one with ID 0.

The general access patterns applied to the drivers and the microkernel (the kernel also needs to operate on the reserved devices) are listed in Listing 5.3 and Listing 5.4; let's examine the read pattern first. The function `is_primary` compares a replica's ID with the ID of the primary replica. Lines 1 and 2 allow the primary replica to perform the read access and store the result in a shared memory buffer which is accessible by all replicas. The barrier at line 3 blocks all non-primary replicas to ensure that the result is ready in the shared memory buffer since all replicas can only pass the barrier when the primary replica finishes the operations and joins the barrier. At line 4, each replica copies the data to its private memory. The barrier at line 5 guarantees that the shared buffer will not be modified by the primary replica at line 2 before every replica finishes copying the data. The pattern for writes is straightforward—the primary replica simply writes the data out.

```
if (is_primary())
  shared = read_data();
barrier();
local = shared;
barrier();
return local;
```

Listing 5.3: Access pattern for reads

```
if (is_primary())
  write_data(local);

return;
```

Listing 5.4: Access pattern for writes

For the devices used by the seL4 kernel, we directly modify the kernel to apply the patterns. For the drivers used in LC-RCoE, we modify the drivers to apply the patterns directly. Each kernel instance also passes its replica ID to the root task, which selectively forwards the ID to the device drivers, but applications are not allowed to observe the ID by any means. The implication is that we trust the root task and drivers not to abuse the IDs. Given the fact that we are able to inspect the source code, we believe exposing the replica IDs to the device drivers is not a concern. For LC-RCoE, we build the barriers in the user-mode cross-replica shared memory region (details in Section 5.4.1). For the drivers used in CC-RCoE, we supply a new system call named `seL4_FT_Mem_Rep` and modify the drivers to use the system call. The system call performs the accesses in kernel mode because otherwise the patterns will cause the replicas of a driver to issue different number of branch instructions (more in Section 5.5). The new system call `seL4_FT_Mem_Rep` uses kernel-mode barriers.

The primary replica is designated to perform I/O accesses; the seL4 kernel and device drivers are modified to follow the access patterns.

5.3 Support for Accessing I/O Ports

I/O ports are governed by the seL4 kernel objects so that only the owners of the I/O port capabilities can read or write the I/O ports. Each kernel replica creates one I/O port capability that covers all ports and passes it to the root task; other I/O port capabilities that authorise accesses to specific ranges of I/O ports can be derived from the initial capability by using the `seL4_CNode_Mint` system call. The root task can then distribute the derived capabilities to different device drivers needing I/O port accesses. In this way, each driver replica has valid capabilities that can be used as parameters for various I/O port related system calls (e.g., `seL4_IA32_IOPort_In8` and `seL4_IA32_IOPort_Out8`).

We modify the kernel code that handles I/O port operations according to the patterns described above, and the input data is propagated through the shared kernel data region. The primary replica accesses the ports on behalf of driver replicas and returns the results for read operations. For output data, the kernel replicas exchange and compare the port number and data through the shared kernel data region to ensure that the output requests are consistent. Incompatible output requests are treated as errors. The code changes are common to both LC-RCoE and CC-RCoE. We list the C code in Listing 5.5 for `seL4_IA32_IOPort_In8` and `seL4_IA32_IOPort_Out8` as two detailed examples to illustrate the access patterns. The variable `ft_io_bar` is the kernel barrier for coordinating the accesses, and `ft_io_input_val` resides in kernel shared memory region so that it serves as the temporary buffer for copying input data. The code also demonstrates the need for the conditional barrier (Listing 4.4) so that `ft_abort_sync_wait()` at lines 11 and 27 can wake up the replicas blocked on the conditional barrier in the synchronisation protocol.

```
 1  /* the barrier for IO port operations */
 2  DATA_GLOB volatile kbar_t ft_io_bar;
 3  /* shared buffer for input replication */
 4  DATA_GLOB volatile uint32_t ft_io_input_val;
 5  /* shared buffer for output comparison */
 6  DATA_GLOB volatile ft_io_out_t ft_io_out_val[NUM_REPLICAS];
 7
 8  uint32_t ft_io_in8(uint16_t port)
 9  {
10    uint32_t val = 0;
11    ft_abort_sync_wait();
12    kbar_waitn(&ft_io_bar, NUM_REPLICAS);
13    /* the primary replica performs the access */
14    if (is_primary()) {
15      ft_io_input_val = in8(port);
16      ft_io_input_val &= 0xff;
17    }
18    kbar_waitn(&ft_io_bar, NUM_REPLICAS);
19    val = ft_io_input_val;
20    return val;
21  }
22
23  void ft_io_out8(uint16_t port, uint8_t data)
24  {
25    ft_io_data[my_replica_id].port = port;
26    ft_io_data[my_replica_id].data = data;
27    ft_abort_sync_wait();
28    /* the barrier ensures that the output data to be
29     * compared is ready in ft_io_data */
30    kbar_waitn(&ft_io_bar, NUM_REPLICAS);
31    for (int i = 0; i < NUM_REPLICAS; i++) {
32      if (ft_data[i].port != ft_data[my_replica_id].port ||
33          ft_data[i].data != ft_data[my_replica_id].data)
34        ft_halt();
35    }
36    /* the barrier ensures the ft_io_data is not modified
37     * until all replicas finish the output data comparison */
38    kbar_waitn(&ft_io_bar, NUM_REPLICAS);
39    /* the primary replica performs the access */
40    if (is_primary()) out8(port, data);
}
```

Listing 5.5: I/O port functions for the DMR configuration

5.4 Kernel Mechanisms and Driver Modifications for LC-RCoE

5.4.1 Cross-Replica Shared Memory Regions (LC-RCoE)

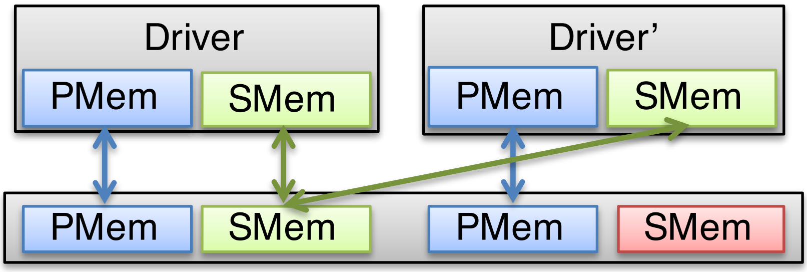

Figure 5.2: Cross-replica shared memory

In the read access pattern Listing 5.3, `local` represents the local data for each replica, and `shared` points to a shared memory region that can be accessed by all the replicas. Figure 5.2 shows a pair of driver replicas. *PMem* (private memory) and *SMem* (shared memory) are all virtual memory regions used by the driver replicas, and the virtual addresses and sizes of the PMem and SMem regions in *Driver* are the same as the ones in *Driver'*. The PMem regions are private to the replicas since they are mapped to different physical memory regions (as indicated by the blue arrows), but the SMem regions are mapped to the same physical memory region (as indicated by the green arrows) so that the replicas can communicate through the regions. We call the green SMem regions as the *cross-replica shared memory regions*. As to the access pattern, the `local` and `shared` variables are allocated in the PMem and SMem regions respectively. The red SMem box is the physical memory region that would be mapped to the green SMem box of Driver', if the mechanism for building cross-replica shared memory regions were not used.

The prototypes of the system calls `seL4_ARM_Page_Map` and `seL4_IA32_Page_Map` are shown in Listing 5.6. These system calls map a `frame` representing a physical memory region into a page table pd to back up a virtual memory region starting from `vaddr` with `rights` and attributes `attr`. The `rights` parameter specifies read-only or read-write permission, and the `attr` parameter describes the cacheability (cache enabled or disabled), write policies (for x86 only; i.e., write back, write through, or write combining), and parity (for ARM only). Note that the corresponding VM mapping system calls issued by the replicas must have the same parameters. In the current form, the system calls do not support the function of constructing cross-replica shared memory regions as in Figure 5.2.

```
int seL4_ARM_Page_Map(seL4_ARM_Page frame,
                      seL4_ARM_PageDirectory pd,
                      seL4_Word vaddr, seL4_CapRights rights,
                      seL4_ARM_VMAttributes attr);

int seL4_IA32_Page_Map(seL4_IA32_Page frame,
                       seL4_IA32_PageDirectory pd,
                       seL4_Word vaddr, seL4_CapRights rights,
                       seL4_IA32_VMAttributes attr);
```

Listing 5.6: Virtual memory mapping system calls

To create the cross-replica shared memory regions, we extend the system calls above with one additional attribute—`SHARED`. The kernel code is changed to handle the new option: (1) The virtual addresses passed by the replicas are compared for consistency; different addresses are rejected. (2) The physical addresses of the `frames` are checked. Instead of comparing absolute values, we examine the offsets to the beginning addresses of the memory regions allocated to the replicas. Similarly, if the offsets mismatch, the mapping requests are rejected. (3) Having successfully checked the system call parameters, each kernel replica uses the physical address supplied by the primary replica to update the page table that is used by the replica, creating a cross-replica shared memory region. (Note the non-primary replicas can calculate the physical address used by the primary replica.) Accordingly, the replicas of the driver can communicate through the shared memory regions and use the regions to propagate input data. We need to emphasise that only the replicas of the same device driver can communicate through the cross-replica shared memory.

For LC-RCoE, the system calls mapping physical frames to virtual memory spaces and the kernel code for handling the system calls are augmented to support the `SHARED` attribute for creating cross-replica shared memory regions. The barriers and shared memory variables for replicating input data in the read pattern are also constructed in the cross-replica shared memory regions .

5.4.2 Driver Modifications for Accessing MMIO Device Registers (LC-RCoE)

Device drivers need to first map device frames into their VSpaces before they can access MMIO-based device registers. Each kernel replica creates the original device frame capabilities according to the PCI device discovery procedure or the predefined device MMIO regions; the root task owns all the device frames so that it can distribute them to the corresponding drivers appropriately. All replicas of a driver map the assigned device frames to their VSpaces at the same virtual addresses so that they can access the MMIO regions. The mapping step is unchanged. Nevertheless, the access patterns described above are applied to the functions that read or write device registers, allowing only the primary replica to perform operations. The reason for permitting all the replicas to map the device frames is for error masking: when the primary replica is faulty and we need to remove it, we can elect a new primary replica so that the new one can read and write devices without mapping the device frames first. Another reason is to minimise the code changes to the microkernel and device drivers.

5.4.3 Driver Modifications for Accessing DMA Buffers (LC-RCoE)

Device drivers allocate DMA buffers through the functions provided by the seL4 user-mode library. We modify the DMA-related library functions so that the cross-replica shared memory regions are created and used for DMA buffer allocations. Since we only expose the physical memory addresses of the primary replica to all replicas, the I/O devices are programmed to utilise the primary replica's DMA buffers. Thus, the buffers created in the cross-replica shared memory regions are accessible by all corresponding driver replicas. The access patterns are applied to DMA buffer operations: (1) the primary replica can write to the DMA buffers; (2) a barrier is required to coordinate reading contents from the DMA buffers consistently. Furthermore, we can compare the output contents of the replicas before writing them to the DMA buffers for transmitting, as shown in Listing 5.7.

```
chksum = gen_chksum(output);
seL4_Add_Event(chksum, true);
if (is_primary())
  write_dma_buf(output);
```

Listing 5.7: Check DMA output data

Each replica generates a checksum (the checksum algorithm is chosen by the driver) based on the output data; the checksums are compared with the `seL4_Add_Event` system call which will be described in Section 6.2.1. Basically, the system call adds the `chksum` to the per-replica execution fingerprint (Section 6.2) and compares the fingerprints before returning to user mode if the second parameter is `true`. If the checksums are not consistent, the system call halts the system if the fail-stop approach is adopted.

Replicating device drivers disables the opportunity of using an optimisation called zero-copy DMA for input data. Take a network card driver for example; when the zero-copy DMA optimisation is applied, the driver may decide to pass the addresses of the buffers to an upper level (e.g., a network stack) for further process, without making a copy of the data first. However, in the case of allocating DMA buffers in the cross-replica shared memory regions, we cannot guarantee that upper-level applications will not modify the buffers which are shared by all replicas. Uncoordinated changes to the buffers made by one replica can be observed by other replicas and subsequently cause divergence, unless the upper-level applications are also modified to follow the access patterns. Therefore, all the replicas must copy the contents in the DMA buffers to their private buffers which will be subsequently passed to upper-level applications. The extra copies introduce runtime overhead for I/O-intensive systems (see Section 7.4.1).

5.4.4 Summary

The key enabling mechanism for supporting driver replication in LC-RCoE is the kernel's ability of creating cross-replica shared memory that is subsequently used to build user-mode barriers and to replicate input data. Based on our redundant co-execution model, the kernel code to support the mechanism is straightforward since LC-RCoE ensures that the parameters for the corresponding seL4_IA32_Page_Map or seL4_ARM_Page_Map calls issued by the driver replicas are the same. Another advantage of this mechanism is that the microkernel does not need to access memory regions belonging to the device drivers. As we will see later, CC-RCoE does not possess this property; thus, the microkernel has either to trust the drivers or to perform tedious checking against the addresses provided by the drivers. Fundamentally, the more relaxed (in term of not requiring consistent user-mode branches) LC-RCoE model allows us to adopt the simple mechanism to support device driver replication.

5.5 Kernel Mechanisms and Driver Modifications for CC-RCoE

5.5.1 New System Calls Implementing the Access Patterns in Kernel Mode

In the closely-coupled redundant co-execution approach, we count user-mode branches by using the performance counters or the compiler plugin that inserts branch-counting instructions; the requirement of CC-RCoE is that the number of branches executed by the replicas of an application must be exactly the same. Therefore, the patterns described in Section 5.2 are not suitable for user-mode drivers in CC-RCoE since the patterns introduce inconsistent branches for the driver replicas. Take the `if (is_primary()) shared = read_data()` in Listing 5.4 for example; even if we assume that the `read_data()` is inlined properly and that it does not contain branch instructions, the primary replica still executes one more or one less branch instruction than the other replicas, depending how the compiler treats the `if` statement. Furthermore, the user-mode barrier implementation based on the cross-replica shared-memory mechanism causes the replicas to execute different numbers of branches because the replica coming early waits others in a busy loop. In summary, to work properly in CC-RCoE, the device drivers have to hide the access patterns so that all the replicas of a driver still issue the same number of branches. We provide two new system calls (function prototypes are shown in Listing 5.8) for device drivers.

```
int seL4_FT_Mem_Access(seL4_Word access_type, seL4_Word va_mmio,
            seL4_Word va_src_dest, seL4_Word size);
int seL4_FT_Mem_Rep(seL4_Word va, seL4_Word size);
```

Listing 5.8: seL4_FT_Mem_Access and seL4_FT_Mem_Rep

The system call `seL4_FT_Mem_Access` targets for MMIO regions. `access_type` specifies an access is a read or write, and `va_mmio` is the MMIO virtual address used for reading input data or writing output data. The parameter `va_src_dest` provides the virtual address where the input data should be stored for a read or where the output data should be taken from for a write. Lastly, the parameter `size` defines the MMIO region (`[va_mmio, va_mmio + size)`) for an operation in bytes. For a read access, the primary replica performs the read and stores the result in the kernel shared memory region. After that, all replicas copy the stored result to their private memory regions specified by the `va_src_dest` parameters. For a write, the primary replica reads out the output data from the address indicated by va_src_dest and writes the data to the MMIO region described by va_mmio.

The system call seL4_FT_Mem_Rep is for accessing DMA buffers. It copies the contents of the primary replica's memory region ([addr, addr + size)) to the corresponding memory regions belonging to other replicas, hiding the input data replication procedure in the kernel mode. To handle the system call, the primary kernel replica reads size bytes from the address specified by addr to the shared kernel data region, and other kernel replicas copy the data to the per-replica memory regions. The pseudo code for this system call is showed in Listing 5.9. Note that the kernel barriers used to implement the system calls belong to the F-barrier type (Section 4.3).

```
DATA_GLOB volatile char ft_mem_rep_buf[BUF_SZ];
DATA_GLOB volatile bar_t ft_mem_rep_bar;
ft_mem_rep(unsigned long va, unsigned long size)
{
  ft_abort_sync_wait();
  while (size > 0) {
    /* the primary replica reads the data to the shared buffer */
    if (is_primary()) {
      if (size > BUF_SZ) {
        memcpy(ft_mem_rep_buf, va, BUF_SZ);
        size -= BUF_SZ;
        va += BUF_SZ;
      } else {
        memcpy(ft_mem_rep_buf, va, size)
        size -= size;
      }
    }
    /* ensure that the data is ready */
    kbar_wait(&ft_mem_rep_bar, NUM_REPLICAS);
    /* other replicas copy the data to their memory regions */
    if (!is_primary()) {
      if (size > BUF_SZ) {
        memcpy(va, ft_mem_rep_buf, BUF_SZ);
        size -= BUF_SZ;
        va += BUF_SZ;
      } else {
        memcpy(va, ft_mem_rep_buf, size);
        size -= size;
      }
    }
    /* ensure other replicas finish the current copy before
       starting the next round */
    kbar_wait(&ft_mem_rep_bar, NUM_REPLICAS);
  }
}
```

Listing 5.9: The pseudo code for seL4_FT_Mem_Rep

For CC-RCoE, two system calls, seL4_FT_Mem_Access and seL4_FT_Mem_Rep, are introduced to support input data replication for device drivers in kernel mode so that the access patterns do not affect the user-mode branch-instruction counters.

5.5.2 Driver Modifications for Accessing MMIO Device Registers (CC-RCoE)

The modified device drivers map the device MMIO regions into their virtual memory spaces as normal. However, every MMIO region access is replaced with the seL4_FT_Mem_Access system call. As an example, the original and modified code is listed in Listing 5.10. Note that ctrl_reg is a variable on the stack and that the virtual addresses of ctrl_reg are the same for all the replicas. The modifications are straightforward and potentially can be automated. Nevertheless, the runtime performance overhead is nontrivial: a simple memory access is replaced by a system call which can cost hundreds or thousands of CPU cycles.

```
/* original MMIO read */
uint32_t ctrl_reg = *((volatile uint32_t *)dev->ctrl_reg);

/* modified MMIO read */
uint32_t ctrl_reg = 0;
seL4_FT_Mem_Access(FT_READ, dev->ctrl_reg, &ctrl_reg,
    sizeof(ctrl_reg));

/* original MMIO write */
uint32_t ctrl_reg |= ENABLE_BIT;
*((volatile uint32_t *)dev->ctrl_reg) = ctrl_reg;

/* modified MMIO write */
uint32_t ctrl_reg |= ENABLE_BIT;
seL4_FT_Mem_Access(FT_WRITE, dev->ctrl_reg, &ctrl_reg,
    sizeof(ctrl_reg));
```

Listing 5.10: Modified MMIO access code

5.5.3 Driver Modifications for Accessing DMA Buffers (CC-RCoE)

For CC-RCoE, the library functions related to DMA buffer creation and allocation are not modified. Thus, the DMA buffers belonging to different replicas are mapped to separate physical memory regions; and only the DMA buffers of the primary replica contain input data from I/O devices. The seL4_FT_Mem_Rep system call is inserted before the places where input data in the buffers is about to be used. By replicating the data in the primary replica's buffer to the corresponding buffers belonging to other replicas, the system call ensures that consistent input data is observed by all replicas when the data is used later. The zero-copy DMA optimisation can be used to amortise the overhead introduced by the additional copies conducted by the system call since each replica has a private copy of the DMA buffer after executing the system call.

5.5.4 Security Concerns

These two system calls seem dangerous if the parameters provided by the drivers are incorrect. Now we examine the system calls and investigate what can happen if the system calls are purposefully or accidentally misused.

access_type	mmio_va + size	va_src_dest + size	Results
FT_READ	Valid MMIO	Valid Data/Stack	(1) Read OK
		Invalid Mem	(2) Kernel-mode VM Faults
		Valid MMIO	(3) Unknown Effects
FT_READ	Invalid Mem Address	N/A	(4) Kernel-mode VM Faults
FT_READ	Valid Mem, not MMIO	Valid Data/Stack	(5) Incorrect Behaviour
		Invalid Mem	(6) Kernel-mode VM Faults
		Valid MMIO	(7) Unknown Effects
FT_WRITE	Valid MMIO	Valid Data/Stack	(8) Write OK
		Invalid Mem	(9) Kernel-mode VM Faults
		Valid MMIO	(10) Unknown Effects
FT_WRITE	Invalid Mem Address	N/A	(11) Kernel-mode VM Faults
FT_WRITE	Valid Mem, not MMIO	Valid Data/Stack	(12) Inconsistent Replicas
		Invalid Mem	(13) Kernel-mode VM Faults
		Valid MMIO	(14) Inconsistent Replicas

Table 5.1: The combinations of inputs for `seL4_FT_Mem_Access`

We discuss `seL4_FT_Mem_Access` first. The first parameter is `access_type`, which can be `FT_READ` or `FT_WRITE`. The second parameter is `va_mmio` that contains the virtual address backed up by device registers, and `va_src_dest` should be the virtual address of data or stack regions. However, a malicious or buggy driver may provide one of the following types of addresses to the `va_mmio` and `va_src_dest`: an invalid address, an address backed up by normal memory, an MMIO address, or even a kernel address. We list the possible combinations and consequences of the input parameters in Table 5.1. To check if an address belongs to the kernel reserved virtual memory region is straightforward, so we skip the cases that use kernel virtual addresses as parameters. We can group the results (1) to (14) into five categories:

- The cases (1) and (8) are correction operations. The parameters are correct, and operations are correctly performed.
- The cases (2), (4), (6), (9), (11), and (13) cause kernel-mode VM faults. The kernel currently halts the system when a VM fault occurs in kernel mode.

- The cases (3), (7), and (10) cause undetermined effects. Unknown data from MMIO regions ((3) and (10)), or memory ((7)) is written to an MMIO region; the exact consequences depend on the contents being written and the functionality of the MMIO region. We expect that these cases will cause the I/O device to malfunction; however, we do recognise the possibility that the DMA engine of the device may be programmed to access physical memory regions that the driver does not have access permissions (e.g., physical memory used by the microkernel). For this issue, we can program the IOMMU to restrict the physical memory regions that an I/O device is able to access.

- The cases (12) and (14) introduce inconsistent replicas of the driver. Only the primary replica's memory is modified with data from memory or from MMIO. If the inconsistency originates execution divergence or comparison failures, the system halts.

- The case (5) may cause the driver replicas to behave incorrectly since incorrect data is read from the primary replica's memory instead of device registers. For example, if the driver needs to check the interrupt status and incorrect information about the interrupt is supplied from a physical memory region, the driver may act incorrectly.

A similar analysis can be conducted for the `seL4_FT_Mem_Rep(va, size)` system call as well. The scenarios are fewer because of reduced number of parameters.

- If the $[va, va+size)$ is a valid memory region used for DMA buffers, the system call performs as expected.

- If the $[va, va+size)$ is a valid and writeable memory region but not used for DMA buffers, the system call performs. But instead of copying from primary replica's DMA buffers, the system call copies data from the primary replica's non-DMA memory regions. However, CC-RCoE ensure that the data in these regions belonging to different replicas of a driver is the same; so the system call has no effects.

- If the $[va, va+size)$ is fully or partially invalid or read-only, kernel-mode VM faults will be triggered; thus, the system halts.

- If the $[va, va+size)$ is a MMIO region, the same contents are read from and written to the MMIO region; how the device is affected by such operations needs further case-by-case analysis.

- If the $[va, va+size)$ contains kernel-reserved virtual addresses, we reject the system call and halt the system.

In summary, these are risks if the provided system calls are deliberately used to interfere correct operation of a system. However, we need to modify the source code of the drivers so that it is unlikely that malicious uses of the system calls will exist in the drivers. In the worst case, we can validate the input addresses by walking through the page tables to identify if the addresses are translated to valid physical memory regions or MMIO regions, at the cost of increased overhead. One crucial character shared by the two system calls is that they can only be used to construct channels among the replicas of a device driver; in other words, the replicas of the driver A cannot communicate with the replicas of the driver B by using the two system calls.

5.6 Interrupt Delivery

Device interrupts are delivered to the primary replica; so we need a mechanism to propagate the interrupts to other replicas. As we introduced in Section 2.4.4, a device driver observes interrupts by waiting on a notification object; so the driver replicas can observe the interrupts consistently since the positions of the `seL4_Wait` system call are deterministic. However, the interrupt delivery procedure usually involves preempting the current threads and context-switching to the replicas of the driver. Thus, the kernel instances need to execute the synchronisation protocol first, ensuring the preemption of the current threads will not cause execution divergence.

> The microkernel is modified to buffer interrupts and to trigger synchronisations before injecting the interrupts to all driver replicas, ensuring consistent observations of the interrupts by all driver replicas through the `seL4_Wait` system call and notification objects.

The code changes we made to the kernel entry point for interrupt handling is shown in Listing 5.11. The original entry point calls the interrupt handler `handleInterrupt(irq)` directly if the received `irq` is valid. The modified entry point has different execution paths for the primary replica and non-primary replicas. The primary replica buffers the valid `irq` and triggers a round of synchronisation by calling `ft_trigger_action` at line 16. After that, `ft_arch_sync_notify` is called to send IPIs (inter-processor interrupts) that interrupt the execution of non-primary replicas. The source of the current `irq` is disabled (the device driver can re-enable it), and the interrupt controller is acknowledged so that interrupts from other devices can still come in. The non-primary replicas only handle the synchronisation IPIs; the function `sync` is called by all replicas to start the synchronisation protocol.

```
1  /* original interrupt handler */
2  handle_interrupt() {
3    irq = get_irq();
4    if (irq != irqInvalid) handleInterrupt(irq);
5    schedule();
6    activateThread();
7    return EXCEPTION_NONE;
8  }
9
10 /* modified interrupt handler */
11 ft_handle_interrupt() {
12   irq = get_irq();
13   if (is_primary_replica() && irq != irqInvalid) {
14     /* the primary replica triggers a synchronisation and
15      * and saves the irq number. */
16     ft_trigger_action(FT_SYNC_EVT_INT, irq);
17     /* notify other replicas by sending IPIs */
18     ft_arch_sync_notify();
19     /* mask the interrupt source */
```

```
    maskInterrupt(true, irq);
    /* interrupts from other sources can still fire */
    ackInterrupt(irq);
  } else {
    /* non-primary replicas only respond to IPIs */
    if (irq != FT_SYNC_IPI) return;
  }
  /* call the synchronisation protocol */
  sync();
}
```

Listing 5.11: The code changes made to the kernel interrupt entry point

When the replicas are synchronised, the saved interrupts are injected in a batch as shown in Listing 5.12. The batch injection implies that multiple drivers can become ready to run. Still, the thread with the highest priority among the runnable threads will be chosen to execute first.

```
/* this function is called when the replicas are synchronised */
handle_event() {
  switch (sync_evt) {
  case FT_SYNC_EVT_INT:
  /* multiple interrupts can be saved by ft_handle_interrupt;
   * we call the original kernel interrupt handle
   * handleInterrupt to process the saved interrupts. */
    foreach (irq in saved_irqs)
      handleInterrupt(irq);
    clear_saved_irqs();
    break;
  /* other synchronisation events are omitted */
  }
}
```

Listing 5.12: The pseudo code for interrupt injection

5.7 Primary Replica Migration

To support error masking in TMR mode, we need the ability to migrate the functionality of the primary replica to another core if the primary replica is voted faulty. We postpone the details of the error masking mechanisms to Section 6.5.2 and focus on the microkernel infrastructure to support the migration. After the migration, the new primary replica will have accesses to the device MMIO regions, I/O ports, and DMA buffers. Since CC-RCoE and LC-RCoE support MMIO and DMA accesses differently, we examine them separately.

5.7.1 I/O Ports

As described in Section 5.3, the I/O port operations are performed by the microkernel on behalf of the device drivers. The approach is used for both CC-RCoE and LC-RCoE, so the driver replicas only using I/O ports are completely unaware of the migration, since the interactions with I/O ports are hidden. Note the kernel I/O port functions supporting error masking are slightly different from the code in Listing 5.5. For example, the function `is_primary()` for a TMR system with error masking needs to know when the primary replica migrates; so the function compares the replica ID with a variable containing the ID of the current primary replica. Additionally, the kernel mode barriers need to adjust their internal counters accordingly when a system downgrades from three replicas to two replicas after the faulty replica has been removed.

5.7.2 Memory-Mapped I/O

CC-RCoE and LC-RCoE establish the mappings of device MMIO regions for all replicas so that each replica, be it the primary replica or not, can access device registers. (This is the reason why we use the access patterns in Section 5.2 to coordinate MMIO accesses.) For CC-RCoE, we provide new system calls Section 5.5.1 to implement the access patterns in kernel mode so that the user-mode branch counting is unaffected by the patterns. Thus, just as the case for I/O port operations, the driver replicas are not affected by a migration since the accesses are handled by the microkernel in a way that the driver replicas observe the completions of accesses atomically.

```
1  if (is_primary())
2    shared = read_data();
3  barrier();
4  local = shared;
5  barrier();
6  return local;
```

Listing 5.13: Access pattern for reads (repeat)

```
1  disable_migration();
2  if (is_primary())
3    shared = read_data();
4  barrier();
5  local = shared;
6  barrier();
7  enable_migration();
```

Listing 5.14: Read access pattern with error masking

On the other hand, LC-RCoE implements the patterns in user mode. The problem here is that a faulty replica can be removed any time while the system is running, since non-deterministic device interrupts can come in and trigger synchronisations. Consider the code in Listing 5.13; if a replica is removed at line 4 and other replicas are waiting at the barrier at line 5, the waiting replicas cannot pass the barrier. Therefore, we need a mechanism to notify the blocking replicas that the number of active replicas has changed, so they can abort the current round and call the barrier again with the reduced number of active replicas. Now, let us consider another scenario: the primary replica is removed between line 1 and line 2, and other replicas are waiting at line 3. Assume the notification mechanism discussed above exists, so the waiting replicas pass the barrier and arrive at line 4 and start to copy the input data from the cross-replica shared memory region. However, the data in the shared buffer is incorrect since the previous primary replica did not perform the read operation. Similarly, if the primary replica is removed between line 4 and line 5, and the newly chosen primary replica resumes execution at line 1, the new primary replica will read the device again at line 2. However, unlike normal memory, reading memory-mapped I/O region usually has side effect so reading twice should be forbidden. The microkernel cannot distinguish if the primary replica has finished the read operation and stored the result to the shared memory buffer without decoding instructions and analysing source/destination memory addresses. Therefore, if any replica is in the middle of accessing MMIO regions, we decide to abort the attempt of migrating the primary replica and to halt the whole system instead (see Section 6.5.2).

For the purpose of the drivers advising the microkernel that an MMIO access is being executed and the purpose of the microkernel notifying the drivers that a faulty replica has been removed, we need to establish bidirectional channels between the microkernel and the drivers. The channels should incur little overhead during normal operations, so frequently executing a system call, which usually costs hundreds of cycles each, is not an ideal choice. Our approach is to set up a dedicated memory region between each driver replica and the microkernel replica, and the region contains the following items.

`num_active_replicas` represents the number of active replicas. The microkernel updates this item when a replica is removed, and the driver replicas use the value as the parameter to the barriers used in the access patterns. This item is updated by the kernel and read by a driver.

`primary_replica_id` indicates the ID of the current primary replica. The microkernel updates this item when a primary replica migration happens. The access patterns compare this item with the exposed replica ID to identify the primary replica. This item is updated by the kernel and read by a driver.

`num_replica_changed` marks that a faulty node has been removed so that the replicas blocking on a user-mode barrier can terminate waiting and restart the barrier with the new value in `num_active_replicas`. This item is updated by the kernel and read by a driver.

`migration_disabled` suggests that an MMIO access is in progress, so the kernel halts the system if a primary migration is required. This item is updated by a driver and read by the microkernel.

When error masking is enabled for TMR, the alternative versions of the access patterns and the user-mode barriers are used. These alternatives communicate with the kernel replicas by the reading or writing the variables described above, and the memory reads and writes are relatively low-cost. For instance, the alternative version of the read access pattern is listed in Listing 5.14. `disable_migration()` sets `migration_disabled` to 1 to mark the beginning of no-migration region, and `enable_migration()` concludes the region.

> When error masking is enabled, a lightweight bidirectional communication channel is established between each driver replica and the microkernel replica for LC-RCoE. The microkernel notifies the driver replicas if the number of active system replicas changes, and the driver replicas advise the microkernel if an MMIO operation is in progress.

5.7.3 Direct Memory Access

For LC-RCoE, DMA buffers are allocated from the cross-replica shared-memory regions (Section 5.4.1). The same approach (the bidirectional communication channel) we designed for MMIO accesses also works for the DMA operations. A slight difference is that we only temporarily disable primary replica migration when writing to the DMA buffers. The reads from DMA buffers are unaffected since each driver replica can directly read the data from the buffers without relying on the primary replica to copy the data first.

For CC-RCoE, the situation becomes more complicated since that the DMA buffers are private to each driver replica and that the driver replicas depend on the system calls to replicate the input data (Section 5.5.1). However, if we remove the faulty primary replica, we lose the only replica that can access the DMA buffers that are actually used by the I/O devices. Fortunately, the actual physical memory addresses of the DMA buffers, owned by the previous primary replica and used by the I/O devices, can be derived from the new primary replica's virtual memory page table entries by the following simple calculation: `phy_addr_new_pri - start_phy_addr_new_pri + start_phy_addr_former_pri`. `phy_addr_new_pri` is a physical address in an entry of the new primary replica's page tables for DMA regions. `start_phy_addr_new_pri` is the start physical memory address of the region assigned to the new primary replica. `start_phy_addr_former_pri` is the start physical memory address of the region assigned to the former primary replica. Accordingly, we are able to patch the page tables of the drivers running on the new primary replica to restore the mappings of the DMA buffers, avoiding reprogramming the I/O devices. The patching procedure is a one-time operation for each driver, but it requires that the microkernel keep track of the virtual addresses assigned to the DMA buffers so that the corresponding entries can be patched.

On the x86 architecture, we exploit the fact that the bits 9 to 11 of a page table entry pointing to a 4 KiB physical frame are ignored by hardware [Int, 2016b], so we use the bit 9 to indicate if an entry in a page table is used for DMA buffers. We also modify the virtual memory mapping system call, in a similar way we augment them for creating the cross-replica shared memory regions (Section 5.4.1), to support the new `DMA` attribute that indicates the requested mapping is for DMA buffers. Hence, the kernel sets the bit 9 in the corresponding page table entry to 1 if the `DMA` attribute is enabled by the system call. When patching the page tables of the drivers is required, the microkernel scans the page table entries, looks for the matching entries, and replaces the entries with the computed physical addresses that previously used by the former primary replica.

> To enable the primary replication migration for CC-RCoE on the x86 architecture, the system call, `seL4_IA32_Page_Map`, is augmented with the `DMA` attribute that distinguishes mappings for DMA regions from other mappings. The microkernel exploits the 9th bit, which is ignored by hardware in a 4-KiB page entry, to indicate if the entry is for a DMA region or not. The marked DMA entries of the new primary replica are patched with actual physical addresses used by I/O devices when a migration occurs.

For the ARMv7-A architecture with the large physical address extension (LPAE), two page table formats are provided: the short-descriptor format and long-descriptor format [ARM, 2014]. The short-descriptor format does not have extra bits that can be used by system software. The long-descriptor format for blocks and pages reserves bits 55 to 58 for software use, but the long-descriptor format is not supported by our i.MX 6-based ARM board [NXP, 2015]. Thus, we do not implement error masking for CC-RCoE on ARM; but there is no fundamental obstacle to adding the support once we switch to an SoC with LPAE.

5.8 A Comparison of the Driver Support in CC-RCoE and LC-RCoE

	Cross-Replica Shared Memory	seL4_FT_Mem_Access/Rep
Performance Overhead	Moderate	High
Risks	N/A	Moderate
Implementation Complexity	Simple	Moderate
Driver Code Changes	Moderate	Small

Table 5.2: A comparison of the approaches to supporting device drivers

In Table 5.2, we compare the approaches to supporting device driver replication from the following four perspectives: performance overhead, risks, implementation complexity, and size of code changes. The cross-replica shared memory for LC-RCoE has the advantage of implementing the required functionalities in user mode, so the performance overhead of driver replication in LC-RCoE is less than that of CC-RCoE; neither does the microkernel take the risk of accessing driver-provided addresses. The in-kernel implementation for the cross-replica shared memory for LC-RCoE is also less complicated than that for the new system calls for CC-RCoE in terms of both lines of code and logic; however, the lines of driver code need to be modified for LC-RCoE are more than that of CC-RCoE.

5.9 Summary

In this chapter, we describe the access patterns that the kernel and replicated drivers must follow to support the replication of device drivers. For LC-RCoE, the system calls managing virtual memory address to physical memory address translation are augmented to support the `SHARED` option for creating cross-replica shared memory regions. This mechanism provides the essential functionality required to implement the access patterns in user mode for the operations on MMIO device registers and DMA buffers. For CC-RCoE, we supply the `seL4_FT_Mem_Access` (for accessing MMIO registers) and `seL4_FT_Mem_Rep` (for DMA buffers) system calls that implement and hide the access patterns in kernel mode, avoiding disturbing the hardware-assisted or compiler-implemented branch counters. We also introduce how the synchronisation protocol is integrated with the procedure of interrupt injection to ensure consistent observations of interrupts by all the replicas.

To support error masking, we introduce the primary replica migration: when a primary replica is faulty, we migrate functions for accessing I/O devices to a new primary replica and retire the old one. For LC-RCoE, a lightweight bi-directional channel enables the notification of removing a replica from the kernel to the drivers and the report of in-progress I/O accesses from the drivers to the kernel. For CC-RCoE, the virtual memory mapping system calls are augmented with the `DMA` option so that the kernel can identify the entries used for DMA buffers in the page tables. When the primary replica migrates, the DMA buffer entries in the new primary replica's page tables are patched by the kernel with the physical addresses used by the I/O devices. By doing so, the new primary replica is able to perform DMA buffer operations.

The mechanisms for supporting driver replication and the synchronisation protocol complement each other, and we design different solutions for LC-RCoE and CC-RCoE based on their requirements and run-time characteristics. Fundamentally, they work together to achieve the following two targets: (1) The non-determinisms introduced by I/O device interrupts and multi-threaded applications are tamed by the synchronisation protocol. (2) The non-deterministic input data from I/O devices is harnessed and replicated by the driver replication mechanisms. Therefore, the replicas of a system are able to observe non-deterministic events consistently and thus to avoid execution divergence during error-free runs.

Chapter 6

Error Detection and Masking

> And now that you don't have to be perfect, you can be good
>
> John Steinbeck

The microkernel mechanisms for redundant co-execution described in the previous chapters establish the foundation for error detection, which is the main topic of this chapter. Section 6.3 discusses two approaches to detecting errors induced by faults: (1) Section 6.3.1 describes how the *execution fingerprints* are compared independently by each replica to avoid a faulty replica being the single point of failure. (2) Section 6.3.2 illustrates how the time-out mechanism equipped by the kernel-mode barriers catches the faults manifested as execution divergence—after a predefined period of time has elapsed, if not all replicas arrive at a barrier, the timed out barrier halts the whole system. Using the execution fingerprinting mechanism for error detection is flexible; system designers can trade performance for improved error coverage and reduced error detection latency by including more states into the fingerprints and triggering comparisons more frequently, or vice versa. Finally, we conclude this chapter by examining the error masking for the triple modular redundancy (TMR) configuration, presenting the benefit and limitation. This chapter includes the work published in [Shen and Elphinstone, 2015; Shen et al., 2019].

6.1 Challenges

The properties of an error detection mechanism can be evaluated using three metrics: *error coverage*, *error detection latency*, and *performance overhead*. Generally speaking, good coverage and short latency imply increased performance overhead; and we need experiments and experiences to balance the three factors in order to build a system that fulfils design targets for error coverage and detection latency with reasonable performance overhead. Allowing system designers to experiment with various error detection configurations requires us not to predetermine error detection policies in the microkernel, but to provide flexible mechanisms that can be used to customise error coverage and error detection latency.

Error coverage is defined as the ratio of the number of errors that can be detected and the number of total potential errors. 100% error coverage means that every error in memory or architecture-visible parts of a processor shall be detected, but implementing such full coverage in software is neither practical nor necessary. Thus, the first question we need to answer is, what should be covered by the error detection mechanism? The following tasks are: How do we collect data for error detection? How do we store the collected data in a storage-efficient way that it is also straightforward to compare?

Error detection latency describes the interval from the time when an error occurs to the time when the error is detected. It is usually infeasible to measure the latency directly in software since the exact time when the error happened is unknown to software. For instance, an error can propagate from a component that is not covered by the error detection mechanism to a component that is protected so that the exact time when the original error occurred is undetermined. For this reason, we do not aim to quantify the exact length of error detection latency, but intend to estimate an upper bound for the period between when an error is captured by the error detection mechanism and when the error is actually detected by comparing the states of the replicas. Capturing an error mandates that the incorrect state be included into the execution fingerprints, and detecting the error is achieved by validating the fingerprints. The question is, how do we enforce the upper-bound error detection latency (assuming a user specifies one)?

For DMR systems, we fail-stop the systems once an error is detected since having two copies is not sufficient to conduct majority voting, and it is risky to assume that one of the replicas is correct. For TMR systems, we have the opportunity of masking an error by voting the replica affected by the error, removing the faulty replica, and downgrading the systems to the DMR mode. Each replica votes the faulty replica independently and validates its result with the results from other replicas. The error masking procedure continues only when all the replicas reach a consensus on which replica is faulty. We will examine the multi-stage voting algorithm in Section 6.5.1.

6.2 Execution Fingerprints

The per-replica (per-core) fingerprint consists of a sequence of events that are captured by the microkernel. An event is defined generally as a sequence of steps that change per-replica state, ranging from a system call invoking multiple kernel functions to a single function updating a virtual to physical memory translation table entry. Various helper functions are implemented to compress an event to a signature—a 32-bit or 64-bit integer. For example, a system call signature can be constructed by adding system call number, input parameters, and output results. Each kernel replica keeps an event counter and assigns a unique sequence number for each signature, forming a pair of `(event_count, event_signature)`.

The storage used for keeping the pairs can become significant for a long-running system, so we use Fletcher checksum [Fletcher, 1982], which is dependent on the values forming the checksum and the order of the values being incorporated, to compress the pairs. By checksumming the events, the execution fingerprint of a replica is effectively represented by a pair, `(event_count, checksum_of_event_signatures)`, which reflects the history of state updates of the replica. Not only does checksumming save storage, but also it simplifies fingerprint validation: comparing checksums is much easier than comparing captured events one by one.

> An execution fingerprint of a replica represents a compressed running history of state updates that have been performed by the replica.

6.2.1 Internal Kernel Functions and the System Call

The microkernel provides internal functions which are listed below, to manage the per-replica fingerprint and to trigger fingerprint validations; the functions are used by the microkernel to implement introspection: isolation-critical state updates of the kernel can be included into the per-replica execution fingerprint for validations.

- `add_event(value)` adds an event to the per-replica fingerprint. It is an operation entirely local to the replica, requiring no cross-replica synchronisation.
- `add_compare_event(value)` adds an event to the local fingerprint of a replica and waits until all replicas reach the same point in the sequence of state updates and then compares the fingerprints. A mismatch of the fingerprints implies the replicas are inconsistent due to an error, and our prototype triggers a graceful fail-stop for the DMR mode or a recovery (if enabled) for the TMR mode.

`add_event(value)` and `add_compare_event(value)` are interchangeable with the former being a fast local operation and the latter forcing an immediate fingerprint comparison. The system call, `seL4_FT_Add_Event(value, compare_now)`, is implemented by the two internal functions; the `compare_now` parameter controls the use of the former or the latter. This creates a trade-off between performance and frequency of fingerprint comparison for system designers to resolve as required.

The `seL4_FT_Add_Event` system call can be used by a malicious user-mode application to trigger fingerprint-comparison failures if the replicas of the application manage to break out from the virtualised runtime environment and to provide inconsistent data as the input parameters for the system call. There are two possible outcomes when parameters mismatch: (1) a failed fingerprint comparison (when the `value` parameters are different), and (2) a kernel-barrier timeout (when the `compare_now` parameters are different). Both outcomes lead to halting the system so that the the malicious application cannot use the call to violate isolation guarantees. Furthermore, we can reserve the call to trusted applications and device drivers only, reducing the possibility of abusing the system call.

6.2.2 What Should Be Included in Execution Fingerprints

The microkernel is modified to collect security- and isolation-related kernel-state updates into the per-replica fingerprint by using the `add_event` and `add_compare_event` functions. Our default implementation includes the following updates into the fingerprint unconditionally.

Virtual address to physical address translation table updates affect the isolation between system replicas, between user-level components and the kernel, and between trusted and untrusted applications within a replica. The MMU of the processor walks the multi-level tables to translate a virtual memory address to a physical memory address. Although the formats of the table entries are architecture-dependent, the virtual address, physical address, and the attributes of the mapping are common properties. We combine these values to form a signature for each update and call `add_compare_event` to include the signature into the fingerprint for validation before returning to user mode. We collect the offsets (physical_address - start_physical_address_of_the_replica) instead of absolute physical addresses since the physical memory regions for the replicas are different.

Capability space updates affect the distribution of authority and the isolation boundaries enforced by the microkernel. A capability space is a single- or multi-level table, and each slot in the table represents a capability. One or more tables can be modified by a system call to manipulate the states or ownerships of the capabilities. For each capability space operation, we collect the source and destination entries as well as the type of operation into the fingerprint and trigger a validation before returning to user mode.

We modify the microkernel to include all capability space and virtual memory space updates that are security- and isolation-critical into the execution fingerprints by calling the internal functions.

From the microkernel's point of view, system calls issued by applications are inputs to the kernel and the return values from the kernel are the outputs. Such input/output data should be consistent (except for the system calls that are reliant on precise preemption), so the system calls and the associated data can be included into the fingerprints to detect any divergence. We introduce a configuration option so that the kernel can be built to incorporate system calls for error detection at the expense of slightly increased performance overhead, which will be demonstrated in the evaluation. The option provides the flexibility of increasing or reducing error coverage.

Inputs and outputs for each system call can be optionally included into the fingerprints for improved error coverage.

From the whole system's point of view, incoming data from I/O devices is the input; the system processes the input data and generates output data which is then released to I/O devices for transmission. Therefore, user-mode drivers should verify the output data. For this reason, we allow the drivers to contribute their output data or key internal state changes into the fingerprints via the new system call: `seL4_FT_Add_Event(value, compare_now)`. The drivers can provide the checksums of the output data as the first parameters. If `compare_now` is `true`, the calling replicas add the first parameter to the fingerprints and wait on the kernel barrier dedicated for this system call; when all replicas arrive at the barrier, the fingerprints can be validated. When `compare_now` is `false`, the calling replicas simply append the first parameters to their fingerprints and continue execution. The microkernel does not know if a data region is important to an application, so in order to increase the error coverage, system designers can employ the same system call to incorporate application-specific data or checksums into the fingerprints for validations.

We also allow the user-mode programs to adjust the frequency of fingerprint validations. This system call, `seL4_FT_Add_Event`, can also be used with zero as the first parameter and `true` as the second parameter, triggering a fingerprint comparison explicitly. We can set up periodic comparisons by programming a hardware timer to generate recurrent interrupts and calling the system call from the interrupt-handler thread. In this case, a user-mode application can increase or decrease the comparison frequency directly without tuning kernel configuration options.

> A multi-purpose system call, `seL4_FT_Add_Event(value, compare_now)`, is implemented. Device drivers and applications can use the system call to incorporate safety- or security-critical data into the execution fingerprints for validations, increasing the error detection coverage. The system call can also be used to trigger fingerprint validations explicitly, allowing user-mode programs to adjust the validation frequency and thus error detection latency.

6.3 Error Detection

The instrumented microkernel relies on two methods for error detection: *fingerprint comparisons* and *timeouts of kernel barriers*. The fingerprints of all the replicas are checked by each replica independently to avoid a replica being the single point of failure; a transient-fault-triggered error that propagates to a per-replica execution fingerprint can be detected by the comparisons. Kernel-barrier timeouts detect control-flow divergence introduced by transient faults by setting proper timeout values for the synchronisation barriers and functional barriers. Choosing appropriate timeout values requires the knowledge of worst-case execution time of the applications to be deployed: inappropriate values would either increase the window of vulnerability or halt the system unnecessarily.

6.3.1 Error Detection by Comparing Fingerprints

As described in Section 6.2, the execution fingerprint of each replica summarises how the important kernel and application states of the replica evolve over a period of time. The code for validating the fingerprints is shown in Listing 6.1, and it is executed redundantly by all the replicas. The barriers (at lines 2 and 6) are used to ensure that the fingerprints are not modified when the replicas are comparing them. Admittedly, the code is straightforward and uninteresting, but the simplicity comes from the thorough preparation of redundant co-execution and fingerprint collection.

```
1 ret = FAIL;
2 kbar_waitn(&bar_cmp, FT_REPLICAS);
3 if (FT_REPLICAS == 2 && fingerprint[0] == fingerprint[1]) ret = OK;
4 if (FT_REPLICAS == 3 && fingerprint[0] == fingerprint[1] &&
5     fingerprint[1] == fingerprint[2]) ret = OK;
6 kbar_waitn(&bar_cmp, FT_REPLICAS);
7 return ret;
```

Listing 6.1: Fingerprint co-comparison

The `fingerprint` is an array in the shared kernel data region so that each replica's fingerprint can be accessed by other replicas. For DMR systems, we halt the systems if one of the replicas returns `FAIL` result. However, the situation for TMR systems is more complicated if we allow error masking, so we will discuss the possible outcomes in Section 6.5.

6.3.2 Error Detection by Kernel Barrier Timeouts

As shown in the SPIN model (Appendix B), several kernel-mode barriers (the S-barriers) are used to coordinate the redundant co-execution. Furthermore, more barriers (the F-barriers) are also used in the seL4_FT_Mem_Access, seL4_FT_Mem_Rep, seL4_FT_Add_Event, and seL4_IA32_IOPort system call handlers. These barriers can be configured with the timeout feature: a kernel barrier halts the system if not all replicas arrive at the barrier after a specified amount of time has elapsed since the first replica arrived at the barrier. The timeout value is supplied as a kernel-configuration option, and it is in the unit of CPU cycles. We use the CPU cycle counters for the following reasons: (1) each core has its own cycle counter so that accesses to the counter do not interfere with other cores; (2) accessing a cycle counter is a relatively cheap operation compared with accessing other platform-specific timers; (3) the timeout values are usually shorter than the architecture-specific counter overflow intervals. However, users have to convert the timeout value from absolute time to CPU cycles so that recalculation is required for CPU running at different frequencies. Choosing appropriate timeout values for the barriers is non-trivial and potentially system-specific; we consider it as a future work (Section 8.2.6).

6.3.3 Bounding the Error-Detection Latency

The comparison frequency determines error-detection latency. Although we provide system designers the flexibility to meet their error detection latency goals by increasing or decreasing the intervals between comparisons with the seL4_FT_Add_Event system call, we also provide an option that enables periodic fingerprint comparison if applications are not modified to take the advantage of the new system call. The observation is that we already need to synchronise the replicas for handling an interrupt, so the fingerprints of the replicas must be consistent after the synchronisation and before injecting the interrupt for error-free runs. Thus, we can compare the fingerprints right before injecting interrupts, without introducing too much additional overhead (e.g., synchronising the replicas only for fingerprint comparisons). Remember that the periodic kernel preemption timer interrupts are treated by the same mechanism, so we can optionally configure the system to check the fingerprints at every preemption-timer tick. The length of the ticks is configurable so that the comparison frequency can also be adjusted accordingly, and the default value is 20 milliseconds (50 ticks per second). For systems with a very high interrupt frequency (e.g., a network server with thousands of network interrupts per second), the checking frequency is dominated by the rates of device interrupts, and we suggest disabling the preemption-timer-triggered comparisons for reduced performance overhead, since the fingerprints are compared by default for each I/O device interrupt.

6.4 Turning the Knob: Coverage, Latency, and Overhead

In this section, we demonstrate how the supplied primitives are used to build systems with different levels of error coverage, error-detection latency, and runtime performance overhead. The principle is to reduce built-in kernel policies as much as possible, but we still safeguard the microkernel with two default rules: (A) Updates to the capability spaces and virtual spaces are included into the execution fingerprints, and each update triggers a comparison immediately so that the update can be verified against with other replicas' updates before returning to user mode. (B) Execution fingerprints of all replicas are compared when the system is synchronised for each device interrupt (other than the kernel preemption timer interrupts). The default rules can be removed easily by reconfiguring the microkernel building options if designers really want to do so. We also strongly recommend that driver writers checksum the output data and validate the data with the `seL4_FT_Add_Event` system call before releasing the data to I/O devices. We present several typical configurations with different degrees of error detection capabilities and overhead. We list several basic error detection modes below, and system designers can configure systems to use the combinations of the basic modes. Please note that the fingerprints are implicitly verified according to rule (B) in all modes.

Barrier-Timeout (BT) Mode only halts a system when replicas diverge or one of the replicas stops making progress. No state updates are included into the fingerprint so that the comparisons are turned off. This execution mode introduces the least runtime overhead, but it is ineffective in catching data corruptions unless the corruptions cause execution divergence that can be captured by the kernel barriers.

Critical-Update (CU) Mode includes CSpace and VSpace updates into the execution fingerprints, and each update triggers a validation to check the integrity and validity of the update immediately before the kernel replica returns to user mode. The CU mode only verifies updates that are critical to preserve the isolation enforced by the microkernel. However, these updates are relatively infrequent once a system is completely initialised.

Application-Contribution (AC) Mode incorporates application-contributed data into the execution fingerprints and allows applications to explicitly trigger a fingerprint validation whenever they see fit. The mode significantly expands the fingerprints to include arbitrary application data for validation. The increase of runtime overhead depends how frequently the `seL4_FT_Add_Event` is called and how often the `compare_now` option is true.

System-Call (SC) Mode incorporates system call numbers, inputs, and outputs into the execution fingerprints; but each system call is not checked immediately before the kernel returns to user mode. Since applications communicate through IPC, this mode also captures data interchanges between applications into the fingerprints.

System-Call Number (SCN) Mode only collects system call numbers into the execution fingerprints, omitting call inputs, call outputs, and IPC buffers; each call is not checked immediately before the kernel returns to user mode.

System-Call Immediate (SCI) Mode is similar to the SC mode above; in addition to capturing system call data into the fingerprints, this mode synchronises the replicas for each system call before the replicas return to user mode and the compares the fingerprints immediately. In other words, the replicas are "lockstepped" at the system-call level. Obviously, the SCI mode incurs higher overhead than the SC mode if other conditions are the same.

From the simple BT mode to the most complicated and comprehensive BT+CU+AC+SCI mode, the designers are able to combine the basic modes and tailor the error-detection mode that suits their needs best. Among the combinations, we recognise that the CU + SC + BT + AC mode achieves a sweet spot balanced with good coverage, low latency, and moderate performance overhead. On the other hand, the CU + SCI + BT + AC mode delivers the most extensive protection at the cost of high overhead. In Chapter 7, we illustrate various performance characteristics of several error detection modes.

6.5 Error Masking

With the TMR (triple modular redundancy) configuration, the microkernel is able to recover from an error without service disruptions, provided that the security policy allows recovery and that the system state is eligible for conducting the operations of removing a faulty replica.

Two approaches are commonly used for error masking: backward recovery and forward recovery. Backward recovery rolls back a system to a previously saved and checked state, so it requires periodic checkpointing for taking snapshots of the system state as well as reliable storage for ensuring the integrity and correctness of the saved snapshots. The checkpointing frequency and the amount of data to be saved for each checkpoint largely determine the performance overhead of backward recovery. One observation is that the checkpoint data is usually proportional to the interval between checkpoints. In terms of our whole-system replication approach, a less optimised approach would produce a checkpoint by taking a snapshot of per-core memory regions, increasing the memory usage further. While several optimisations could be applied, such as saving only one replica's memory to a globally accessible storage, incrementally saving the memory changes since the last checkpoint, and compressing checkpoints, we still need to consider the possibility of the checkpointing process being affected by faults, resulting in incorrect checkpoints. In spite of increased memory and computation overheads, backward recovery provides the opportunity to restore a faulty replica to a previously correct state and to keep running the system in DMR or TMR configurations without losing a replica. Furthermore, backward recovery does not need to identify which replica is faulty since it can simply revert all replicas to a former state, so it is possible for a DMR system to deploy backward recovery.

Forward recovery relieves the needs for periodic checkpointing, but it requires TMR and majority voting to reach a consensus on which replica is faulty. Then, the recovery algorithm can decide to reintegrate a faulty replica by copying data from a correct replica or bringing up a spare replica into service, retaining TMR; or it can choose to downgrade the system to DMR, reducing implementation complexity. We adopt forward-recovery-based error masking by using majority voting and dynamically downgrading to DMR for the following reasons: (1) Our redundant co-execution already significantly increases pressure on the memory subsystem bandwidth (see Section 7.3.1) and doubles or triples memory consumption. (2) The reintegration process is not redundantly executed and checked, neither is the process of copying or modifying important kernel data structures (e.g., virtual memory and capability tables), introducing a window of vulnerability. Although we consider reintegration as future work, we will outline the steps should be performed in Section 6.5.3. (3) Fail-stop is an acceptable, sometimes even preferred method, if reliable error masking cannot be achieved.

For a TMR system, the error masking algorithm described in Listing 6.2 and Listing 6.3 attempts to remove the faulty replica and downgrades the system to DMR, without service interruption or reboot. Since we do not support the reintegration of the faulty replica, a full system reboot is required to restore TMR if the system is downgraded to DMR. Another limitation is that the kernel can only initiate recovery when an error is detected by the fingerprint comparisons; a kernel barrier timeout halts the whole system without attempting to recover. For a DMR system, should an error be detected, we halt the system instead of trying to mask the error. Please refer to Section 5.7 for the underlying kernel mechanisms supporting primary replica migration.

> Error masking is attempted for TMR systems and achieved by voting the faulty replica and removing the replica from the system, implementing a forward-recovery approach.

6.5.1 The Algorithm for Voting A Faulty Replica

The voting algorithm in Listing 6.2 is specialised for our scenario, and we designed it in a way that it can fulfil the following requirements: (1) if only one checksum is inconsistent with other checksums, the replica ID of the incorrect checksum is returned; (2) an error is returned for all other scenarios, which include all the checksums are the same, all the checksums are different, etc. The algorithm is only called when the checksums mismatch, so it reports an error when all the checksums are the same. We aim to fail-stop the system if the replicas fail to reach a consensus, no matter what the underlying causes are (the checksums are corrupted, two or more replicas misbehave, the code region for the voting algorithm is corrupted, etc.). Although our TMR configuration can only tolerate one faulty replica if certain conditions are met, the voting algorithm does not assume that only one replica is faulty. This is because an SEU can affect more than one replica, as described in Section 4.1.4. We want to emphasise that the algorithm is executed by all the replicas independently, and the arrays ft_votes and ft_fault_replica are in shared kernel memory region so that they can be accessed by all replicas.

```
global_shared int ft_votes[NUM_REPLICAS];
global_shared int ft_fault_replica[NUM_REPLICAS];

int vote_fault_replica(void) {
  int least_vote = NUM_REPLICAS + 1;
  int fault_replica = NUM_REPLICAS + 1;
  ft_votes[my_replica_id] = 0;
/* If the checksum of the current replica is the same as
   another replica's, the current replica receives one vote.
   A replica gets at least one vote from itself. */
  for (int i = 0; i < NUM_REPLICAS; i++) {
    if (checksum[i] == checksum[my_replica_id])
      ft_votes[my_replica_id]++;
  }

  kbarrier(bar, NUM_REPLICAS);
/* Now we count which replica receives the least votes */
  for (int i = 0; i < NUM_REPLICAS; i++) {
```

```
19     if (ft_votes[i] < least_vote) {
20       least_vote = ft_votes[i];
21       fault_replica = i;
22     }
23   }
24 /* If the current replica does not receive enough votes,
25    it marks itself as the faulty replica. Otherwise, the
26     replica with the least votes is marked. */
27   if (ft_votes[my_replica_id] != NUM_REPLICAS - 1)
28     ft_fault_replica[my_replica_id] = my_replica_id;
29   else
30     ft_fault_replica[my_replica_id] = fault_replica;
31 
32   kbarrier(bar, NUM_REPLICAS);
33 /* Check if all the replicas reach a consensus. */
34   for (int i = 0; i < NUM_REPLICAS; i++) {
35     if (ft_fault_replica[i] != ft_fault_replica[my_replica_id]) {
36       return ERROR_DIFF_FAULT_REPLICA;
37     }
38   }
39   kbarrier(bar, NUM_REPLICAS);
40   return fault_replica;
41 }
```

Listing 6.2: The algorithm for voting a faulty replica

The function, `vote_faulty_replica`, first compares each replica's checksum with other replicas' and increases the per-replica counter `ft_votes[my_replica_id]` if the checksums match (lines 11-14). The barrier at line 16 ensures all replicas finish before proceeding to the next stage, or halts the system if the barrier timeouts. Lines 18–23 find out the smallest value in the array `ft_votes`; the values in the array represent how many checksums in the array `checksum` are the same as one indexed by `my_replica_id`. Thus, the replica has the smallest value is the faulty one. Lines 27 to 30 further check for the cases that more than one of the replicas are faulty and that all the checksums are the same: the votes received by each non-faulty replica should be the replica number (`NUM_REPLICAS`) minus one if only one checksum is incorrect. Each replica stores the faulty replica ID in the globally shared, per-replica variable `ft_fault_replica[my_replica_id]` if the check succeeds; otherwise, each replica stores its ID. The second barrier at line 32 ensures that all the replicas have finished the checking stage. Finally, all the replicas check if the faulty replica voted by others is the same as the one chosen by itself. An error is returned if the faulty replica IDs are different, and the system halts (lines 34–38). If all replicas agree on the faulty replica, they pass the third barrier and the faulty replica ID is returned by the function (lines 39–40). The three barriers play important roles in robust error handling through fail-stop. It is worth mentioning that we use the function when `NUM_REPLICAS` is 3, but the algorithm can be used to vote a faulty replica when `NUM_REPLICAS` is greater than 3.

Now let us see how the voting algorithm works with examples in Table 6.1. The first two examples are for three replicas, and the other two are for four replicas. In the first example, R_2

	R_0	R_1	R_2	R_3
checksum	0xdeadbeef	0xdeadbeef	0xdeedbeef	N/A
ft_votes	2	2	1	N/A
ft_fault_replica	2	2	2	N/A
checksum	0xdeadbeaf	0xdeadbeef	0xdeedbeaf	N/A
ft_votes	1 (<2)	1 (<2)	1 (<2)	N/A
ft_fault_replica	0	1	2	N/A
checksum	0xdeadbeef	0xdeadbeef	0xdeedbeef	0xdeedbeef
ft_votes	2 (<3)	2 (<3)	2 (<3)	2 (<3)
ft_fault_replica	0	1	2	3
checksum	0xdeadbeef	0xdeadbeef	0xdeedbeef	0xdeedbeaf
ft_votes	2	2	1	1
ft_fault_replica	0	1	2	3

Table 6.1: Examples of the voting algorithm

has the incorrect checksum and the least `ft_votes` value, so R_2 has been voted by all replicas as the faulty node. In the second example, all the checksums are different so that the `ft_votes` values are all 1. In this case, each replica set its `ft_fault_replica` to its ID, indicating multiple faulty replicas. The third example shows that the replicas are split into two equally-sized sets and that each replica is unable to receive enough votes, so an consensus cannot be achieved. The last example demonstrates that the last two replicas are faulty so that the replicas fail to reach an agreement.

6.5.2 Removing A Faulty Replica

```
global_shared int num_replicas;
global_shared int new_primary_replica;
global_shared int cur_primary_replica;
global_shared int former_primary_replica;

/* rid is the id of the replica to be removed */
void remove_fault_replica(int rid) {
  int num_replicas = num_active_replicas;
  if (is_in_lcrcoe() && rid == cur_primary_replica &&
      !migration_allowed()) {
      /* migration is not allowed, abort */
      halt();
  }
  if (rid == cur_primary_replica) {
    /* if the current primary replica is faulty, *
     * we need to choose a new primary replica.  */
    new_primary_replica = choose_new_primary_replica();
    kbarrier(bar, num_active_replicas);
    /* redirect interrupts to the new primary replica */
20      if (my_replica_id == new_primary_replica) {
21        reroute_interrupt(new_primary_replica);
22        /* perform platform-specific operations */
23        ...
24        former_primary_replica = cur_primary_replica;
25        cur_primary_replica = new_primary_replica;
26      }
27    }
28    kbarrier(bar,  num_active_replicas);
29    /* now we remove the faulty replica */
30    if (rid == my_replica_id) {
31      if (is_former_primary_replica()) {
32        disable_kernel_timer();
33        /* save pending interrupts */
34        while (irq = interrupt_pending()) {
35          save_pending_interrupt(irq);
36          mask_interrupt(irq);
37          ack_interrupt(irq);
38        }
39      }
40      node_data[my_replica_id].status = FT_NS_FAULT;
41      num_active_replicas--;
42      /* the faulty replica loops forever */
43      infinite_loop();
44    }
45    /* other replicas spin until the faulty replica removes itself */
46    while (num_replicas == num_active_replicas);
47    /* for LC-RCoE, threads must be notified about the changes. */
48    if (is_in_lcrcoe()) { notify_threads(); }
49    /* for CC-RCoE, we need to patch page tables if a primary     *
50     * replica migration happened so that the new primary replica *
51     * can access DMA buffers.                                    */
52    if (is_in_ccrcoe() && is_primary() && is_primary_switched()) {
53      patch_page_tables();
54    }
55    kbarrier(bar, num_active_nodes);
56    return;
57  }
```

Listing 6.3: Pseudo code for removing a faulty replica

The function, `remove_fault_replica` (Listing 6.3), actually retires a faulty replica; it also chooses a new primary replica and reroutes interrupts if a primary replica is the one to be removed. The replica ID to be removed is passed as the parameter `node`. All the replicas first make a local copy of the global shared variable `nun_active_nodes` that represents the number of active replicas. The value of `num_active_nodes` is decreased by one at line 41 to indicate the completion of removing the faulty replica, so other replicas can exit the loop at line 46. If the system is managed by LC-RCoE and the primary replica is to be removed, the replicas check if primary replica migration is allowed (lines 9–11). When the primary replica is voted as faulty, `migration_allowed()` scans all threads in the run queue to determine if a device driver thread, be it running or runnable, is in the middle of accessing device registers, The system will be halted at line 12 if the migration is not allowed (see Section 5.4). Lines 14–27 handle the case of removing the current primary replica (`cur_primary_replica`) and switching to a new primary replica (`new_primary_replica`) by performing the following steps: (1) choose a new primary replica; (2) reroute interrupts to the newly selected primary replica; (3) perform platform-specific operations.

Lines 30–44 remove the faulty replica, and we do assume that the removal process is not affected by errors. If the replica is the former primary replica (lines 31–39), the replica stops its own kernel preemption timer and saves pending interrupts, if any, for processing. Finally, the faulty replica marks itself as faulty (line 40), decreases `num_active_replicas`, and enters an infinite loop. By doing so, the faulty replica stops execution. The replicas waiting at line 46 now can proceed after the faulty replica has removed itself. For LC-RCoE, all device driver threads need to be notified about the changes (line 48). The `notify_threads()` function iterates through all threads and notifies the driver threads through the bidirectional notification channels described in Section 5.7.2. For CC-RCoE, as described in Section 5.7.3, the page tables of the device driver threads must be patched when a primary replica migration happens (lines 52–54); so the new primary replica is able to access DMA buffers.

6.5.3 Reintegration

For a TMR system, if a faulty replica is removed, we downgrade the whole system to DMR, without attempting to reintegrate a new replica or the faulty replica (if the faulty replica can be repaired) into the system and to restore operations in TMR. Although reintegration might not be a critical feature, it is definitely desirable for systems unattended for an extended period. We identify the following steps to reintegrate a core into the system. We adopt a "stop-the-world" approach since no specialised hardware, as in [Bernick et al., 2005], is available to monitor memory traffics and conduct memory copy in background.

- If we build a new replica from a spare core, we first activate the core and set the core's page directory address to the faulty replica's address and put the core in idle mode. After that, the core of the faulty replica is deactivated. We call the new replica as the *target replica*.
- Choose a valid replica as the *source replica*; we reconstruct and repair the state of the target replica based on the source replica.
- Compare the kernel page table entries of the target replica with the entries of the source replica and correct the entries that are corrupted. We check the attributes of the entries as well as the relative addresses (Section 4.2.3). Correcting the physical address in an erroneous entry can be done by using the relative address of the corresponding entry from the source replica plus the start physical address of the target replica.

- Compare the checksums of the kernel code section. If the checksums mismatch, we copy the code section for the source replica to the target replica.
- Compare the kernel variables in kernel data section (ksCurThread, ksReadyQueue, etc.) and correct the variables of the target replica if necessary.
- For each user-mode thread, we conduct the following inspections and repair a thread if necessary.
  - Compare the data in the thread control block. This step verifies the kernel-mode state of the thread.
  - Compare the CSpaces of the source and target replicas and make corrections accordingly. This step verifies all the capabilities owned by the thread.
  - Compare the VSpaces of the source and target replicas. This step checks the validity of the entries in the page directory of the thread.
  - Compare the user-mode code, data, and stack sections. This step aims to correct any user-mode corrupted data.
- Having verified all user-mode threads, now the replicas can reset kernel data used by redundant co-execution and resume execution.

The steps described above provide a rough guideline for implementing reintegration; we have not realised the steps in C code yet. The main obstacle is that the steps can take a long time since they perform systematic checks on the target replica to be reintegrated, but the checking steps are not protected by redundant co-execution; thus the new replica may contain errors. A more dangerous scenario is like this: (1) The source replica is affected by a transient fault (e.g., a single-bit flip in a page table entry) while checking and repairing the target replica, and (2) the repair steps find a mismatch and copy the corrupted data from the source replica to the target replica. In this case, we have two faulty replicas becoming the majority so that the redundant majority voting may wrongly determine the only valid replica as the faulty one. Although such scenario is low probability, the consequence can be disastrous. Thus, ensuring that the reintegration process is being executed correctly and that the new replica is error-free is an important research topic for future work.

6.6 Summary

In this chapter, we introduce the concept of the execution fingerprint that captures updates made by a replica. Kernel internal functions (`add_event` and `add_compare_event`) and the system call (`seL4_FT_Add_Event(value, compare_now)`) are provided so that system designers can tune the error coverage by increasing or decreasing the state updates included in the execution fingerprint. Each replica compares its execution fingerprint with the fingerprints of other replicas independently as the first error detection mechanism. To cater various needs for tuning error detection latency and performance overhead, setting the `compare_now` parameter of the system call above to `true` triggers an immediate fingerprint comparison, so user-mode applications can adjust the comparison frequency. However, the kernel still checks the fingerprints for each I/O device interrupt as a default setting so that error detection is in operation even if the applications do not trigger comparisons explicitly. The second mechanism for error detection is using timeouts of the kernel barriers to uncover execution divergence of the replicas. The kernel barriers used for different purposes require different timeout values, and some of the values depend on worst-case execution time of the applications running on the system. Determining accurate timeout values for the kernel barriers is left as future work Section 8.2.6. We also introduce a list of error checking modes with various levels of error detection capabilities and overheads, demonstrating that our mechanisms can be straightforwardly tailored to diverse scenarios.

Error masking is only available for TMR systems, and the kernel only tries to recover from an error if the error is detected by the fingerprint comparison mechanism. The voting algorithm for choosing the faulty replica is discussed, and we also show that the algorithm does not assume an error-free execution environment. However, the procedure to remove the faulty primary replica is neither redundantly executed nor checked.

Chapter 7

Evaluation

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald E. Knuth

The aims of the chapter are fourfold: (1) Investigate to what extent redundant co-execution affects the performance of CPU-bound, memory-bound, and I/O-bound benchmarks. (2) Demonstrate that our approach is applicable to real-world workloads by benchmarking Redis server, an in-memory key-value data store, with a mix of memory loads and stores, I/O operations, and computations. (3) Validate that our approach can detect errors induced by transient faults effectively by conducting fault-injection experiments on real hardware. (4) Show that the microkernel primitives are elastic and accessible for building systems with different error-coverage or error-detection-latency targets. This chapter is based on the work published in [Shen et al., 2019].

7.1 Benchmark Configurations

We implement our approaches on both ARM and x86 architectures, and the hardware platforms used for benchmarks and fault injection experiments are listed in Table 7.1. We term the benchmarks running directly on seL4 and invoking seL4 system calls as *seL4-native benchmarks*, and the benchmarks executing inside a Linux VM and exercising Linux sytem calls are called as *VM benchmarks*. In the case of the replicated Linux virtual machines, we list the guest VM software environment in Table 7.2.

In Table 7.1, the sizes of L1 and L2 caches for the Core i7 6700 are for each core; the L3 cache is shared by all four cores. The L1 caches use separate instruction cache and data cache (64 KiB in total for each core), and the L2 and L3 caches are unified. The L1, L2, and L3 caches are protected by parity, single-bit ECC, and multi-bit ECC respectively. As to the i.MX6 quad-core SoC, each core features 32 KiB data cache and 32 KiB instruction L1 cache; the 1 MiB L2 cache is unified and managed by the PL310 cache controller from ARM.

For x86 processors, I have implemented and tested our approaches (both LC-RCoE and CC-RCoE) on four different processor models (Core i7-2600, Core i5-4590, Core i7-4770, and Core i7-6700) spanning three microarchitectures (Sandy Bridge, Haswell, and Skylake). LC-RCoE works on all four models, and CC-RCoE is supported by Core i5-4590, Core i7-4770, and Core

	Sabre Lite	Dell
CPU/SoC	NXP i.MX6	Intel Core i7 6700
uArch	ARM Cortex-A9 ARMv7	Skylake
Cores	4	4 (Hyper-Threading off)
Clock	1 Ghz	3.40 GHz (Turbo Boost off)
L1	4×32 KiB D/I	4×32 KiB D/I
L2	1 MiB	4×256 KiB
L3	N/A	8 MiB
Memory	1 GiB DDR3-1006	4 GiB DDR4-2133×2
NIC	1000 Mbit/s ENET	Intel I219-LM

Table 7.1: Hardware platforms (seL4 native/VM benchmarks)

i7-6700. We are able to run replicated Linux VMs managed by hardware-assisted CC-RCoE on the three models; the results of the relatively recent model, Core-i7 6700, are presented in the following sections.

Software	Version	Notes
Kernel	Linux v4.10.10	built with GCC 6.3.0
RootFS Image	Buildroot 16/04/2017 Git	target_arch: i386, arch_variant: i586
C library	musl libc v1.1.16	shared libary
Init System	Busybox v1.26.2	
Ramspeed	v2.6.0	memory bandwidth benchmark
Dhrystone	v2.1	integer benchmark
Whetstone	v1.2	floating-point benchmark

Table 7.2: Linux virtual machine software configurations

We choose the i.MX6 quad-core SoC [NXP, 2015] for the following reasons: (1) It is the first official ARM-v7 platform supported by the verified seL4 kernel. (2) The user-mode device drivers for the network card, serial port, and timer have been implemented and tested by our colleagues. (3) The SoC is equipped with an 1 Gbps ENET network interface, ensuring that the network link is not the bottleneck. According to the chip errata [NXP, 2016], the actual performance is limited to 400 Mbps, which is still higher than the peak network throughput when running the Redis benchmark on this platform.

7.2 A Case Study: Adapting CC-RCoE to Support Linux VM Replication

Before we present the benchmarking results, we describe how we use hardware-assisted CC-RCoE (Section 4.4) to replicate a Linux virtual machine as a case study, providing necessary background for the VM benchmarks. Running virtual machines requires less effort to deploy mature or legacy applications than porting the applications to a new operating system. For binary-only applications written for other operating systems, deploying them in a virtual machine could be the only viable or affordable way to reuse them. Being able to support Linux virtual machines on x86 greatly expands the scenarios that the seL4 microkernel can be used while still benefiting from the strong isolation guarantee provided by the kernel.

As introduced in Section 3.2.3, existing virtual machine replication approaches [Bressoud, 1998; Cully et al., 2008; Scales et al., 2010; VMware, 2015] focus on providing high availability instead of mitigating the SEU fault model. Neither the hosting operating systems nor the virtual machine monitors are replicated and redundantly executed. Moreover, output data from a primary VM is transferred to a backup VM without validating the integrity of the data, opening a door for transient-fault-induced data corruptions.

We describe how we adapt hardware-assisted CC-RCoE to support replication of a virtual machine and virtual machine monitor. Certainly, the underlying microkernel is also duplicated. Replicating the Linux kernel is much more complicated than replicating native seL4 applications because of the huge code base and complexity of the Linux kernel. In addition to the code base, it is unlikely that we are able to examine all applications running inside a VM to ensure that they are data-race-free. Although the Linux kernel source code is available, we aim to treat the Linux kernel as a black box and avoid modifying its code. The resulting system is shown in Figure 7.1. The orange boxes represent the Linux virtual machines and applications running inside the VMs. The blue boxes are native seL4 applications: The VMM is the virtual machine monitor that creates, starts, and manages the Linux VM; and the native drivers for the network card and serial port are in the boxes labelled with *Drivers*. We customise the configuration options (i.e., make menuconfig) of the Linux kernel to remove unused device drivers and to enable the VIRTIO [VIRTIO-v1.0, 2013] network driver as the front-end for sending/receiving network packages to/from the native network card driver.

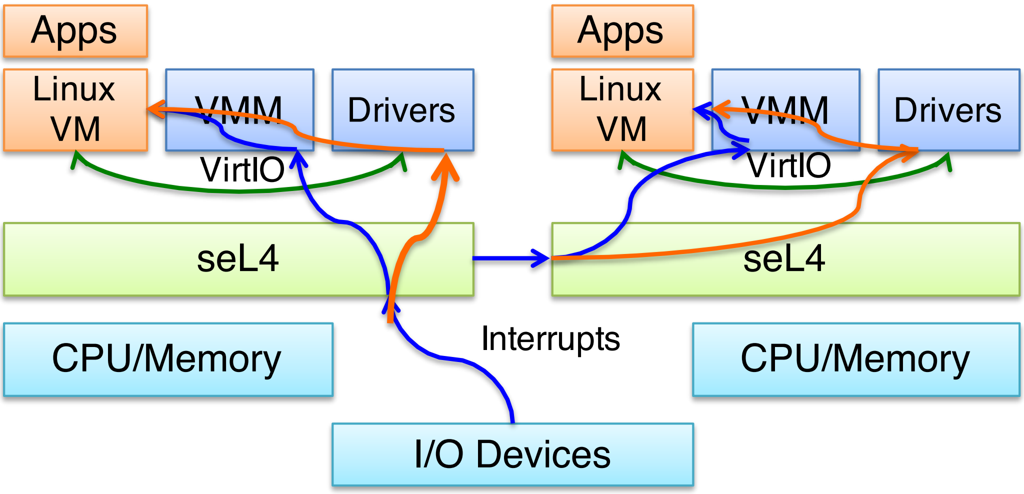

Figure 7.1: Replicated virtual machines

7.2.1 Adapting Hardware-Assisted CC-RCoE

Hardware virtualisation support [ARM, 2014; Int, 2016b] is required to launch virtual machines on seL4. We provide a brief introduction to x86 hardware virtualisation technology in Section D.6 for reference. In addition to the kernel objects described in Section 2.4, the seL4 extension for hardware virtualisation introduces a *VCPU* object type to represent a virtualised CPU assigned

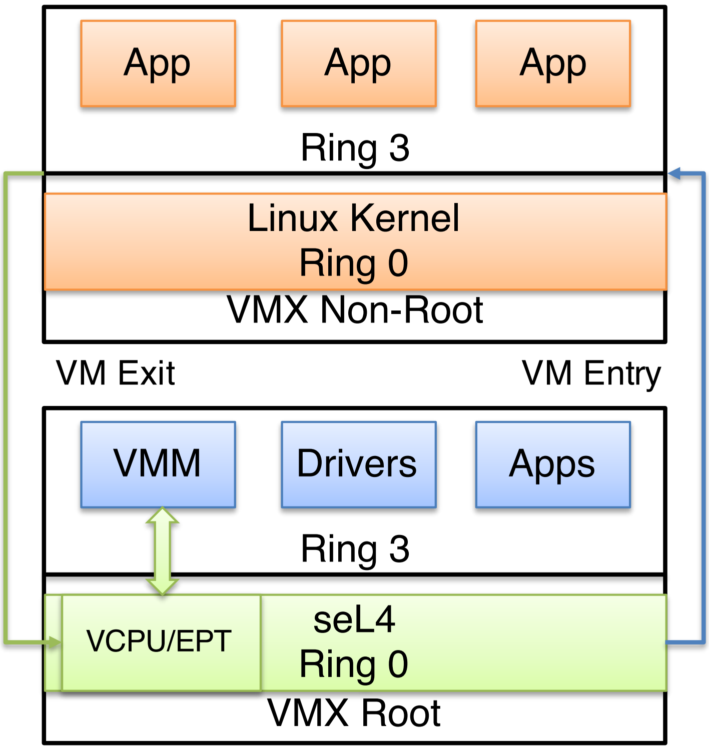

Figure 7.2: The architecture of an seL4-based Linux VM

to a virtual machine and *extended page table objects* used to control the second-level address translations (from guest physical addresses to host physical addresses). The seL4 kernel changes to support hardware virtualisation, the general user-mode libraries for building a VMM (virtual machine monitor), and the VMM were developed by our colleagues; I took an usable snapshot of the project as the base for my development.

The architecture of the Linux VM system and the privilege levels of the components are shown in Figure 7.2. All the components in the figure are running on a single core and are supposed to be replicated for redundant co-execution. The virtual machine extension (VMX) for x86 introduces two VMX operations: VMX root operation and VMX non-root operation. Each operation has complete ring 0 to ring 3, but the system software running in ring 0 of the non-root operation does not have full control over the processor's behaviour. The seL4 kernel runs in the ring 0 of the root operation, so it is the most privileged system software. The VMM manages the life-cycle of the virtual machine by manipulating the VCPU and EPT kernel objects, and it runs in the ring 3 of the root operation. The guest Linux kernel and applications execute in ring 0 and ring 3 of the non-root operation respectively. The execution of software in the non-root operation is monitored by the VMM, so are the accesses to hardware resources. Native seL4 applications and drivers run in ring 3 of the root operation as normal, and the device drivers provide services for the Linux VM and VMM.

As we do not assume the Linux kernel or the applications running inside the VM are data-race-free, device interrupts for the VM replicas must be delivered precisely and consistently to avoid divergence. In fact, even if the Linux kernel and the applications were data-race-free, it would be expensive to track locking and unlocking operations since VM exits are expensive [Agesen et al., 2012]. In addition to the system calls issued by native seL4 applications, some VM exits triggered by certain instructions (e.g., IO port instructions, HLT, CPUID, RDTSC, etc) or exceptions (e.g., page faults) can also be considered as deterministic events. The event counters are also incremented by these VM exits. One simple approach to ensuring precise and consistent injection of interrupts is to buffer all interrupts and inject the interrupts only when seL4 replicas are handling deterministic VM exits and when the event counters are the same. While the interrupts are injected consistently, the approach suffers a significant issue – a buggy or malicious application in the VM can stop the whole VM by simply looping. The looping application prohibits the deterministic VM exits so that no interrupts can be injected to the VM, and thus the Linux kernel is unable to get back in control and to stop the application. Because of this issue, we abandon the approach that only injects interrupts at deterministic VM exits.

From the seL4 kernel's point of view, the kernel does not distinguish between the non-root operation kernel mode (ring 0) and the non-root operation user mode (ring 3), treating the whole virtual machine as a single user-mode program. Thus, we need to program the performance counters to count both kernel-mode and user-mode branches when we switch from the root operation (seL4 kernel, virtual machine monitor, and device drivers run in this mode) to the non-root operation in which the guest VM runs, and revert to count user-mode branches when switching from the non-root operation to the root operation. Switching counting mode can be achieved by programming the following entry and exit control fields of the virtual machine control structure (VMCS): VM-entry MSR-load count, VM-entry MSR-load address, VM-exit MSR-load count, and VM-exit MSR-load address. The VM-entry MSR-load count has the number of MSRs (model-specific registers) we want to load when the VM is about to be executed, and the VM-entry MSR-load address contains the details about the MSRs and the values that should be loaded into the hardware MSRs. The VM-exit fields have similar functionality, and they load the MSRs with the specified values when VM exits. Based on experiment results, for the Core i7 6700 processor, we modify the `IA32_PERFEVTSEL1` and `IA32_PERFEVTSEL2` MSRs that control the counting modes for the counters `IA32_PMC1` (`BR_INST_RETIRED.ALL_BRANCHES`) and `IA32_PMC2` (`BR_INST_RETIRED.FAR_BRANCH`). For VM entries, `IA32_PMC1` and `IA32_PMC2` are programmed to count branches in both kernel mode and user mode without changing their existing values. In the course of VM exits, `IA32_PMC1` and `IA32_PMC2` are reverted to count user-mode branches only. Note that we still maintain the per-replica event counters which also count deterministic VM exits so that we can find out the leading replica quickly when these event counters are not equal. Using this method with slight tunes for different micro-architectures, we are able to launch replicated Linux VMs and run various tests on a machine with an Intel Core i7 4770 CPU, a machine with an Intel Core i5 4590 CPU, and a machine with an Intel Core i7 6700 CPU.

7.3 Microbenchmarks

The microbenchmarks consist of native-seL4 benchmarks and standard benchmarks running inside a Linux VM. For the native-seL4 benchmarks, we report the results of the DMR and TMR systems managed by LC-RCoE and CC-RCoE. By executing similar microbenchmarks under different redundant co-execution modes, we can clearly observe the performance characteristics of LC-RCoE and CC-RCoE. The Linux benchmarks were measured on a baseline single-core Linux VM and replicated Linux VMs managed by hardware-assisted CC-RCoE.

7.3.1 Native Microbenchmarks on seL4

CPU-bound microbenchmark

To evaluate RCoE's effect on CPU-bound applications, we port Dhrystone [Weicker, 1988] and Whetstone [Painter, 1998] to run on seL4, for both x86-64 and ARM platforms. Dhrystone and Whetstone are both synthetic benchmarks. Dhrystone emphasises on integer operations with the following approximate distributions of operators: arithmetic operators (50.8%), comparison operators (42.8%), and logic operators (6.3%). Whetstone is designed to exercise float-point units, representing numeric applications. In addition to floating-point arithmetic operations, a significant portion of execution time is attributed to mathematical library functions (`sin`, `cos`, `atan`, `log`, `exp`, and `sqrt`). A comparison and characteristic analysis of the two benchmarks can be found in [Weicker, 1991]. We use the raw CPU cycles to measure the execution time by reading CPU cycles counters. We set the `Number_Of_Runs` variable to 1,000,000,000 (x86-64) and 200,000,000 (ARM) for Dhrystone, and the `loopstart` variable to 2,000,000 (x86-64) and 500,000 (ARM) for Whetstone; the variables control iterations and thus the execution time. The GCC compiler optimisation level is set to `-O0`. LC-DMR and LC-TMR represent DMR and TMR configurations running in LC-RCoE mode. For ARM, CC-DMR and CC-TMR are managed by compiler-assisted CC-RCoE; for x86-64, they are managed by hardware-assisted CC-RCoE. The averaged CPU cycles of 10 runs and standard deviations are shown in Figure 7.3 and Figure 7.4 for x86-64, and in Figure 7.5 and Figure 7.6 for ARM.

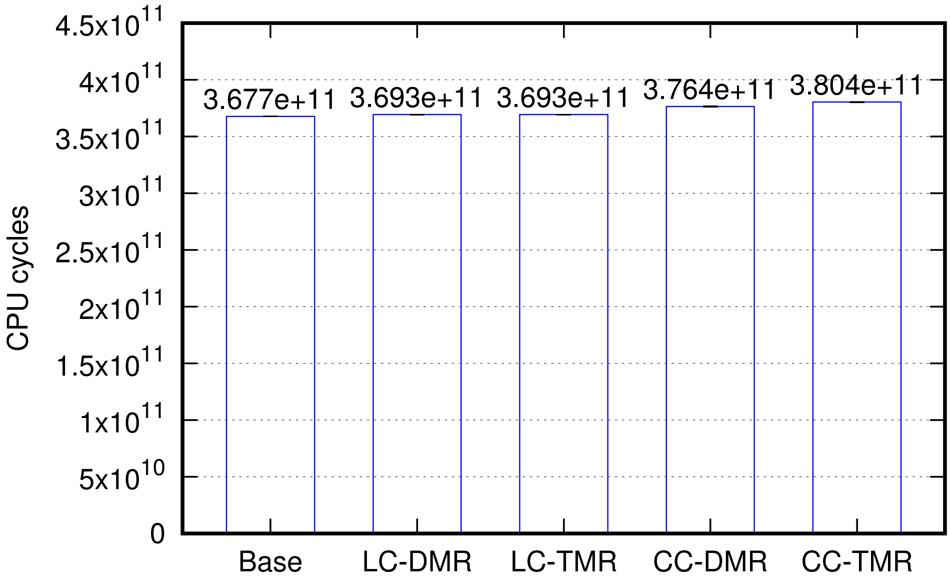

Figure 7.3: Dhrystone on x86-64.

We examine the results of the LC-RCoE variants first. For both Dhrystone and Whetstone, the LC-DMR and LC-TMR variants on ARM and x86-64 exhibit negligible overhead. These results represent the best-case scenario for the following reasons: (1) The benchmarks are CPU-bound and the computations are being executed on each core locally. (2) The benchmarks do not invoke

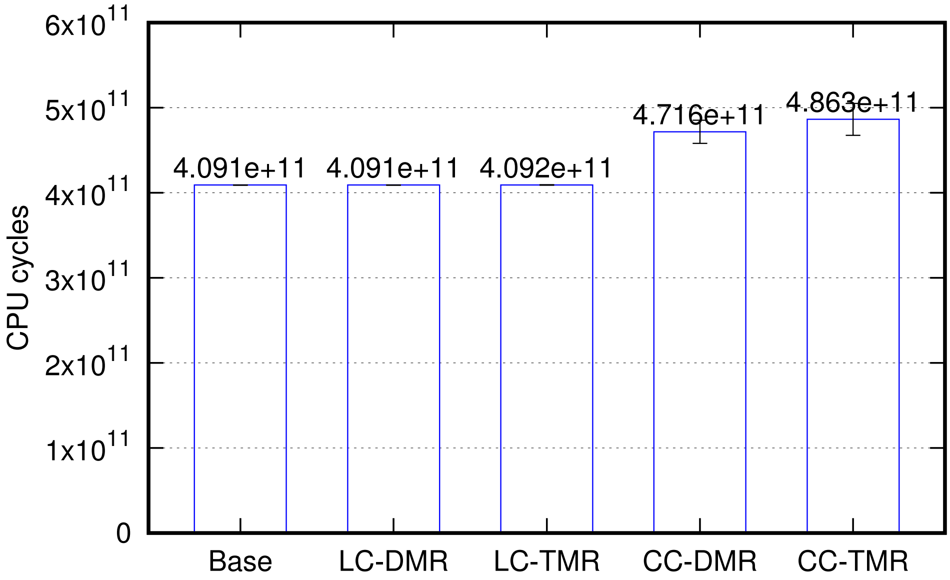

Figure 7.4: Whetstone on x86-64.

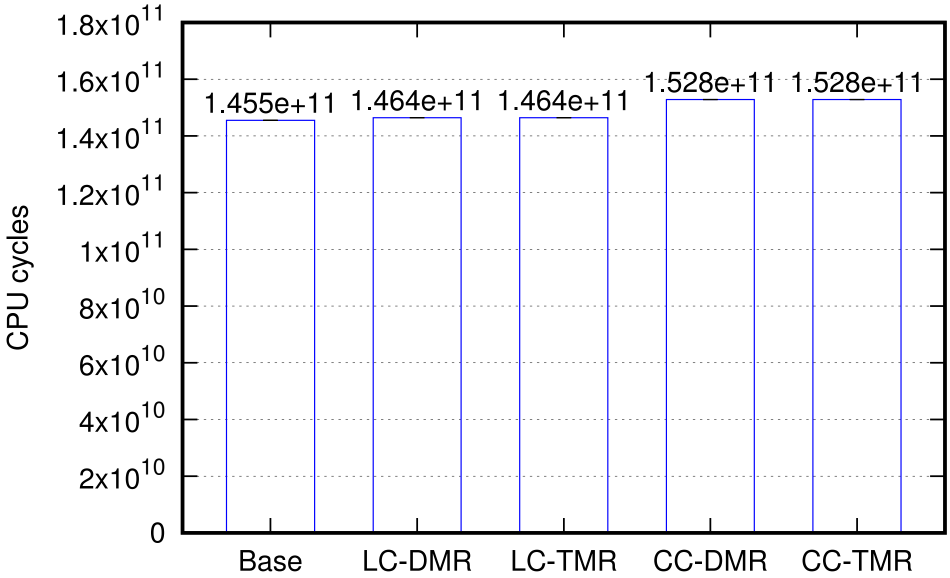

Figure 7.5: Dhrystone on ARM

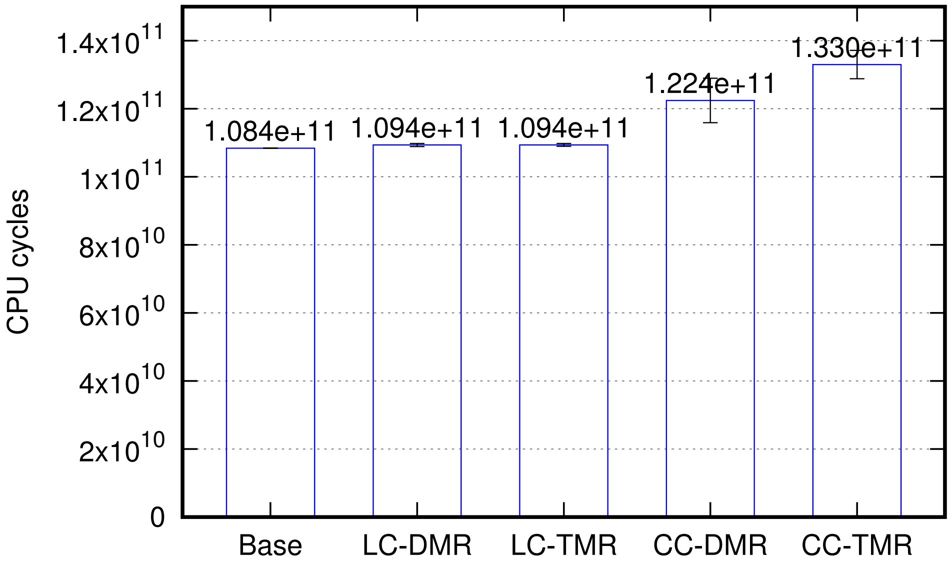

Figure 7.6: Whetstone on ARM

system calls. (3) The active datasets are small so that they can fit in the caches, avoiding contentions on the memory bus. We only expect to observe the synchronisation overhead introduced by the kernel preemption timer interrupts for LC-DMR and LC-TMR configurations. Indeed, our LC-RCoE framework has nearly no effect on such CPU-bound applications when the dataset fits in the caches.

For the CC-RCoE variants, as we expected, the CC-RCoE framework introduces more overhead: (1) Programming debug registers and handling debug exceptions for achieving precise preemption incur overhead. (2) If a breakpoint is set at an instruction that is inside a tight loop, handling repeated debug exceptions induces significant overhead. The CC-DMR and CC-TMR variants' standard deviations for Whetstone on both ARM and x86-64 are significantly larger than the standard deviations of the baseline and LC-RCoE variants. We recognise that the execution time of the CC-RCoE variants is highly dependant on the places where the instruction breakpoints are set when synchronising the replicas. When a breakpoint is set at an instruction inside a tight loop, the execution time of a catching up replica increases because of the increased overhead of handling debug exceptions triggered by the breakpoint. A reader may wonder why the CC-DMR and CC-TMR variants' standard deviations for Dhrystone are insignificant. This can be explained by examining the overall structures of Dhrystone and Whetstone, as shown in Listing 7.1 and Listing 7.2. The main body of Dhrystone is a long loop, which includes invocations to various functions. However, Whetstone is made of several tight loops in which the loop control variables `N1` to `N11` are derived from the user-supplied parameter (`loopstart`). As we can see, Whetstone has a higher probability of setting an instruction breakpoint inside a tight loop than Dhrystone; therefore, we observe higher variances.

```
start = get_time();
for (Run_Index = 1; Run_Index <= Num_Of_Runs; ++Run_Index) {
  Proc_5();
  Proc_4();
  /* other tests */
  ...
  Proc_2();
}
end = get_time();
```

Listing 7.1: Dhrystone structure

```
start = get_time();
/* Module 1 */
for (I = 1; I < N1; I++) {
  /* various operations */
}
/* other modules */
...
/* Module 11 */
for (I = 1; I <= N11; I++) {
  /* various operations */
}
end = get_time();
```

Listing 7.2: Whetstone structure

Memory bandwidth micro-benchmark

To quantify the effect of the redundant co-execution on a memory-intensive benchmark, we used the following simple copy-based benchmark. The benchmark uses two memory regions (a source and a destination), and each region is four times the size of the last-level cache on the platform under test (each region is 32 MiB on x86-64 and 4 MiB on ARM). The regions are pre-mapped to avoid page faults, and they are 4 KiB aligned. The benchmark uses `memcpy()` to copy the source buffer to the destination buffer 100 times. We use a barrier to coordinate the start and finish of the replicas, together with the platform's time-stamp counter to record start and finish times for each run. We run the benchmark 10 times in a loop, and report average copy bandwidth achieved in mebibyte/s together with standard deviations in Figure 7.7 and Figure 7.8. The bandwidths are calculated by the following equation: $block\_size * copy\_times * 2/execution\_time$. We double the total copy size since each copy operation includes one read and one write.

On both platforms, we observe that the replicated configurations, LC-DMR, LC-TMR, CC-DMR, and CC-TMR, split the total available memory copy bandwidth between the replicas; the CC-DMR and CC-TMR variants, as expected, show slightly lower bandwidths than the LC-DMR and LC-TMR variants. In Figure 7.7, the available bandwidths for each replica of the DMR and TMR configurations are almost 1/2 and 1/3 of the baseline bandwidth. Compared to x86-64, the ARM platform has a lower penalty when moving from a non-replicated to a replicated scenario. This is due to a single core on ARM being insufficient to saturate the available memory copy bandwidth of approximately 2.21 GiB/s. For x86-64, a single core is much closer to saturating the available memory copy bandwidth of approximately 25 GiB/s. The theoretical maximum memory bandwidths are 31.78 GiB/s (2133 mega transactions per second * 64 bits per transaction * 2 channels) for the x86-64 machine and 7.7 GiB/s (1033 mega transactions per second * 64 bits per transaction) for the Sabre board.

The main bodies of the `memcpy` functions are shown in Listing E.1 and Listing E.2; note that we omit various checks for parameters and data alignments. The code for the i.MX6 SoC heavily

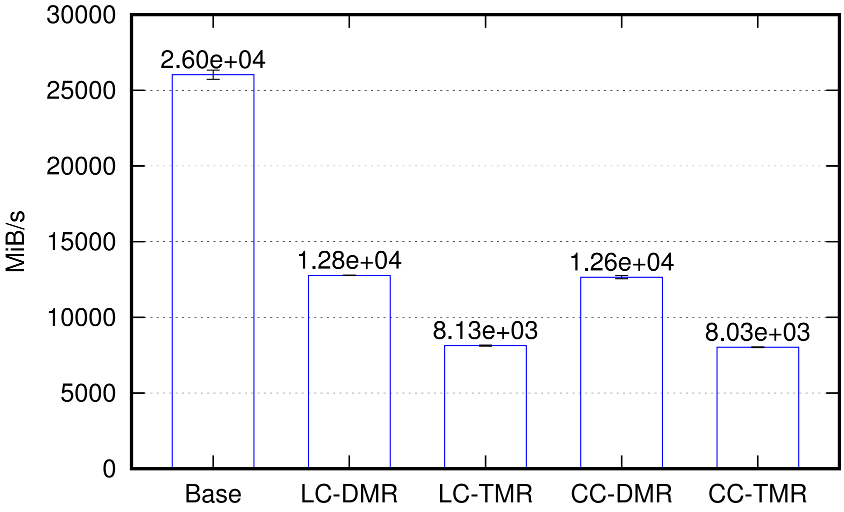

Figure 7.7: A comparison of memory-bound (`memcopy`) micro-benchmark on x86-64.

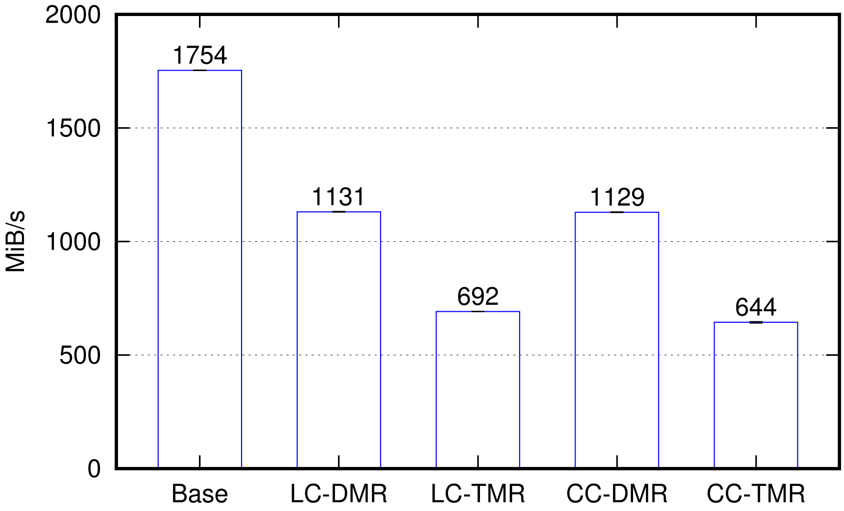

Figure 7.8: A comparison of memory-bound (`memcopy`) micro-benchmark on ARM.

uses the pld instruction to overlap the prefetching of memory into the L1 data cache and the read/write operations. We determine the constant offsets used by the pld instructions based on experimental results on this ARM platform for optimal memory copy bandwidth, and the offsets may or may not be ideal for other platforms. For Cortex-A9 processors, the pld instruction is handled by dedicated hardware units to avoid using integer or load-store units [ARM, 2010]. The two pairs of ldmia and stmia copy 64 bytes of data each time, using the general purpose registers r3 to r10 (LC-RCoE) or r1 to r8 (CC-RCoE) as the buffers. For the x86-64 machine, the memcpy implementation uses SSE (streaming SIMD extensions) registers as the buffers and copies 256 bytes of data each time. It is necessary to use the movntdq (store double quadword using non-temporal hint) for the optimal results since the movntdq instruction is implemented by using a write-combining memory-type protocol which avoids writing data into the cache hierarchy and fetching the cache lines corresponding to the write addresses into the cache hierarchy.

Interrupt latency

Interrupt delivery in our approach requires propagation to all replicas and an agreement on when an interrupt becomes visible to the replicas in order to preserve consistency. Interrupt latency is highly dependent on system activity at the time (e.g., interrupt disabling), so our benchmark involves an idle system consisting of only the in-kernel idle thread, and a user-level timer driver which is effectively an interrupt handling thread. Note that the the kernel disables interrupts while running in kernel mode and re-enables interrupts when the kernel idle thread is running.

The timer driver programs the platform-dependent hardware timer to trigger an interrupt and then blocks waiting for its arrival via IPC. To measure the effect on latency, we instrument the in-kernel interrupt handler on the interrupt-handling core to take a time-stamp early when the system traps to kernel mode. We also take a time-stamp after the user-level driver receives the interrupt notification, i.e., after the interrupt is propagated across all cores and the notification is consistently delivered. The difference between the two timestamps is our metric for interrupt latency. We measure the latencies 500 times in a loop for a baseline unprotected seL4, and DMR and TMR variants managed by LC-RCoE. The average of the 500 runs together with standard deviations are shown in Figure 7.9 and Figure 7.10.

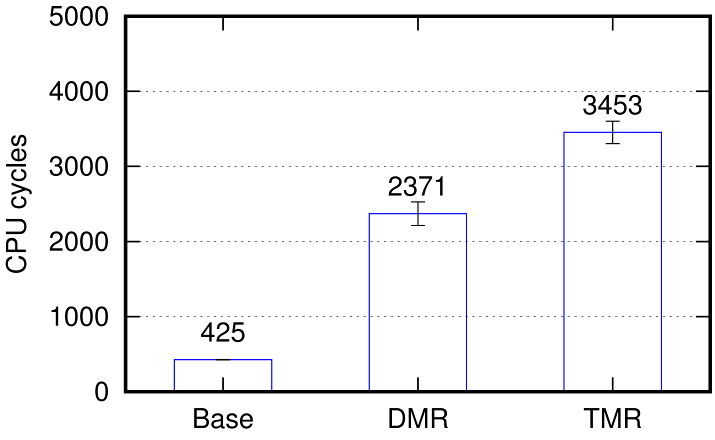

Figure 7.9: A comparison of interrupt latency on x86-64 (LC-RCoE).

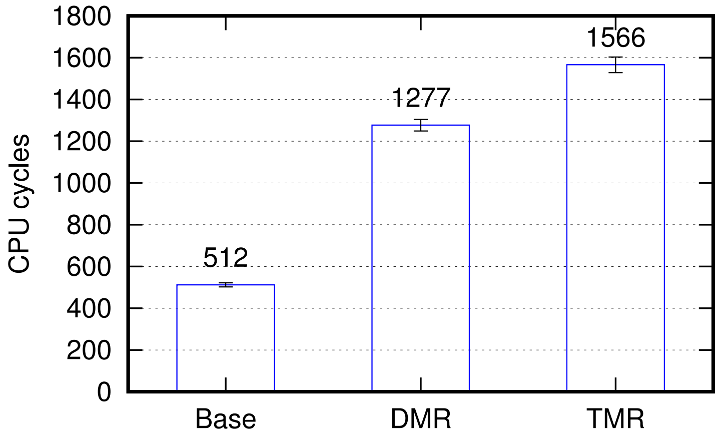

Figure 7.10: A comparison of interrupt latency on ARM (LC-RCoE).

As we can observe from the results, the interrupt latency increases significantly for the DMR and TMR variants. The increase in latency is due to the latency of sending and receiving inter-processor interrupts (IPIs), and the three barriers used to coordinate consistent interrupt observation across replicas. Note that this is the optimal scenario since the replicas already wait for the interrupt, so the kernel synchronisation protocol can skip the leader-waiting-follower-catching-up part.

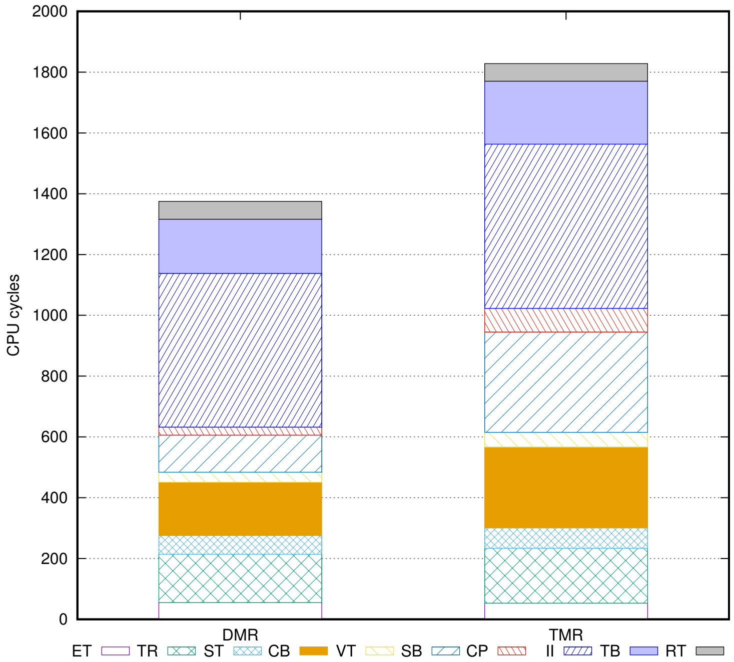

Figure 7.11: The breakdown of interrupt latency on ARM.

Stage	DMR	TMR
ET	55	53
TR	159	181
ST	62	67
CB	175	266
VT	33	48
SB	122	330
CP	26	78
II	506	540
TB	178	207
RT	59	58
Total	1375	1828

Table 7.3: CPU cycles for each stage.

We further break down the latency for the DMR and TMR configurations on the ARM machine by instrumenting the synchronisation code, as shown in Figure 7.11. The results are averaged numbers of 500 runs, and the meanings of the keys in the figure are explained below. We also list the averaged CPU cycles for each stage in Table 7.3; note that the numbers are all from the primary replica.

ET (entry) indicates the cycles spent on saving user-mode registers, invoking kernel interrupt handler, and reading the active interrupt request number from the interrupt controller.

TR (trigger) represents the cycles that the primary replica spends on triggering a synchronisation. It includes setting the synchronisation flags of the replicas, sending inter-processor interrupts to other replicas, and acknowledging the interrupt.

ST (synchronisation start) is the cycles that the primary replica spends on travelling from the interrupt handler function (after a synchronisation has been triggered) to the place before the conditional barrier.

CB (the conditional kernel barrier) indicates the cycles that the primary replica spends on the first conditional barrier.

VT (vote) marks the cycles that the primary replica uses to vote the leading replica.

SB (the second kernel barrier) is the cycles that the primary replica spends on the second kernel barrier.

CP (comparing fingerprints) represents the cycles for comparing fingerprints.

II (interrupt injection) stands for the cycles of interrupt injection operations.

TB (the third kernel barrier) denotes the cycles that the primary replica spends on the third kernel barrier.

RT (return) is the cycles for returning to user mode.

The total cycles in Table 7.3, 1375 and 1828, are higher than the corresponding numbers, 1277 and 1566, in Figure 7.10. We recognise that the inserted code reading timestamps can disturb the execution of the code being measured. The baseline case roughly consists of the ET, II, and RT stages. For the DMR configuration, the total cycles for these three stages are 620, which is larger than the averaged baseline 512 cycles. In addition to the disturbance mentioned above, the II stage for the DMR also has more code to walk through an array of pending interrupts to be processed. Given the increased latency, we expect that high interrupt frequency workload will suffer from RCoE, but as our Redis system benchmark results in Section 7.4.1 demonstrate that real-world workloads can still achieve acceptable performance when RCoE is employed.

7.3.2 Microbenchmarks on Replicated Linux Virtual Machines

Dhrystone and Whetstone on Virtual Machines

We also use Dhrystone and Whetstone shipped as parts of Buildroot system [Buildroot] on the baseline Linux VM and the DMR Linux VMs managed by hardware-assisted CC-RCoE. For Dhrystone, we set the number of runs to 1,000,000,000, making the total execution time on the baseline Linux VM around 86 seconds. As to Whetstone, the number of loops is set to 2,000,000 so that the total execution time on the baseline Linux VM was approximately 55 seconds. Please note that no direct comparison should be made between the results of the benchmarks running on the VMs and the results of the benchmarks running as native-seL4 applications (Section 7.3.1), since the software configurations, building environments, supporting libraries are quite different. For instance, the native-seL4 benchmarks are 64-bit, statically linked, and compiled with optimisation level `-O0`; but the VM benchmarks are 32-bit, dynamically linked, and compiled with optimisation level `-Os`.

The timekeeping of the Linux kernel relies on periodic interrupts from architecture-specific clock sources. In our VM configuration, the 8254 programmable interval timer (PIT) is used to generate interrupts at a frequency of 250 Hz. As we discussed in Section 5.6, the delivery of device interrupts in the DMR and TMR configurations must be coordinated by the synchronisation protocol. Therefore, the delivery of timekeeping interrupts can be delayed non-deterministically in CC-RCoE, which requires setting instruction breakpoints and handling the exceptions triggered by the breakpoints. For this reason, we use the following command to measure the execution time based another machine running native Linux.

```
time ssh user@the_test_machine 'time dhrystone 1000000000'
```

The outer `time` command reports the total execution time of sshing from the native Linux to the testing VM, executing the command `time dhrystone 1000000000`, and returning to the machine. The reported time is based on the native Linux machine. The inner `time` reports the execution time of the command based on the time of the testing VM, which can be inaccurate. I also measured the execution time of the command, `time ssh user@the_test_machine 'echo'`, to establish a baseline round-trip time when the testing machine was running the baseline Linux VM. The average of 10 runs of the command is 0.14 second, and we consider the small overhead can be ignored.

Configuration	VM Time (s)	SSH Time (s)	
Base Linux VM	86 (0)	86 (0)	
DMR Linux VMs	117 (0)	130 (11)	1.5×

Table 7.4: Results for Dhrystone on Linux VMs

Configuration	VM Time (s)	SSH Time (s)	
Base Linux VM	55 (0)	55 (0)	
DMR Linux VMs	82 (1)	159 (11)	2.9×

Table 7.5: Results for Whetstone on Linux VMs

The results for Dhrystone and Whetstone are presented in Table 7.4 and Table 7.5. The *Base Linux VM* is a single-core VM; and the *DMR Linux VMs* are two Linux VMs (each VM runs on one core) managed by CC-RCoE, interacting with other machines as a single logic VM. *VM Time* represents the averaged execution time reported by the `time` of the VMs, and *SSH Time* shows the mean execution time reported by the `time` of the native Linux machine. Standard deviations are in the parentheses. The timekeeping of the baseline Linux VM is relatively accurate. Nevertheless, the timekeeping of the DMR Linux VMs is heavily affected by CC-RCoE, which delays interrupt delivery non-deterministically. The actual execution time of Dhrystone is approximately 1.5 times of the baseline, and the DMR Linux VMs almost triple the baseline time to finish the Whetstone benchmark. Compared with the results in Figure 7.3 and Figure 7.4, the DMR Linux VMs exhibit more overhead when running the benchmarks. This is not a surprise given that the Linux VMs need to execute various background threads and handle network interrupts (ssh connection), while the native-seL4 benchmarks runs in an environment with less noise. However, we consider the main cause of the slowdown is the cost of increased number VM exits that are triggered by instruction breakpoint exceptions. As discussed in Section 7.3.1, setting an instruction breakpoint inside a tight loop further exacerbates the situation for the DMR Linux VMs; the performance degradation of Whetstone that includes many small loops, once again, confirms the theory.

SPLASH-2

Name	Parameters	Introduction
BARNES	65536 bodies	BARNES application implements the Barnes-Hut method to simulate the interaction of a system of bodies (N-body problem).
CHOLESKY	inputs/tk29.O	Cholesky Factorization
FFT	-m22	Fast Fourier Transform.
FFM	163840 particles	FMM implements a parallel adaptive Fast Multipole Method to simulate the interaction of a system of bodies.
LU-C	-n2048	LU-C factors a dense matrix into the product of a lower triangular and an upper triangular matrix with contiguous block allocation.
LU-NC	-n2048	LU-NC is similar as LU-C, but it uses non-contiguous block allocation.
OCEAN-C	default	OCEAN-C simulates large-scale ocean movements based on eddy and boundary currents with contiguous partition allocation.
OCEAN-NC	default	OCEAN-NC is similar as OCEAN-C, but it uses non-contiguous partition allocation.
RADIOSITY	-batch -largeroom	RADIOSITY computes the equilibrium distribution of light in a scene using the hierarchical diffuse radiosity method.
RADIX	-n 26214400 -r1024 -m52428800	RADIX implements an integer radix sort.
RAYTRACE	inputs/teapot.env	RAYTRACE renders a 3-D scene onto a 2-D image plane using optimised ray tracing .
VOLREND	inputs/head.den	VOLREND renders a 3-D volume onto a 2-D image plane using optimised ray casting.
WATER-NS	default	WATER-NSQUARED solves the molecular dynamic N-body problem.
WATER-S	default	WATER-SPATIAL is similar as WATER-NS, but it uses a different algorithm.

Table 7.6: Configurations for SPLASH-2

We execute SPLASH-2 [Woo et al., 1995] to understand how the redundant co-execution affects scientific applications, and we use the same method to measure the execution time of each benchmark as we did for Dhrystone and Whetstone in Section 7.3.2. Specifically, we use the following command, `time ssh user@the_test_vm 'time run_splash2.sh N test'` (`N` is the times to repeat, and `test` is the benchmark to run), to repeat each test `N` times and ensure that the baseline execution time of each test is more than 40 seconds, which are significantly longer than the added latency for executing the `ssh` command. SPLASH-2 is compiled as statically linked 32-bit x86 ELF binaries with POSIX thread enabled, and the parameters and brief introductions for the benchmarks are listed in Table 7.6. All benchmarks are executed with `NPROC = 1` by specifying command-line options or defining parameters in input files.

Name	N	Base	CC-DMR	Factor
BARNES	30	60 (0)	68 (1)	1.14
CHOLESKY	300	41 (0)	100 (8)	2.43
FFT	100	72 (0)	145 (7)	2.00
FFM	20	74 (0)	136 (9)	1.83
LU-C	30	64 (0)	520 (42)	8.14
LU-NC	20	63 (0)	390 (43)	6.23
OCEAN-C	1000	62 (0)	184 (1)	2.96
OCEAN-NC	1000	64 (0)	186 (1)	2.92
RADIOSITY	25	66 (0)	74 (1)	1.12
RADIX	20	66 (0)	99 (12)	1.50
RAYTRACE	1000	58 (0)	64 (1)	1.10
VOLREND	100	72 (0)	88 (1)	1.22
WATER-NS	600	63 (0)	90 (2)	1.43
WATER-S	600	67 (0)	85 (3)	1.27
Geometric mean				2.02

Table 7.7: SPLASH-2 results on VMs. The total time in seconds represents the length of N repeated runs. Standard deviations (in units of seconds) for the total execution time are shown in the parenthesis.

We present the benchmark results in Table 7.7. Base and CC-DMR represent the total execution time in seconds for the baseline VM and the DMR VMs, measured by the Linux machine issuing the `ssh` command. Since we use the script, `run_splash2.sh`, to execute a kernel or an application `N` times, the total execution time actually includes the overhead of creating and destroying processes as well as interpreting the script. In the last column, the DMR results are normalised to the corresponding baseline results. The applications or kernels that experience significant or severe performance degradation in the DMR mode are highlighted in blue or red colour respectively. Evidently, we observe that the performance of the LU-C and LU-NC is massively affected by redundant co-execution. These two applications perform various operations on matrices, so, not surprisingly, the main bodies are composed of loops accessing or modifying matrix elements. The remaining applications demonstrate varied increases in execution time from 1.10 to 2.96 times, and the geometric mean of the increases in execution time for all applications is 2.02 times. As to some applications, the significantly hiked execution time limits the applicability of CC-RCoE; therefore, we set reducing the performance overhead of CC-RCoE as future work (Section 8.2.5). Nevertheless, in some cases (BARNES, RAYTRACE, RADIOSITY, etc.), we consider the overhead is tolerable.

Memory bandwidth – RAMspeed on VMs

RAMspeed benchmarking program [Hollander and Bolotoff] is used to understand how CC-RCoE affects memory-intensive applications in a VM environment. We run the following selected tests: `INTmem`, `FLOATmem`, and `SSEmem(nt)`, which use integer registers, floating-point registers, and SSE registers as the temporary data storage and corresponding instructions for data movement respectively. The `SSEmem(nt)` test uses non-temporal store instructions (`movntps`), and the non-temporal hint is implemented with a write combining memory protocol to avoid polluting caches when writing to memory. The processors implemented the protocol do not write the data into cache hierarchy, nor do they fetch the corresponding cache line from memory into the cache hierarchy. Each test contains four sub-tests, `Copy`, `Scale`, `Add`, and `Triad`, which simulate real-world applications. `Copy` simply transfers data from one memory location to another ($A = B$). `Scale` modifies the data before writing by multiplying the data with a constant value ($A = m * B$). `Add` reads data from two memory locations and writes the sum of the data to a third memory location ($A = B + C$). `Triad` combines `Add` and `Scale` ($A = m * B + C$). We supply the following parameters in the command below for the tests—8 GiB (-g 8) memory per pass and 32 MiB (-m 32) arrays for source and destination locations; each test is repeated 10 times (-l 10). The following command is used to start the benchmark.

```
time ssh user@machine ‘time ramspeed -b test_id -g 8 -m 32 -l 10‘
```

We report the memory bandwidths measured by the VMs in Figure 7.12, and Table 7.8 shows the execution time measured by the VMs and by the Linux machine initiating the benchmark remotely. `IntBase`, `FloatBase`, and `SSEBase` represent the results of `INTmem`, `FLOATmem`, and `SSEmem(nt)` for the baseline VM; `IntDMR`, `FloatDMR`, and `SSEDMR` are the results of the DMR Linux VMs.

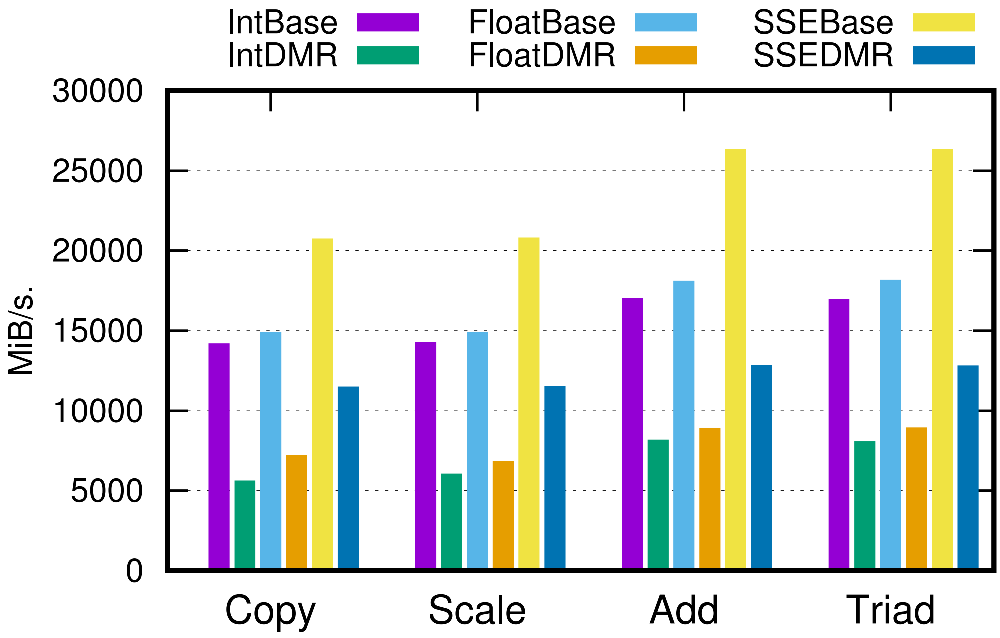

Figure 7.12: RAMspeed memory bandwidth in MiB/s

	IntBase	FloatBase	SSEBase	IntDMR	FloatDMR	SSEDMR
VM Time	55.12	52.06	45.07	123.65	107.81	85.86
SSH Time	55.236	52.207	45.216	126.188	110.242	88.769

Table 7.8: RAMSpeed execution time in seconds

CC-RCoE still affects the time keeping of the DMR VMs when RAMspeed is executing, but the impact is less profound than that in the Dhrystone and Whetstone benchmarks (Section 7.3.2). Thus, the DMR bandwidths in Table 7.8 are slightly higher than the actual bandwidths. The DMR variant coordinated by the CC-RCoE roughly achieves 40% to 55% of the baseline VM's bandwidths, depending on the sub-tests (`Copy`, `Scale`, `Add`, or `Triad`) and registers (integer, floating-point, or SSE). We take a closer look at the results of the `Copy` sub-test using SSE registers and non-temporal store instructions, since the `SSEDMR` bandwidth (11450 MiB/s) is more than half of the `SEEBase` bandwidth (20711 MiB/s). To get the maximal memory copy bandwidth on this machine, we use the improved version of RAMspeed which can spawn multiple processes and report the total bandwidth. We set the process number to 4 and ensure that the processes are running on different cores. The total memory copy bandwidth reported by RAMspeed/SMP on native Linux is approximately 25253 MiB/s with SEE registers and non-temporal store instructions. The bandwidth is very close to the memory copy bandwidth (2.5 GiB/s) of the native seL4 `memcpy` benchmark in Figure 7.7. Thus, RAMspeed running on the baseline VM is unable to saturate the memory bandwidth; this explains why the DMR VMs achieve around 55% of the baseline memory copy bandwidth with SSE registers and non-temporal store instructions.

Another observation is that `IntDMR` `Copy` only achieves 40% of the baseline. We inspect the source code of RAMspeed and find out that the copy function using integer registers copies 128 bytes for each iteration; but the copy functions using floating-point registers and SSE registers copy 256 bytes and 1024 bytes respectively for each iteration. The copy function using integer registers has more loops in order to copy the same amount of data, and more loops lead to higher overhead for the CC-RCoE managed VMs.

7.4 System Benchmarks

7.4.1 Redis on Native seL4

To test our system under a more realistic workload, we choose Redis [RedisLabs, 2009], a key-value store set up as shown in Figure 7.13, for the following reasons: (1) Redis is implemented in ANSI C without external dependencies, and our colleagues already ported it to run on native seL4. (2) Redis adopts a single-threaded, event-driven design and thus saves us from analysing source code for data races. (3) Redis stores data in memory and supports various data structures as well as related operations on the data, exercising both the CPU and memory. (4) The load generator for Redis can be executed on another dedicated machine connected by 1 Gbps network so that the network stack and network card driver are also being stressed during the benchmark, simulating a close-to-real-world scenario including I/O devices. (5) Not only does the mixed workload (CPU, memory, and I/O) reveal the overhead of the synchronisation protocol but also the cost of supporting driver replication.

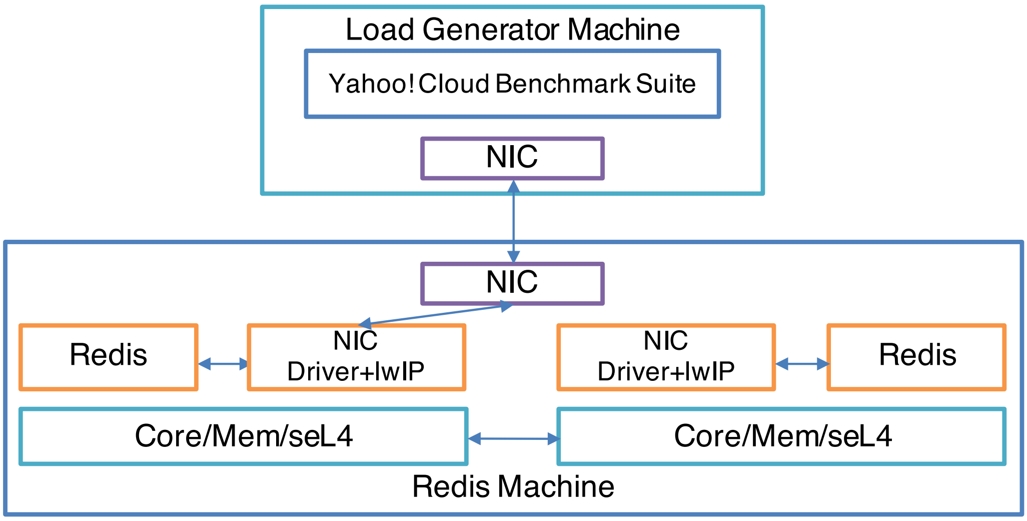

Figure 7.13: Redis-based benchmark architecture (DMR).

The target system runs seL4 with one process dedicated to lwIP [Dunkels, 2001] combined with the Ethernet device driver, which handles I/O interrupts and network packages; and another process runs an instance of Redis. Note that we run Redis as volatile store with file system access disabled, as our prototype lacks a port of a file system. The device driver was modified so as to support replication in LC-DMR-SC, LC-DMR-SCN, LC-DMR-SCI, LC-TMR-SC, LC-TMR-SCN, LC-TMR-SCI, CC-DMR-SC, CC-DMR-SCN, CC-DMR-SCI, CC-TMR-SC, CC-TMR-SCN, and CC-TMR-SCI configurations as described below.

- *Base*: The unprotected single-core baseline.
- *LC-DMR-SC*: The whole system is replicated onto two cores and executed in LC-RCoE. System call parameters are collected into fingerprints, and the fingerprints are compared only before handing I/O device interrupts. The network drivers contribute the checksums of the output data into the fingerprints; the kernel barrier timeout is also enabled. This configuration corresponds to the BT + CU + AC + SC configuration described in Section 6.4.
- *LC-DMR-SCN*: This configuration is similar to LC-DMR-SC except that we only collect system call numbers into the fingerprints and omit the system call parameters; the fingerprints are compared only before handling I/O device interrupts. The network drivers contribute the checksums of output data into the fingerprints; the kernel barrier timeout is also enabled. This configuration corresponds to the BT + CU + AC + SCN configuration described in Section 6.4.
- *LC-DMR-SCI*: The whole system is replicated onto two cores and executed in LC-RCoE. System call parameters and IPC buffers are collected into fingerprints; and for each system call, the replicas synchronise and compare the fingerprints, in addition to the comparisons before handling I/O device interrupts. The network drivers contribute the checksums of the output data into the fingerprints; the kernel barrier timeout is also enabled. This corresponds to the BT + CU + AC + SCI configuration described in the Section 6.4.

- *LC-TMR-SC*: The whole system is replicated onto three cores and executed in LC-RCoE; the error-detection configuration is BT + CU + AC + SC.
- *LC-TMR-SCN*: The whole system is replicated onto three cores and executed in LC-RCoE; the error-detection configuration is BT + CU + AC + SCN.
- *LC-TMR-SCI*: The whole system is replicated onto three cores and executed in LC-RCoE; the error-detection configuration is BT + CU + AC + SCI.
- *CC-DMR-SC*: The whole system is replicated onto two cores and executed in CC-RCoE. For the ARMv7 SoC, compiler-assisted CC-RCoE (Section 4.4.3) is adopted, and hardware-assisted CC-RCoE (Section 4.4.2) is used for the x86-64 machine. The error-detection configuration is BT + CU + AC + SC.
- *CC-DMR-SCN*: The whole system is replicated onto two cores and executed in CC-RCoE; the error-detection configuration is BT + CU + AC + SCN.
- *CC-DMR-SCI*: The whole system is replicated onto two cores and executed in CC-RCoE; the error-detection configuration is BT + CU + AC + SCI.
- *CC-TMR-SC*: The whole system is replicated on three cores and executed in CC-RCoE; the error-detection configuration is BT + CU + AC + SC.
- *CC-TMR-SCN*: The whole system is replicated on three cores and executed in CC-RCoE; the error-detection configuration is BT + CU+ AC + SCN.
- *CC-TMR-SCI*: The whole system is replicated on three cores and executed in CC-RCoE; the error-detection configuration is BT + CU + AC + SCI.

	For ARM	For x86
CPU	Core i5 4250U dual-core@1.30 GHz (2.3 GHz dual-core TurboBoost)	Xeon E5-2683 v3 14-core@2.00 GHz×2 sockets
Cache	2×32 KiB L1 D/I, 2×256 KiB L2, 3 MiB L3	14×32 KiB L1 D/I, 14×256 KiB L2, 35 MiB L3
Memory	8 GiB LPDDR3-1600	16×16 GiB DDR4-2133
NIC	Thunderbolt to Gigabit Ethernet Adapter	Broadcom NetXtreme BCM5720

Table 7.9: Load generator machines for Redis

We evaluate performance of the baseline and LC-RCoE Redis servers using Yahoo! Cloud Serving Benchmarks (YCSB) [Cooper et al., 2010], running on dedicated load generator machines as in Table 7.9 for x86 and ARM target machines, with dedicated Gigabit Ethernet links between the load generators and the machines under test. The hardware configurations are chosen to ensure that the load generators are more powerful than the machines running the Redis server. During the benchmarks we monitor the CPU-load and network bandwidth to ensure the benchmark performance is not limited by the load generators. YSCB consists of several workloads. We use the same A–F benchmarks as presented by the benchmark developers, which are as follows.

A: update-heavy workload (50/50 reads and writes) using zipfian distribution for record selection in the store.

B: read-mostly workload (95/5) with zipfian distribution.

C: read-only workload with zipfian distribution.

D: new records inserted, then most recently inserted record are read.

E: short ranges of records are queried, where record selection is zipfian, but the number of records in the range is uniformly distributed.

F: read-modify-write workload (50% read and 50% read-modify-write) using zipfian distribution.

We set `recordcount` to 70,000 on ARM and x86-64 for all workloads; `operationcount` is 10 times `recordcount`, except for workload E which is 1. The goal of tuning the parameters was to give a database size (around 160 MiB on ARM and 190 MiB on x86-64 as reported by the `info memory` Redis client command) significantly larger than the last-level cache sizes. For each platform, we run the YCSB benchmark set 10 times for an unprotected single-core baseline and protected DMR and TMR variants of the system. The averaged throughput results are reported in Figure 7.14 and Figure 7.15; the standard deviations are shown as error bars in the figures, and they are less than 2.3% (x86-64) and 2.6% (ARM). Note: we multiply the throughputs of workload 'E' by 50 to make it comparable on the scale.

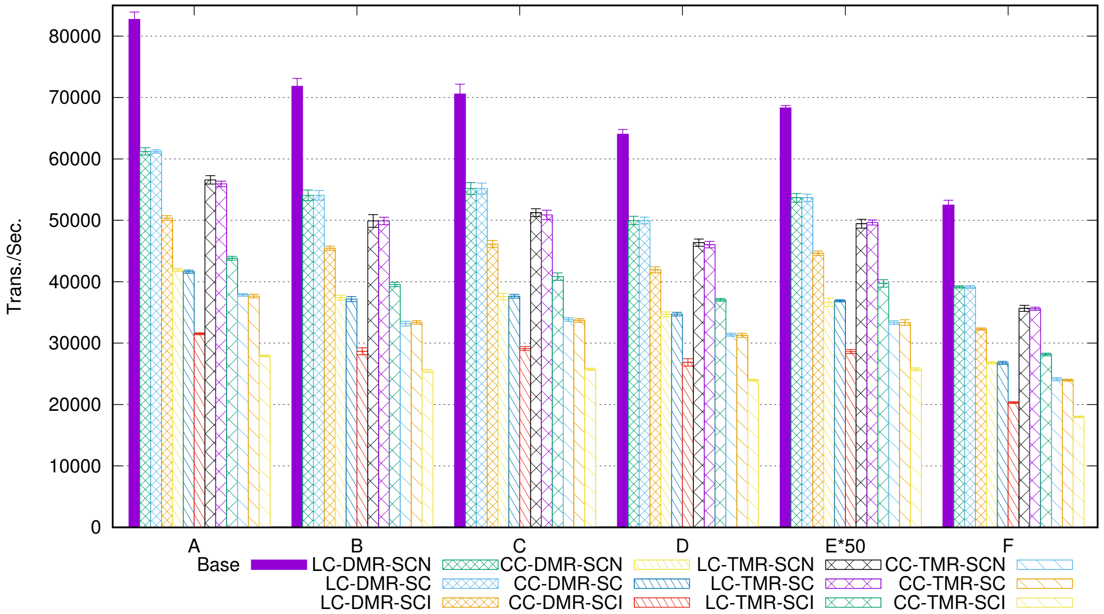

Figure 7.14: Average Redis transactions per second on x86-64 for each configuration and workload. 'E' multiplied by 50.

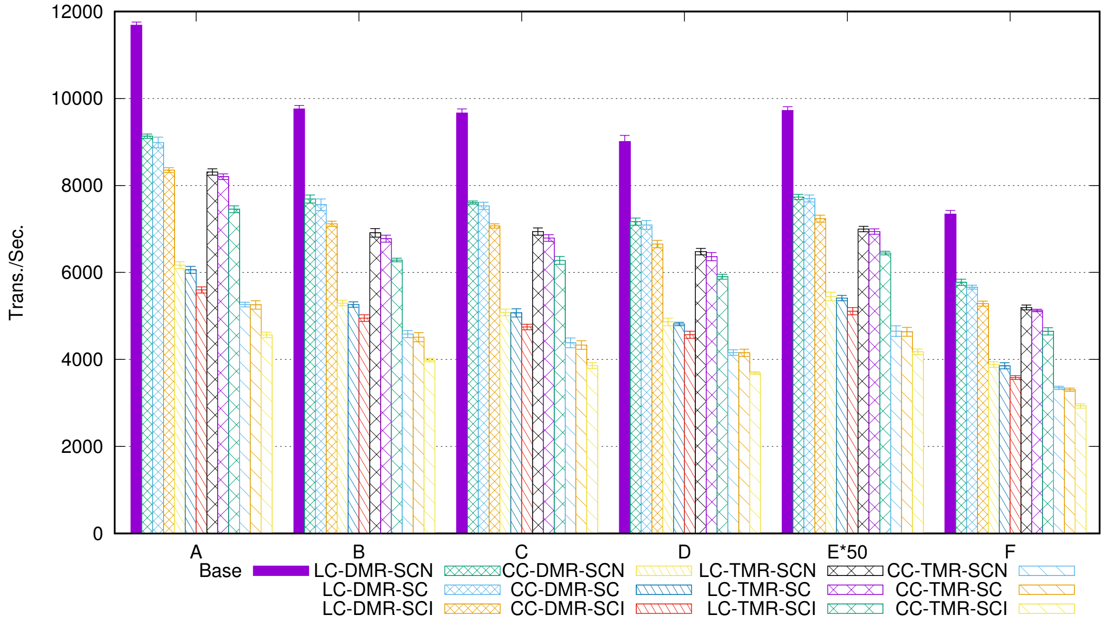

Figure 7.15: Average Redis transactions per second on ARM for each configuration and workload. 'E' multiplied by 50.

	LC-RCoE		CC-RCoE	
Mode	ARM	x86	ARM	x86
DMR-SCN	78–80%	74–79%	53–56%	51–54%
DMR-SC	77–79%	74–79%	52–56%	50–54%
DMR-SCI	72–75%	61–66%	48–53%	38–42%
TMR-SCN	71–72%	68–73%	45–48%	46–49%
TMR-SC	69–71%	68–73%	45–48%	46–49%
TMR-SCI	63–66%	53–58%	39–43%	34–38%

Table 7.10: Redis throughputs normalised to baseline

The throughputs normalised to the baseline for various configurations are listed in Table 7.10. Our recommended LC-DMR-SC and LC-TMR-SC perform reasonably well: The performance of the LC-DMR-SC systems is between 77%–79% (ARM) and 74%–79% (x86-64) of the baseline, and LC-TMR-SC systems achieve 69%–71% (ARM) and 68%–73% (x86-64) of the baseline. The CC-RCoE variants, as expected, show significant performance overhead: The compiler-assisted CC-DMR-SC and CC-TMR-SC on ARM achieve only 52%–56% and 45%–48% of the baseline, and the hardware-assisted CC-DMR-SC and CC-TMR-SC on x86-64 deliver 50%–54% and 46%–49% of the baseline. Compared with the CPU-bound results (which show insignificant slowdown) in Section 7.3.1, we can see that the performance overheads when executing the Redis system benchmark are quite significant. The main reason for that is the Redis benchmark exercises the cores, memory, and I/O devices altogether instead of only the cores performing limited accesses to memory and I/O devices. Furthermore, the high-frequency network and timer interrupts of the Redis benchmark trigger more synchronisations; more synchronisations imply higher overheads.

The performance degradation of the CC-RCoE variants deserves some discussions. The overheads are mainly from two factors: (1) The additional work introduced by the CC-RCoE mode is discussed in Section 7.3.1. (2) The cost, introduced by supporting driver replication as described in Section 5.5, increases linearly to the frequency of I/O operations, such as accessing memory-mapped registers or DMA buffers. The cost of accessing MMIO registers also exists in LC-RCoE, but it is less than that of CC-RCoE since LC-RCoE implements input data replication in user mode rather than invoking the `seL4_FT_Mem_Access` system call, which requires two mode switches. Similarly, reading data from DMA buffers in CC-RCoE imposes much more overhead than that of LC-RCoE since the `seL4_FT_Mem_Rep` system call has to be called for each DMA read access to copy the data in the buffers. Each `seL4_FT_Mem_Rep` involves two mode switches, two or more calls to the kernel-mode barriers for coordinating the data copy, and loops for actually copying the data to the corresponding DMA buffers of the non-primary replicas. Contrastingly, reading DMA buffers in LC-RCoE only needs a user-mode barrier. Due to the performance degradation introduced by CC-RCoE, we recommend LC-RCoE whenever it is applicable and reserve CC-RCoE for executing binary-only applications or virtual machines, which cannot be easily supported by LC-RCoE.

We also observe that the SCN variants, which do not include system call parameters into the fingerprints, have an insignificant performance advantage over the SC variants. Therefore, we suggest enabling the SC option for more complete coverage. The SCI variants impose a modest performance penalty on ARM, but the x86-64 SCI versions exhibit more overhead. These variants trade performance for more frequent fingerprint comparison and thus the lower the error-detection latency. Overall, we consider our approach is applicable even for systems that are I/O-intensive and high interrupt frequency, if high performance is not the first priority. Furthermore, we can turn various parameters to trade performance for improved error coverage and error-detection latency or vice versa.

7.5 A Comparison of the CC-RCoE and LC-RCoE

It is useful to analyse the advantages and disadvantages of CC-RCoE and LC-RCoE so that we can understand which approach fits better for a given software system and hardware platform. Table 7.11 compares the important attributes of the three approaches to implementing redundant co-execution. HA stands for hardware-assisted, and CA means compiler-assisted. Note that we exclude device drivers in the discussion since the drivers need modifications to support the RCoE (i.e., the source code of the drivers must be available), and Chapter 5 is dedicated to discussing the microkernel support for device drivers.

Clearly, if the software is a collection of data-race-free applications, be they single-threaded or multi-threaded, the ideal choice is LC-RCoE with the improvement of supporting multi-threaded applications. The selection of hardware platform is flexible since LC-RCoE does not require hardware features, and thus LC-RCoE is able to run on any architectures supported by seL4.

For x86 binary-only applications or software with the huge code base, which are unsuitable for analysing data races, hardware-assisted CC-RCoE may be a viable approach. However, the reliance on accurate hardware performance counters restricts the choices of hardware—currently,

	LC-RCoE	HA CC-RCoE	CA CC-RCoE
Applicability	Data-race-free apps.	x86 binary apps. or VMs	Source code
Hardware	Nil	Accurate branch counters, debugging	Debugging
Overhead	Low	High	High
Complexity	Moderate	High	Very high

Table 7.11: A comparison of LC-RCoE, HA CC-RCoE, and CA CC-RCoE

only x86 processors with accurate branch counting are supported by hardware-assisted CC-RCoE. Compared with LC-RCoE, the additional performance overheads of hardware-assisted CC-RCoE are mainly from three factors: reading performance counter, programming debug registers, and handling debug exceptions. What makes performance analysis harder is that an instruction breakpoint can be set at an instruction inside a loop in which the instruction may be repeated hundreds or thousands of times. Subsequently, hundreds or thousands of debug exceptions must be handled by the catching up replicas before synchronisation finishes; these additional debug exceptions can significantly increase the required time for synchronising the replicas. We attribute the unpredictable runtime overhead an important reason why LC-RCoE should be preferred to CC-RCoE if possible, and it is on our road map to reduce the overhead of hardware-assisted CC-RCoE (Section 8.2.5).

Compiler-assisted CC-RCoE is the most complex approach in terms of applicability and usability: The need to recompile everything (i.e., applications, libraries, and even the functions in libgcc; see Section D.5) makes the approach unfriendly to end users. CA CC-RCoE also suffers from the issue of unpredictable runtime overhead when the instruction breakpoints are set inside loops. Furthermore, on ARMv7 processors, each catching-up kernel replica needs to handle two debug exceptions for each activated instruction breakpoint until synchronisation finishes (also consult Section D.5 for details). For the reasons above, we expect the chance of applying this approach to a software system is limited; but we still developed the approach as a mitigation plan for the cases that cannot be handled by LC-RCoE and HA CC-RCoE.

7.6 Fault Injection Experiments

Since SEUs are rare and unpredictable, significant time and hardware resources would be required to collect statistical meaningful results if we only rely on capturing naturally occurred SEUs. Fault injection tools are useful and time-saving in evaluating system dependability, and the tools can be implemented in hardware or software [Hsueh et al., 1997]. For our purpose, software-implemented fault injection (SWIFI) tools are more suitable and accessible since we can have control over how and where faults will be injected. In this section, we first survey existing tools for conducting fault injection experiments and point out why the existing tools are not suitable for our experiments. Then, we introduce our fault-injection framework tailored for the seL4 kernel and redundant co-execution, explaining how various faults are injected. Lastly, we present the results of the fault injection experiments and evaluate the effectiveness of our error detection mechanisms.

Ideally, we should compare our software system protected by RCoE on COTS hardware with the software system running on hardware with ECC memory and radiation-hardened processors, to understand how our software-implemented fault-tolerant mechanisms perform compared with expensive hardware components. Conducting such comparisons requires a high radiation environment (e.g., neutron beams) in which hardware components can have higher SEU rates, and thus we can collect results during a reasonable time frame. We did not have the opportunity of gaining access to such an environment, despite that several attempts were made.

7.6.1 Existing Fault Injection Tools

Various fault injection tools based on CPU debugging or monitoring features, virtualisation technology, or simulators have been developed over the years. Xception [Carreira et al., 1998] uses debugging and performance monitoring units of PowerPC MPC601 processors to trigger hardware exceptions when the conditions for injecting faults are met, and the exception handler, which is integrated with the PARIX kernel, modifies the targeted components, such as memory locations, general-purpose registers, floating-point registers, etc. This fault injection tool is tightly integrated with the PARIX kernel so that it is unsuitable for our purpose. FAUmachine [Potyra et al., 2007] is an open-source emulator for PC hardware and capable of running a complete operating system. It also incorporates fault injection functions featuring the following fault models: transient memory bit flips, permanent memory stuck-at errors, transient or permanent disk block faults, whole disk faults, or transient/intermittent/permanent network sending/receiving faults. However, injecting faults to CPU registers is not supported; neither is multicore, which is required by our approach, supported by the FAUmachine. QEMU-based fault injection methods are explored in [Ferraretto and Pravadelli, 2015; Geissler et al., 2014], but simulating multiple cores with multiple host threads is still in active research [Ding et al., 2011; Wang et al., 2011] and development. FAIL* [Schirmeier et al., 2015] is capable of conducting fault injection campaigns for ARM and x86 architectures; it supports three simulators (Bochs, QEMU, and Gem5) and Cortex-A9 boards that can be controlled with JTAG (e.g., PandaBoard). Bochs [Bochs, 2017] multiplexes a single host thread to execute instruction streams from multiple emulated virtual CPUs if the SMP option is enabled, serialising the instructions issued by different virtual CPUs. Neither QEMU nor Bochs is suitable for conducting the fault injections since their modelling and implementations of multiple virtual CPUs are unable to faithfully reflect the runtime behaviour of real multicore processors.

7.6.2 The Fault Models and Fault Injection Framework

We decide to build our own fault injection tools running on real hardware for the following reasons: (1) Our kernel barrier timeout mechanism requires the measurement of elapsed time, but a simulator-based system is unable to fulfil the requirement. (2) CC-RCoE for the x86 architecture mandates accurate values for the two performance monitoring events. (3) Hardware multicore processors implement the hardware parallelism exploited by our our redundant co-execution (RCoE) approaches. (4) Physical network cards with DMA capability are required to assess to what extend the non-replicated parts of the system affect the effectiveness of the error detection mechanisms.

Since our aim is to conduct fault injections on real hardware and the seL4 kernel is the lowest-level system software, our strategy is to modify the kernel to inject random or targeted faults while a system is running, without relying on other software components (e.g., a hypervisor inserted under the seL4 kernel). We describe fault models used in the experiments, and then introduce how the models are implemented without diving into technical details.

Fault models

We consider the following fault models: (1) single-bit flips in user-mode general-purpose registers, (2) corrupted system call parameters, (3) bit flips in kernel memory, and (4) random bit flips in memory. The model (1) focuses on application-level faults, and it is adopted in [Kuvaiskii et al., 2016; Reis et al., 2005; Shye et al., 2009] to evaluate error detection techniques targeting applications. The model (2) simulates the errors propagated from user mode to kernel mode; the model (3) evaluates how the synchronisation protocol and error detection mechanisms behave when faults are injected into the kernel. The models (2) and/or (3) are used to assess the reliability of COTS microkernels [Arlat et al., 2002], to characterise Linux kernel's behaviour under different fault types [Jarboui et al., 2002], and to compare the error behaviour of different operating systems running on various CPU architectures [Chen et al., 2008]. The model (4) simulates extreme cases that multiple errors happen at random memory locations, and it is employed by Messer et al. [2004] to discover the susceptibility of Linux kernel and JVM to memory errors. We use three experiments (Section 7.6.3, Section 7.6.4, and Section 7.6.4) to cover the fault models (1), (3), and (4). The model (2) is omitted since corrupted system call parameters can be easily detected by the fingerprint comparison if the parameters are included in the fingerprints.

The random number generator

The fault injection experiments need to choose an address, a register, or a bit in a data word randomly as a fault injection target. For this purpose, we use a complementary multiply-with-carry (CMWC) [Marsaglia, 2003] pseudo random number generator. The sequences of random number generator seeds for the baseline (unprotected) and the protected DMR and TMR are the same, ensuring relatively fair comparisons.

Injecting bit flips into user-mode general-purpose CPU registers

To corrupt user-mode registers is straightforward since the kernel can access all registers of a thread. The register values of the running thread are saved in the TCB (thread control block, see Section 2.4) when the thread is interrupted by hardware interrupts; the saved values will be popped back to physical registers when the kernel resumes the thread. Thus, we modify the interrupt entry point of the kernel; the inserted code reads the value of the targeting register from the TCB, flips a chosen bit, and writes the corrupted value back to the TCB, if the fault injection condition is met for the current interrupt. Note that we manually specify the injection conditions in C code.

Injecting bit flips into memory

For memory errors, we use a spare CPU core, which is not used by redundant co-execution, to modify the selected memory addresses while other cores are executing normally. During bootstrapping, the seL4 kernel activates an additional core for the purpose of fault injection, starts that core with the fault injection code, and continues normal initialisation. The fault injection core runs in kernel mode and has access to all physical memory regions, so it can modify any physical memory address as required. The injection regions can be specified, so we can conduct targeted or random fault injections. The main advantage of using a spare core is that the cores used for redundant co-execution are not interrupted by fault injection activities.

Overclocking cores for random CPU faults

We increase the CPU frequency of the ARM SoC beyond the maximal operational frequency, aiming to induce random CPU glitches. For this method, we have no control over the number of faults induced, neither can we specify which components (registers, ALUs, internal buses, caches, etc.) should be affected. However, we use this method as an extreme case study to understand how our protected system behave when multiple cores may be affected by induced faults.

7.6.3 Fault Injection Campaign against a CPU-Bound Application

For this fault injection experiment, we evaluate how effective our approach is able to detect data corruptions in CPU registers by using a CPU-bound application, `md5sum`, which is a part of the BusyBox [BusyBox, 2017]. The `md5sum` program implements the MD5 [Rivest, 1992] algorithm and produces a 128-bit hash value for a file, and the value is very sensitive to the input data: a random (i.e., not sophisticatedly designed) single-bit flip is likely to produce significantly divergent hash values.

```
#!/bin/mksh
runs=0
# remove the old file
rm -f tmp
# a new file with random data
head -c 128m < /dev/urandom > tmp
# baseline hash value
md5sum tmp > base
# md5sum in a loop
while true
do
  ((runs=runs+1))
  # generate a hash value again
  md5sum tmp > target
  # compare the hashes
  diff base target
  # different hash values?
  if [[ $? -ne 0 ]]; then
    if [[ -s target ]]; then
      # corrupted values
      echo corruptions
    else
      # md5sum crashes
      echo crash
    fi
    echo $runs
    exit 1
  fi
done
```

Listing 7.3: The script to execute md5sum in a loop

We use the Linux virtual machine as our testing platform and conduct targeted fault injections. Once the virtual machine has booted up, the script in Listing 7.3 is executed to ensure that `md5sum` dominates the CPU usage. We first create a 128 MiB file filled with random data and produce an error-free digest file named `base`. Then, we execute `md5sum` in a loop; in each loop, we compare the newly produced digest file `target` with the digest file `base` to detect potential data corruptions that may happen during the computations. If the `diff` utility returns 0, the files are the same. Otherwise, the `target` file mismatches the `base` file so that we stop the current run and attribute the error to data corruptions. Another possible outcome of injected faults is an abnormal termination of the `md5sum` process; in this case, a zero-sized `target` file is generated so that we classify the error into crashes. For each run, we keep injecting register faults until an error happens (a mismatch or an abnormal termination) or one of our kernel error detection mechanisms halts the system. A random delay is inserted between two injected faults. We specify the fault injection conditions to ensure that only the Linux virtual machine is affected by the faults. Note that we cannot guarantee that every injected fault impacts the `md5sum` process since doing that requires analysing the guest Linux VM kernel data structures. However, since the `md5sum` process dominates the CPU usage, there is a high possibility that the injected faults affect the `md5sum` process.

In Table 7.12, we present the fault injection results that include the number of faults injected to each general-purpose register (EAX to EIP) and the total number of faults injected (*Total Injected*). The column *Base* is for the unprotected single-core VM, and we classify the failures as *Crashes* (`md5sum` failed to produce a hash value) and *Corruptions* (`md5sum` produced an incorrect hash value). The *DMR* column represents the protected DMR VMs managed by hardware-assisted CC-RCoE, and we categorise the errors detected by the two error detection mechanisms, *Timeouts* (kernel barrier timeouts) and *Mismatches* (execution fingerprints are inconsistent). In this experiment, we only injected register bit flips into the primary replica of the DMR VMs. We categorise the data corruptions and crashes as *Uncontrolled Failures* and the halts triggereed by timeouts and mismatches as *Controlled Failures*.

For the unprotected configuration, a significant portion of the failures is data corruptions that account for approximately 64.5% of the total uncontrolled failures. The high data corruption rate can be attributed to the fact that the steps for computing the hash values are very sensitive to variations of input data and intermediate computation values. In contrast, the protected DMR configuration does not allow data corruptions to go out of the system by timely halting the system when the fingerprint comparison fails (96.04%). The kernel-barrier timeout mechanism (3.96%) catches the errors caused by execution divergence.

This experiment demonstrates the effectiveness of our approach in terms of detecting application-

	Base	DMR
EAX	626 (21.80%)	624 (22.19%)
EBX	259 (9.02%)	249 (8.86%)
ECX	47 (1.63%)	35 (1.24%)
EDX	647 (22.53%)	648 (23.04%)
ESI	265 (9.23%)	257 (9.14%)
EDI	48 (1.67%)	39 (1.37%)
EBP	641 (22.32%)	636 (22.62%)
ESP	301 (10.48%)	288 (10.24%)
EIP	38 (1.32%)	36 (1.28%)
Total Injected	2872	2812
Crashes	887 (35.5%)	0
Corruptions	1613 (64.5%)	0
Timeouts	N/A	99 (3.96%)
Mismatches	N/A	2401 (96.04%)
Uncontrolled Failures	2500	0
Controlled Failures	0	2500

Table 7.12: Fault injection results. Random bit flips are injected into user-mode registers when `md5sum` is running.

level divergence and catching errors that causing erroneous output data, significantly reducing the chances of data corruptions.

7.6.4 Fault Injection Campaign against the Redis System

Random memory faults

In this experiment, we use a spare CPU core to perform fault injections into memory. While our goal is to handle relatively rare transient faults or SEUs, our fault model in this experiment injects multiple SEUs to compress the time required to run the campaign. The campaign repeatedly injects until the system under test fails, restarts the system, and then continues fault injections. The victim physical address and bit to be flipped are chosen randomly based on the random number generator in Section 7.6.2. The delay time between two fault injections is also chosen randomly between 3,489,660,928 to 4,294,901,760 CPU cycles for x86-64 and between 268,435,456 to 4,294,901,760 cycles for ARM, as reported by the `rdtsc` and `c15` cycle counter registers on x86-64 and ARM respectively. We rely on having enough samples to observe overall trends in the data.

The system under test is the Redis and YCSB benchmarking software used previously. The YCSB benchmarking client is modified to embed CRC32 checksums of the key-value pairs into the values written to the Redis server. The YCSB client can then validate the correctness of data returned by Redis by comparing the embedded checksum with a recalculated checksum. We test an unprotected and various protected versions of the system. A script monitors the outputs from the Redis server and YCSB client. Once the script detects the Redis server failure, errors reported by YCSB client, or errors reported by our error detection mechanisms, it logs the reason for failure and restarts the Redis server and YCSB client.

Table 7.13 and Table 7.14 show the total number of fault injections performed over all runs for an unprotected Redis configuration and the protected configurations managed by LC-RCoE, and the breakdown of the observed failures. The difference in number of faults injected is not indicative of anything other than length of time each experiment has executed. We experiment with two different fault injection strategies for the x86-64 and ARM machines. On the x86-64 machine, we choose to inject single-bit flips into the physical memory regions used by the kernel replicas (including the code and data sections for all kernel replicas and the kernel shared memory region for building the synchronisation protocol) and the regions allocated to the primary replica (including the applications and the DMA buffers allocated to the I/O devices). As to the ARM machine, the memory addresses targeted for fault injections are all memory regions used by the unprotected, DMR, or TMR system. We use the same sequences of seeds for the random number generator (RNG) that generates the memory addresses and positions in data words to inject faults for the baseline and protected systems of each machine. The ARM machine has two additional configurations, DMR-N and TMR-N, which do not include user-mode driver contributed updates into the the fingerprints. For our particular system, the checksums of the out-going network packages are not included in DMR-N and TMR-N. The protected configurations of x86-64 and ARM run in LC-RCoE. The meanings of the first column in the tables are explained below.

Total Injected Faults: The number represents the total single-bit flips injected into various memory locations. Note that not all injected faults manifest as errors; for instance, the faults injected at unused memory locations do not introduce errors.

Observed Errors: The number of observed errors caused by the injected faults. We categorise the errors to the following classes: (1) user-mode exceptions (*User VM Exceptions* and *User Other Exceptions*), (2) kernel-mode exceptions, (3) errors reported by the YCSB (*YCSB Corruptions* and *YCSB Errors*) , and (4) errors spotted by our error detection mechanisms (*KBarrier Timeouts* and *Fingerprint Mismatches*). (1), (2), and (3) are called *uncontrolled errors* that we aim to reduce. In our protected configurations, we attribute the errors in (4) as *controlled errors* that are reported when our framework observes inconsistency between the replicas. The remedy for a controlled error is a graceful fail-stop, avoiding replying the YCSB with corrupted data. The details of the observed errors are listed below.

User VM Exceptions: User-mode invalid virtual memory exceptions triggered by applications. A memory access (be it a data access or an instruction fetch) beyond the ranges of of valid data or code sections is reported as this error type. Note that on-demand paging is not treated as an error.

User Other Exceptions: Other user-mode exceptions triggered by applications. For instance, invalid instruction exceptions belong to this error type. Also, the exceptions set off by the root task are taken into this type.

Kernel Exceptions: Various exceptions occurred in kernel mode.

YCSB Corruptions: The number of results returned by Redis that are not correct. Remember that the YCSB client validates data integrity by comparing the embedded checksums and recalculated checksums; thus, inconsistent checksums are reported and categorised as data corruptions.

YCSB Errors: Run-time exceptions (e.g., SocketTimeoutException, JedisDataException, or StringIndexOutOfBoundsException) reported by the YCSB client.

KBarrier Timeouts: A graceful fail-stop because of a kernel-mode barrier timeout in the protected configurations. A consequence of one of the cores becoming non-responsive or the cores diverging.

Fingerprint Mismatches: A graceful fail-stop because of a fingerprint comparison failure in the protected configurations.

	Base	DMR	TMR	DMR-N	TMR-N
Total Injected Faults	243277	201558	183794	223776	214387
Kernel Exceptions	0	0	0	0	0
User VM Exceptions	291 29.1%	0	0	0	0
User Other Exceptions	5 0.5%	0	0	0	0
YCSB Corruptions	647 64.7%	3 0.3%	1 0.1%	381 38.1%	299 29.9%
YCSB Errors	57 5.7%	1 0.1%	0	13 1.3%	10 1.0%
KBarrier Timeouts	N/A	304 30.4%	304 30.4%	602 60.2%	678 67.8%
Fingerprint Mismatches	N/A	692 69.2%	695 69.5%	4 0.4%	13 1.3%
Observed Errors	1000	1000	1000	1000	1000
Uncontrolled Errors	1000 100%	4 0.4%	1 0.1%	394 39.4%	309 30.9%
Controlled Errors	N/A	996 99.6%	999 99.9%	606 60.6%	691 69.1%

Table 7.13: Number of system failure type occurrences and percentages of total failures (ARM)

	Base	DMR	TMR
Total Injected Faults	60449	91033	92299
User VM Exceptions	832 36.033%	2 0.086%	2 0.085%
User Other Exceptions	339 14.682%	0	0
Kernel Exceptions	0	0	3 0.128%
YCSB Corruptions	1001 43.352%	11 0.470%	16 0.684%
YCSB Errors	137 5.933%	6 0.256%	4 0.171%
KBarrier Timeouts	N/A	1238 52.906%	1184 50.598%
Fingerprint Mismatches	N/A	1083 46.282%	1131 48.333%
Observed Errors	2309	2340	2340
Uncontrolled Errors	2309 100%	19 0.812%	25 1.068%
Controlled Errors	N/A	2321 99.188%	2315 98.932%

Table 7.14: Number of system failure type occurrences and percentages of total failures (x64)

In the unprotected case, we see that many failures are application crashes due to user-mode exceptions (29.6% for ARM and 50.715% for x86-64). We also observe a significant fraction of failures as errors propagated as incorrect results returned to YCSB (64.7% and 43.352%) or errors that caused YCSB to throw Java run-time exceptions (5.7% and 5.933%). The high percentages of YCSB corruptions demonstrate the severity of silent data corruptions.

The results confirm that the our error detection mechanisms successfully convert most of the uncontrolled errors to controlled errors for the DMR (99.6% and 99.188%) and TMR (99.9% and 98.932%) configurations on both ARM and x86-64 machines. Although all the protected DMR and TMR configurations failed to eliminate YCSB data corruptions completely, the improvements are momentous: the percentages of YCSB corruptions were reduced from 64.7% to 0.3% (DMR) and 0.1% (TMR) for ARM and from 43.352% to 0.470% (DMR) and 0.684% (TMR) for x86-64. We attribute the remaining corruptions due to the fact that the DMA buffers used by network drivers are not redundant so that a data corruption in the buffers can cause all the replicas observe incorrect data. The x86-64 machine has slightly higher data corruption percentages due to the following two factors: (1) The DMA buffer size on x86-64 machine is several times larger than the buffer size of the ARM machine. (2) The fault injection addresses for x86-64 are restricted to the memory regions used by the kernel replicas and the primary replica, but the addresses for ARM are randomly selected from the memory regions used by all replicas. The higher YCSB error percentages of the x86-64 DMR and TMR configurations can also be explained by the second factor. Furthermore, the design choice of allowing applications to contribute data into the fingerprints and to trigger comparisons pays off: the network drivers of the protected configurations contribute checksums of output data into the fingerprints for comparisons, resulting in efficient captures of corrupted output data (69.2% and 69.5% for ARM, and 46.282% and 48.333% for x86-64). In contrast, the DRM-N and TMR-N configurations, which exclude the checksums of out-going network packets, still suffer from YCSB corruptions (38.1% and 29.9%) and show low fingerprint mismatch percentages (0.4% and 1.3%).

The three kernel exceptions occurred on the x86-64 machine deserve our attention. We analyse the log files and find that two of errors were caused by bit flips at the 5th bit of physical memory address `0x36bd03`. The reason for the two errors being caused by exactly the same fault is that the kernel RNG was coincidently seeded with the same number in two different runs. That particular address was in the text section of the second kernel replica, and the fault changed the instruction at the address from `cmp $0x1, %edx` (`0x83 0xfa 0x01`) to `.byte 0xa3 cli .byte 0x1 (0xa3 0xfa 0x01)`. Evidently, the bit flip changed the victim byte from `0x83 (0b1000 0011)` to `0xa3 (0b1010 0011)`, causing an invalid instruction in the kernel function `decodeX64MMUInvocation`. We also look into the log for the DMR configuration and find that the same physical address, `0x36bd03`, was affected as well. However, the address was located in the shared kernel data region of the second kernel replica. As described in Section 4.2.1, that particular physical memory region was not used; thus, the fault did not ignite a kernel exception. As to the baseline configuration, the address fell into the memory region for the applications so that no kernel-mode exception was observed. Another kernel exception was caused by a kernel-mode page fault. We do not observe faults injected into kernel memory regions during that run. However, according to the faulting instruction and kernel source code, we identify the potential cause: a bit flip changed the type of a capability from `endpoint_cap` to `irq_handler_cap` so that an invalid pointer was returned; thus, the kernel faulted when it was referencing the invalid pointer. The seL4 formal verification proves that there are no kernel exceptions (assuming correctly-function hardware), and the default handler for kernel exceptions halts the system. Therefore, in a sense, the three kernel exceptions could be also categorised as controlled errors. To detect such errors, we added additional kernel barriers in the kernel exception handlers when executing the ARM fault injection experiment; thus, we do not observe any kernel exceptions on ARM. According to the log, several kernel data aborts did happen, but they were caught by kernel barrier timeouts.

It is worth mentioning two arguments related to our fault injection results. The first one is that a DMR system built with the same hardware components with identical FIT (failure in time) rates as in an unprotected system is more likely to fail because the doubled hardware components lead to higher overall system failure rate. The kernel exceptions of the x86-64 TMR configuration, to some extent, support the argument. However, we need to recognise that the DMR or TMR configurations can detect errors that can cause various failures ranging from system resets to silent data corruptions. The key factor is that the protected configurations have the opportunity to handle a detected error before the error potentially causes severe consequences. The second argument is that the approach of end-to-end checksumming used by the modified YCSB client is able to detect data corruptions so that the DMR or TMR configurations are not necessary in terms of preventing data corruptions. We agree that end-to-end checksumming is effective in protecting data integrity. Notwithstanding, we also perceive the possibility that some data can be corrupted before a checksum is calculated for the data; thus, a checksum is computed for the corrupted data. In this case, the end-to-end approach cannot identify such corruptions.

Still, the very small percentages of YCSB errors in fault injection experiments reveal an issue: when the primary replica stops functioning, the primary replica receives no interrupts and ceases triggering synchronisations. The consequence is that if there is no active or pending synchronisation request when the the primary replica halts, the other replicas cannot observe the failure of the primary replica unless the other replicas are already blocked or are going to block on a kernel barrier for I/O operations or output comparisons. Subsequently, the other replicas eventually switch to the idle thread because of the lack of external input events. A system under such condition is unable to receive inputs or to produce outputs, and thus we deem the system to be halted. This issue can be resolved by distributing interrupts to all replicas and allowing each replica to initiate synchronisations once it receives a device interrupt. However, we have not fully evaluated the performance impact of such an approach.

Random CPU faults

Overclocking processors has negative effects on the reliability of the processors [Memik et al., 2005]. We overclock the Cortex-A9 cores of the Sabre board to around 1.09 GHz, aiming to trigger random CPU faults or anomalies in various processor components. Note that we have no control over the locations and timings of the faults, so it is possible that multiple faults occur in several components during a very short interval. The system under test is the same as the above Redis system except that we adopt overclocking cores instead of corrupting memory to induce random glitches. The results for the baseline, DMR, and TMR systems are shown in Table 7.15. Both the protected DMR and TMR systems are managed by LC-RCoE, and the error detection configuration is BT (barrier timeout) + CU (critical updates) + AC (application contribution) + SC (system call). The meanings of the error types in the table are the same as above, and we remove the row named “Total Injected Faults” since we cannot accurately collect the data for random CPU faults. We can only observe errors, but we cannot determine the causes of the errors.

	Unprotected		DMR		TMR	
User VM Exceptions	632	63.2%	0		0	
User Other Exceptions	345	34.5%	0		0	
Kernel Exceptions	2	0.2%	0		0	
YCSB Corruptions	1	0.1%	0		0	
YCSB Errors	20	2.0%	25	2.5%	24	2.4%
KBarrier Timeouts	N/A		724	72.4%	853	85.3%
Fingerprint Mismatches	N/A		251	25.1%	123	12.3%
Observed Errors	1000		1000		1000	
Uncontrolled Errors	1000	100%	25	2.5%	24	2.4%
Controlled Errors	N/A		975	97.5%	976	97.6%

Table 7.15: Number of system failure type occurrences and percentages of total failures when the ARM SoC is overclocked

We have the following observations based on the results in Table 7.15.

- For the unprotected baseline system, a majority of the observed errors are various user-mode exceptions. However, there is still one data corruption.
- The percentage of YCSB corruptions for the unprotected system is significantly lower than the corresponding percentage in Table 7.13. The main reason for this situation is that the system crashed relatively soon after we increased the frequency so that the YCSB client did not have the opportunity of executing the benchmark stage during which the data was read back and checked for any potential corruptions.
- The percentages of YCSB errors of the DMR and TMR configurations are higher than the corresponding percentages in Table 7.13. I examined the log files and determined that 6 of the total 49 errors were caused by system reboots induced by overclocking. The remaining 43 errors reported were various network exceptions, indicating unresponsive failures. We expect that our error detection mechanisms may not be able to handle the cases in which multiple faults affect several components within a short interval.
- Both the DMR and TMR configurations report nontrivial numbers of fingerprint mismatches. We speculate that these mismatches were triggered by register errors.

In summary, the overclocking experiment reveals one important but expected limitation of our error detection schemes: when multiple components experience errors in a relatively short period, the system may enter a state of complete failure that is beyond the capability of our software-implemented mechanisms to handle. Fortunately, this kind failure is relatively straightforward to handle: a watchdog constantly monitors the status of the system and reboots the system when it misses heartbeats.

7.7 Error-Masking Experiments

The error-masking mechanisms are evaluated on the x86-64 machine for both CC-RCoE and LC-RCoE and the ARM SoC for LC-RCoE only. As we mentioned in Section 6.5, we only try to recover from a fingerprint mismatch for a TMR system. Our experiment method is to trigger a fingerprint mismatch while the TMR system is running, and we specifically remove the primary replica, exercising the primary replica migration mechanism described in Section 5.7. The system under test is still the Redis server; YCSB is also used to generate the workload (workload B). For each configuration (CC-RCoE or LC-RCoE), we perform two runs that remove the primary replica and one non-primary replica respectively. Approximately, the system is programmed to trigger a fingerprint mismatch 60–70 seconds after the kernel brings up the root task.

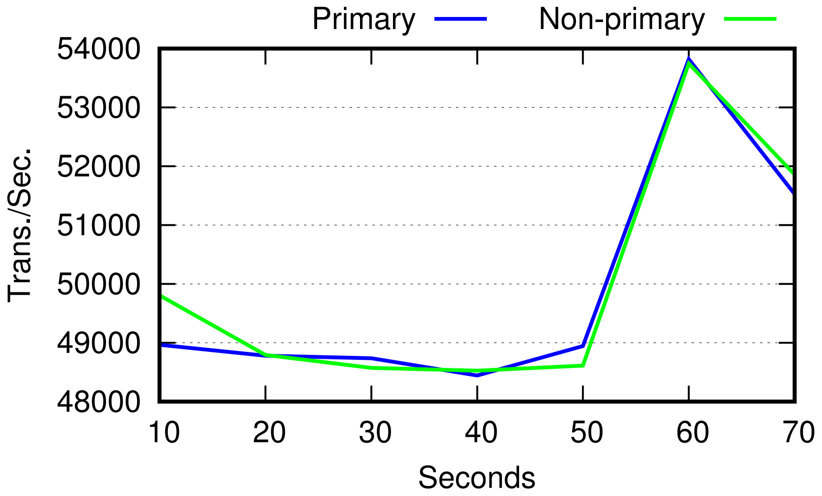

Figure 7.16: Redis throughputs with error masking on x86-64 (LC-RCoE)

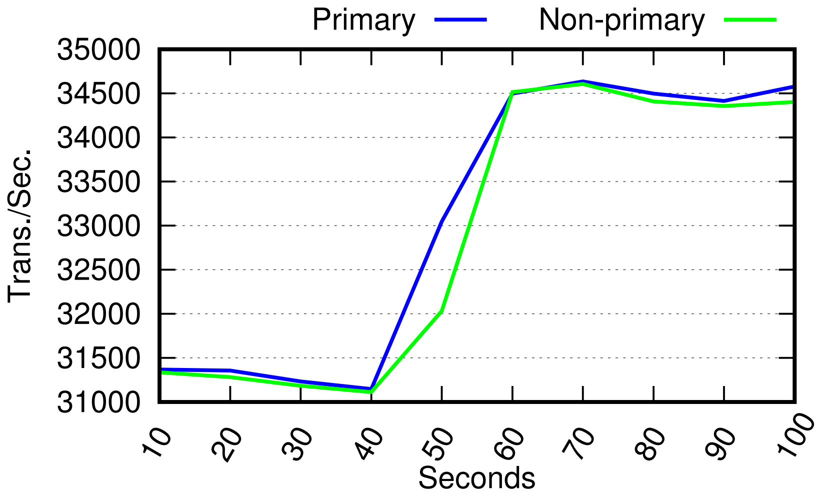

Figure 7.17: Redis throughputs with error masking on x86-64 (CC-RCoE)

In Figure 7.16 and Figure 7.17, we show the throughputs reported by the YCSB load generator every 10 seconds for the x86-64 machine. The x-axis is the execution time, and the y-axis represents the throughputs. *Primary* and *Non-primary* designate the replica to be removed. Clearly, we can observe that the throughputs go up significantly during the intervals of 50s to 60s, which conform to the time when the mismatches are triggered. (The time for starting up the Redis system and populating the database takes around 15 seconds.) The increases in throughputs are because that the system downgrades from TMR to DMR by removing the faulty replica which holds the incorrect fingerprint. We also demonstrate that removing the primary replica, which is responsible for performing I/O operations, does not cause service interruptions when the criteria for error masking are satisfied.

CC-RCoE Primary	CC-RCoE Non-primary	LC-RCoE Primary	LC-RCoE Non-primary
2,869	3	532	8

Table 7.16: Measured time (microseconds) for error-masking operations (x86-64)

Table 7.16 shows the measured time for the error-masking operations for the 4 particular runs reported in the figures above. Note that the time for error-masking operations varies from run to run and can change significantly if hardware and software configurations change. However, these operations are not critical for performance since normal execution does not invoke the error-masking operations. Removing the primary replica while the system is running in CC-RCoE costs the most (2,869 microseconds). As we have discussed in Section 5.7.3, primary replica migration involves patching the page tables of the device drivers running on the new primary replica so that the drivers can access the DMA buffers that are allocated in the memory regions of the former primary replica. The patching step examines the page table entries one by one and updates some entries if necessary. Thus, the execution time of the patching step is linear to the number of the page table entries used by the drivers. In our case, we only need to patch the page table entries of the network driver. The cost (532 microseconds) for removing the primary replica of the system in LC-RCoE is also more than that of removing non-primary replicas. This is because rerouting hardware interrupts to the new primary replica is still needed, although patching page tables is not required for LC-RCoE. The cost of reprogramming interrupt controllers is specific to the controller type and to the accessing method (e.g., memory-mapped or I/O port). The reported time includes the cost for reprogramming IO-APIC through I/O ports. The costs (3 and 8 microseconds) for removing the non-primary replica are relatively low since they do not need to migrate the primary replica, and the costs are mainly from voting and deactivating the faulty replica. The LC-RCoE has a slightly higher cost since it needs to inform the user-mode drivers about the removal of the non-primary replica through the bidirectional communication channel (Section 5.7.2).

LC-RCoE Primary	LC-RCoE Non-primary
2,621	21

Table 7.17: Measured time (microseconds) for error-masking operations (ARM)

We perform similar experiments on the ARM board with the LC-RCoE TMR configuration. As we mentioned in Section 5.7.3, due to the hardware limitation, we have not added primary-replica migration support (Section 5.7) for CC-RCoE on the ARMv7-based board. In Figure 7.18, we can observe similar throughput increases during the interval of 40s to 60s, for the same reason mentioned above. The costs for error-masking activities are listed in Table 7.17; and not surprisingly, removing the primary replicas takes much longer than removing the non-primary replica because of the steps for performing primary replica migration in which the ARM generic interrupt controller is reprogrammed and pending interrupts are saved.

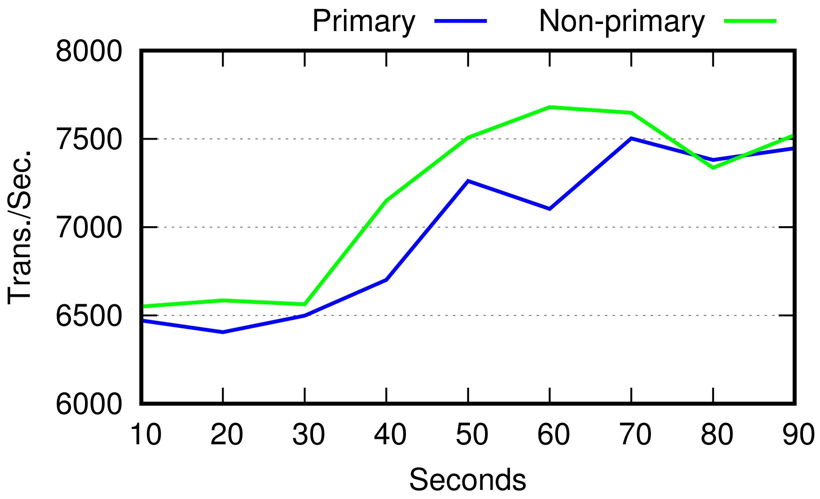

Figure 7.18: Redis throughputs with error masking on ARM (LC-RCoE)

Overall, although there are various restrictions imposed on the error-masking operations, we demonstrate that a viable forward recovery approach can be constructed based on existing kernel mechanisms for error detection and primary replica migration, with reasonable cost. Furthermore, our whole-system redundant co-execution creates possibilities for exploring other recovery approaches or reintegration (Section 6.5.3).

7.8 Summary

In this chapter, we first evaluated the performance characteristics of the resulting systems, using the single-core unprotected system as the baseline. The microbenchmarks running directly on seL4 reveal the following observations when LC-RCoE is adopted: (1) For CPU-bound workload, the overhead of redundant co-execution is very small. (2) For memory-intensive workload, the redundant co-execution significantly reduces the available memory bandwidth for each core. (3) The interrupt latency is prolonged by the synchronisation protocol coordinating the replicas. Further, we delineated the traits of hardware-assisted CC-RCoE by running Dhrystone, Whetstone, SPLASH-2, and RAMspeed on the baseline Linux VM and the DMR Linux VMs. For Dhrystone and Whetstone, which respectively measure integer and floating-point performance of processors, we observed notable increases in execution time (1.50 and 2.89 times of the baseline VM). LU-C and LU-NC from SPLASH-2 exhibited severe performance degradation (8.14 and 6.23 times of the baseline execution time), and the remaining SPLASH-2 applications showed varied increases in execution time in the range of 1.10–3 times. The geometric mean of the increases in execution time for all applications is 2.02 times. RAMspeed demonstrated that the overhead for running memory-bound applications on the DMR VMs is significant, since two Linux VMs running on different cores compete for the memory bandwidth that is otherwise monopolised by the baseline VM. We attribute the overhead to the fact that CC-RCoE needs hardware breakpoints to stop the replicas exactly at the same instruction, so applications containing tight loops are likely to be slowed down if the breakpoints happen to be at instructions inside tight loops.

We used Redis as the system benchmark to assess the performance when our approach is applied in a real-world scenario. The LC-RCoE-based DMR and TMR systems configured with our recommended error detection options (LC-DMR-SC and LC-TMR-SC in Table 7.10) achieved 74%–79% and 68%–73% of the baseline across 6 workloads on the x86-64 machine; the Cortex-A9 ARM board achieved 77%–79% (LC-DMR-SC) and 69%–71% (LC-TMR-SC) of the baseline. Given that the Redis benchmark exercises the cores, memory, and I/O devices (thus high interrupt frequency), we consider the resulting system managed by LC-RCoE performs reasonably well.

We conducted fault injection campaigns on real hardware to determine the effectiveness of the error detection mechanisms. In the fault injection experiment targeting user-mode registers, our protected DMR VM system successfully prevented corrupted MD5 digests from escaping the system. In the campaigns injecting single-bit flips into various memory addresses, protected configurations significantly reduced the uncontrolled errors; more importantly, the data corruptions were lowered from 64.7% to 0.3% (DMR) and 0.1% (TMR) for ARM and from 43.352% to 0.470% (DMR) and 0.684% (TMR) for x86-64. We also demonstrated that allowing user-mode drivers or applications to contribute critical state updates into execution fingerprints for comparisons unquestionably helped error detection. The CPU overclocking experiments, which induced random CPU faults, illustrated a limitation of our approach: when multiple components experience errors in a relatively short period, the system may enter a complete failure that is beyond the capability of our software-implemented error detection mechanisms. However, our DMR and TMR systems still showed much lower uncontrolled error percentages—2.5% and 2.4%. Finally, we performed the error-masking experiments to demonstrate that a system running in TMR can be gracefully downgraded to DMR without interrupting services when a faulty replica is detected by comparing fingerprints and when the criterion for performing error masking are satisfied.

In summary, LC-RCoE systems usually outperform CC-RCoE systems. The hardware-assisted CC-RCoE on x86 machines has the advantage of running replicated Linux virtual machines that can be leveraged to support existing applications, although the performance suffers. More importantly, our error detection mechanisms perform well in terms of reducing data corruptions and uncontrolled crashes.

Chapter 8

Summary and Looking Forward

> We can only see a short distance ahead, but we can see plenty there that needs to be done.
>
> Alan Turing

We conclude the thesis by reiterating the problem and our approaches, followed by examining potential directions for performance improvements and better error recovery mechanism. We recognise that transient hardware faults still pose serious threats to COTS computer systems that require a high level of security and data integrity, despite that many software-implemented approaches have been proposed to detect or tolerate errors caused by transient hardware faults. One critical software component ignored by existing software-implemented fault-tolerant approaches is the lowest-level system software that plays a crucial role in ensuring safe and fair hardware resource sharing. The main theme of the thesis is to address the need by improving the formally verified seL4 microkernel with self-checking capabilities. Arguably, our proposed mechanisms, especially the synchronisation protocol, may also be adopted by other microkernels. We choose the seL4 kernel mainly because its unique property—its C implementation is bug-free (conditions apply) and is guaranteed to follow the design specifications; the microkernel already demonstrated its advantages in terms of building real-world secure systems [Fisher, 2014].

8.1 Summary

8.1.1 Transient Hardware Faults

Our journey begins with single-bit flips; we described how alpha particles or neutrons interact with silicon crystals and how a transient current triggered by a particle striking a sensitive node of a transistor can cause a single event upset (SEU) or single event transient (SET) (Section 2.1). Transient hardware faults have significant impacts on the reliability of CMOS circuits. Scaling feature sizes, reducing supply voltages, lowering noise margins, and increasing clock rates make CMOS devices more susceptible to transient faults. However, the demands for state-of-the-art computing capacity and technology, dramatically reduced costs, improved energy consumption, and decreased physical volumes motivate system designers to build fault-tolerant, safety- and/or mission-critical systems based on COTS hardware instead of using customised or radiation-hardened hardware components, which are generations behind the up-to-date COST components in terms of raw computation power and functionalities.

Transient faults impact the dependability of the systems; as shown in previous studies, transient-fault-induced errors can introduce security vulnerabilities and system failures. Despite that technology advancements improve the soft error rate of COTS hardware, transient hardware faults are not going to disappear completely in the foreseeable future. Even worse, multi-cell faults are also becoming an urgent issue since that SEC-DED ECC is unable to correct a multi-cells fault in a data word (a multi-bit fault) and that the replication-based error detection approaches assuming single-bit flip fault model may not be able to handle such faults well.

8.1.2 The Mechanisms for Whole-System Redundant Co-Execution

Various software-implemented fault-tolerant approaches, aiming for protecting applications, assume that the low-level system software is protected by other measures or is free from hardware faults. The system software (kernels, hypervisors, etc.) ensures safe and fair hardware resource sharing, so its correct operation has direct impacts on the whole system security; a soft error affecting the system software has more significant negative effects on the whole system dependability.

Our answer to the challenge is whole-system redundant co-execution that exploits the hardware redundancy provided by multicore processors, instantiates a whole-system replica (including the microkernel, device drivers, and applications) onto different cores, redundantly executes the replicas, and compares the replicas for error detection. Applying SMR (state machine replication Section 4.1.2) to a whole software system requires taming non-deterministic events (Section 2.4.5) in the system, so we design a synchronisation protocol and various microkernel mechanisms for instantiating and managing the replicas of the whole software system on different cores. The protocol (Section 4.3) coordinates the execution of the replicas, and we implement two prototypes of the protocol on ARM and x86 platforms. Loosely-coupled redundant co-execution (LC-RCoE Section 4.5) relies on kernel-maintained deterministic event counters to measure the progress of the replicas and only supports single-threaded applications or multi-threaded applications that are data-race-free. Closely-coupled redundant co-execution (CC-RCoE Section 4.4) is able to precisely preempt the replicas as the same position in the instruction streams so that it can support multi-threaded applications (even with data races), but it requires hardware performance counter support on the x86 machines (Section 4.4.2) and recompilation of all user-mode code with the GCC plugin on ARM machines (Section 4.4.3). A comparison of LC-RCoE and CC-RCoE is presented in Section 7.5.

To support device driver replication, we introduce cross-replica shared memory regions (Section 5.4.1) for LC-RCoE and two new system calls (Section 5.5.1) for CC-RCoE to facilitate the implementation of the access patterns (Section 5.2) that coordinate the accesses to I/O devices and DMA buffers. Essentially, the replicated drivers function as the input data duplicators and output data comparators. We call the replica that actually performs the I/O operations in the access patterns as the primary replica. For a TMR system, if error masking is enabled and an error is detected by comparing execution fingerprints, the system downgrades to the DMR mode by removing the faulty replica. If the faulty replica is the primary replica, the system needs to conduct a procedure called primary replica migration (Section 5.7) that retires the old primary replica and migrates the functions to the new primary replica. The necessary kernel supports for the migration is illustrated separately for LC-RCoE and CC-RCoE: the bidirectional communication channel (Section 5.7.2) and the mechanism (Section 5.7.3) for dynamically patching the page table entries that cover the virtual memory addresses used as DMA buffers.

8.1.3 The Mechanisms for Error Detection and Masking

Based on the solid foundation provided by redundant co-execution and driver replication, we introduce two kernel mechanisms for detecting errors: execution fingerprint comparison (Section 6.3.1) and kernel barrier timeout (Section 6.3.2). Internal kernel functions (Section 6.2.1) are provided to collect kernel state updates into the fingerprints and trigger the comparisons to validate the fingerprints. The microkernel is modified to employ the functions for including virtual-memory-space and capability-space updates (Section 6.2.2), and system calls (configurable) into the fingerprints. The new system call, `seL4_FT_Add_Event(value, compare_now)`, allows user-mode processes to contribute their internal data into the fingerprints. Specifically, device drivers use this system call to validate checksums of output data before they release the data to I/O devices. The kernel barrier timeout catches the execution divergence triggered by transient faults, and it is effective in detecting the errors causing "stuck-at" situations.

Error masking can be optionally enabled for the TMR configuration so that the system can mask an error detected by fingerprint comparisons with forward recovery (Section 6.5); The algorithm (Section 6.5.1) for voting the faulty replica is crucial. The algorithm cannot assume that only one replica is faulty, neither can it assume that the voting procedure is free from faults. Thus, the algorithm is designed to be executed redundantly, and it finishes successfully only when all the replicas reach an agreement on the faulty replica ID; the algorithm halts the system otherwise. If the faulty replica is not the primary replica, we can simply remove the replica; otherwise, we need to conduct primary replica migration first (Section 5.7).

8.1.4 Realisations and Evaluation

We have implemented our redundant co-execution, error detection, and error masking mechanisms based on the seL4 microkernel for ARM and x86 machines. The performance overheads vary according to the types of workloads and the redundant co-execution mode (CC-RCoE or LC-RCoE) chosen by a system. Applications demanding high memory bandwidth may suffer from noticeable slowdown (Section 7.3.1). The obvious worst case is applications whose performance is limited by memory bandwidth. This limited bandwidth must now be shared between the replicas, resulting in roughly a factor-two slowdown for DMR, and a factor of three for TMR. This slowdown is an inevitable result of redundant co-execution, and the good news is that our design imposes little performance cost in this case. As shown in Section 7.3.1, for computation-intensive programs with small cache footprints, the overhead is mostly negligible if the system is running in LC-RCoE mode. Nevertheless, CC-RCoE may cause significant performance degradation for applications consisting of tight loops. Furthermore, the slowdown can vary significantly across runs, depending on the positions of the instruction breakpoints. The cost of programming debug registers and handling instruction breakpoint exceptions is the main reason for the slowdown, especially when the breakpoints are in loops.

In addition to the synthetic benchmarks, we also adopt a real-world server application, Redis, to benchmark the resulting systems (Section 7.4.1); we observe modest degradation of throughputs and consider that the systems are suitable for practical uses. Finally, the fault injection experiments demonstrate the effectiveness of our error detection mechanisms by showing that a dominating portion of the errors induced by bit flips in registers or memory is captured by the error detection mechanisms; and thus the data corruptions are eliminated (Section 7.6.3) or significantly reduced (Section 7.6.4).

8.1.5 Application Areas

Based on the characteristics of LC-RCoE and CC-RCoE, we envision that the following areas (including but not limited to) are good candidates for adopting our approach.

- CubeSats [Toorian et al., 2008] are very small and low-cost satellites with restricted size (10 cm × 10 cm × 10 cm) and weight (1.33 kg) requirements. Thus, COTS hardware is widely adopted in order to meet the size, weight, and cost constraints [Gregorio and Alimenti, 2018]. Our LC-RCoE for ARM SoCs should be able to run on the payload computers, which perform image processing, data analysis, data compression, etc. If the real-time requirements of the primary computer can be achieved through careful design and measurement, an ARM SoC powered by LC-RCoE can potentially be used as well. The tasks for the payload and primary computers are usually well defined and static, so it is possible to conduct extensive analysis to ensure that the resulting system meets various requirements despite that the LC-RCoE adds non-trivial overhead in trade for improved reliability in harsh space environments.

- The hardware-assisted CC-RCoE on x86 is capable of running virtual machines, thus it can be used to built a secure computing platform (e.g., NetTop [Meushaw and Simard, 2000]) on which classified data can be processed in conjunction with unclassified data. Usually, absolute performance and real-time response are not vital requirements for such systems, so the overhead of CC-RCoE is likely to be tolerated. Furthermore, seL4 represents a small trusted computing base (TCB), and CC-RCoE further enhances seL4 with the capability of protecting itself from transient hardware faults.

- Autonomous driving technologies require significant computing power, which necessitates the state-of-the-art COTS hardware for not only performance but also low costs and energy consumption. For instance, 10 Intel dual-core processors are used in Boss [Kim et al., 2013], an autonomous vehicle developed at CMU. Thus, the importance of tolerating transient hardware faults is rising since more and more electronic components are used in cars [Baleani et al., 2003]. Using replicated hardware increases the production costs and energy consumption significantly. The increased energy consumption directly translates to reduced fuel economy for petrol cars or shortened range for electric cars. Our LC-RCoE variants for x86 and ARM architectures may be used to alleviate the issue.

8.2 Road Ahead

8.2.1 Multicore Replicas

Nowadays, multicore processors completely dominate the COTS market; four-core and eight-core CPUs are common. In the DRM and TMR configurations, we only use two and three cores for redundant co-execution, leaving other cores idle. The need to utilise the idle cores for improved performance is both apparent and urgent.

Extending the verified seL4 kernel to support multicore processors is introduced by Elphinstone et al. [2016]. The main idea is using a big kernel lock, as shown on the left-side of Figure 8.1, to restrict the concurrency in the kernel. In the figure, both cores, C_0 and C_1, are managed by the kernel, and the *big lock* ensures that only one of two cores is in kernel mode at a time. Each core grabs the big lock before entering kernel mode and releases the lock before returning to user mode. Using a big lock to avoid concurrency is to comply with verification requirements. The multicore version of seL4 sets up a run queue containing all ready-to-run threads for each core and provides a new system call to specify the affinity of each thread (i.e., pinning a thread to a core). In the figure, T_0 and T_2 are pinned on the core C_0, and T_1 and T_3 are bound to the core C_1. The kernel does not try to balance work-load of the cores automatically by moving threads between cores, but relies on directives from a user-mode scheduler.

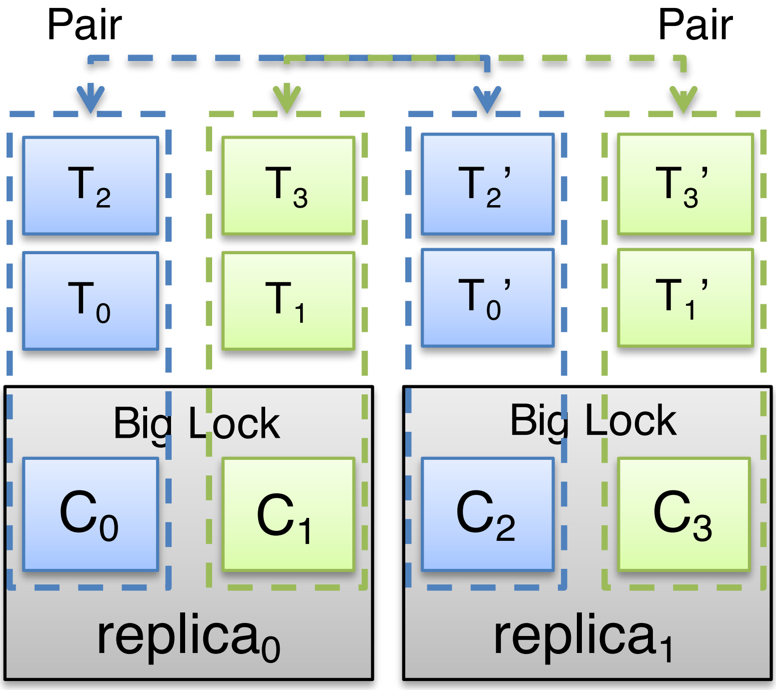

Figure 8.1: A multicore DMR example

Figure 8.1 shows a straightforward method to replicate the dual-core system, forming a *multicore DMR* configuration—each replica runs on two cores. In this configuration, before the replicas handle an interrupt, the states of the replicas have to be synchronised. In this figure, C_0 should be synchronised with C_2, so are C_1 and C_3. The dashed-line rectangles with different colours group cores belonging to different replicas together to form two *pairs*, and the we call the individuals making a pair as *items*. The synchronisation process is applied to each pair in order to achieve the multicore version of redundant co-execution. Because the existence of the big kernel lock in each replica, an item has to drop the lock when waiting on a kernel mode barrier so that other items in the same replica but belong to other pairs can enter kernel mode and make more progress. For example, when items C_0 and C_3 are waiting for items C_2 and C_1 to catch up, they have to release the locks so that C_2 and C_1 can proceed to catch up without blocking on lock acquisitions. In the case of a dual-core TMR configuration, we have three replicas, and each replica has two items. However, there are still two pairs, and each pair includes three items.

Now let's have a look how an interrupt is handled in the case of the dual-core DMR configuration. The following steps are executed:

1. A user-mode thread is interrupted by the external interrupt on the core C_0, and then, C_0 switches to kernel mode and grabs the big lock of $replica_0$.

2. C_0 triggers a round of synchronisation by calling the *trigger_action* function with the interrupt request number (IRQ) to handle. The function sends IPIs to all other cores to notify the pending synchronisation request.

3. C_0 releases the lock and begins the synchronisation mode. C_1, C_2, and C_3 receive the notifications and enter the synchronisation mode without holding the locks.

4. Each pair votes a leading item that will wait on the barrier, and the other item in a pair catches up until it makes the same amount work as the leading item. By doing so, we synchronise the pairs individually. When a pair is synchronised, it stops at the first pair barrier to wait the other pair to finish.

5. When all the pairs finish the synchronisation, they pass the first pair barrier. Then, the items in the $replica_0$ acquire and release the kernel lock deterministically according to the core IDs. After grabbing the lock, an item handles the interrupt and then releases the lock. By doing so, the interrupt is handled by all items in the replica sequentially and deterministically. The same steps are repeated on the other replica. The items stop at the second pair barrier to ensure that all replicas finish handling the interrupt.

6. Now the interrupt is handled by all replicas consistently, and the synchronisation process finishes.

Two *kernel pair barriers* must be added to coordinate the execution of the pairs. In Figure 8.1, the pair barriers require all four cores to arrive before any one of cores can proceed. However, the barriers used to coordinate the items inside a pair still require two cores.

8.2.2 Error Masking Support for Kernel Barrier Timeouts

As described in Section 6.5, error masking is only attempted for a TMR system when execution fingerprints mismatch; kernel barrier timeouts halt the system directly. We plan to enhance error masking so that downgrading from TMR to DMR can be optionally enabled for kernel barriers. Given a TMR system, we consider the following situations:

- Three replicas block on three different barriers. In this case, there is no simply way to identify which replica is faulty, since it is possible that more than one of the replicas are incorrect.
- Two replicas block on the same barrier. Apparently, in this case, two replicas correctly arrive at the barrier; the remaining replica, blocking on a different barrier or still running, could be the faulty one. Therefore, we can force the replicas to perform a majority voting, as described in Section 6.5.1. The voting results can also be discussed case by case:
  - The voting result identifies that the faulty replica is not one of the two replicas blocking on the same barrier. We remove the faulty one and resume normal execution in DMR.
  - The voting result shows that the faulty replica is one of two replicas blocking on the same barrier. We halt the system since there is a possibility that two replicas are incorrect.
  - The voting fails to identify a faulty replica because the fingerprints are the same. In this case, we consider the two replicas blocking on the same barrier are correct, based on the assumption that random hardware faults are unlikely to cause two replicas to fail the same way.
  - For other voting results, we halt the system.
- For other situations, we halt the system.

8.2.3 Tolerating Heisenbugs and Replica Checkpointing

A software bug that disappears or changes its symptom when one tries to debug the bug is called a Heisenbug [Gray, 1985]. Software that has gone through traditional testing-based quality assurance has few bugs that can be triggered deterministically. However, a Heisenbug may not be easily captured by various debugging methods (e.g., single-stepping, breakpoints, debug log), since the methods may just disturb the conditions that triggered the Heisenbug. Therefore, debugging Heisenbugs is frustrating and time-consuming.

We speculate a variation of redundant co-execution can help tolerating Heisenbugs. We briefly describe how we plan to use a variation of RCoE to tolerate Heisenbugs. Take a TMR system for example; two of the replicas are managed by LC-RCoE, and we call them *execution replicas*. The remaining replica is the *checkpoint replica*, which runs slightly behind the execution replicas and serves as a running checkpoint. The execution replicas run as a DMR configuration, but their inputs and checked outputs are stored and forwarded to the checkpoint replica, which takes the inputs and outputs, updates its own state, and creates checkpoints. When the execution replicas diverge or disagree on their outputs, we revert them back to the previous checkpoint based on the checkpoint replica and resume execution, hoping that the slightly different scheduling and timing of input data in the resumed run will not trigger the Heisenbug. Obviously, we need to design and implement checkpoint mechanisms, which can be used for building checkpoint-based recovery. Indeed, we need to address the issues of adopting checkpoint recovery as described in Section 6.5.

8.2.4 Real-time Applications

The synchronisation protocol (Section 4.3) holds up the leading replica and lets the chasing replicas to catch up; so during that period, no external event is delivered to applications for processing. The delayed delivery of external events affects real-time applications since it increases the latency between the occurrence of an event and the observation of the event by its handler. The latency introduced by the protocol can be measured (Section 7.3.1), but the intervals that the leading replica waits for the chasing replicas to catch up cannot be determined precisely, since they depend on how faraway the chasing replicas are left behind by the leading replica. The distance between the leading and chasing replicas continuously changes when the system is running so that the lengths of delays vary as well.

For CC-RCoE, the performance overhead can become significant if hardware breakpoints are set at instructions inside loops. CC-RCoE is not suitable for real-time applications because the overhead can vary dramatically and unpredictably. For LC-RCoE, the overhead during the catching up stage is constant for each kernel trap and linear to the distance; the cost of handling hardware breakpoints in tight loops does not exist. Thus, we conjecture that LC-RCoE can be used with some real-time applications through careful design, engineering, and validation.

8.2.5 Optimising the Performance of CC-RCoE

As we observed in Section 7.3.1 and Section 7.3.2, CC-RCoE pays the price of non-deterministic, increased execution time for precise preemption, when the applications being replicated contain many small and tight loops. We also conclude the slowdown is mainly caused by the instruction breakpoints, which are set at an instruction inside a tight loop, frequently triggering debug exceptions that are expensive to handle. Our plan is to reduce the overhead associated with handling debug exceptions, and it can be achieved by reducing the frequency of the debug exceptions triggered by instruction breakpoints.

The intuition is that if a chasing replica is tens of thousands of branches behind the leading replica, we can allow the chasing replica to run without triggering instruction breakpoint exceptions until it is very close (e.g., 50 branches behind) to the leading replica. Therefore, the implementation of CC-RCoE is slightly modified as below (assuming DMR).

- If the difference of branches between the leading replica and the chasing replica is less than an experimentally-determined value `N`, the original synchronisation steps are executed.
- Otherwise, the follow additional steps are executed to bring the chasing replica close enough before resuming execution of the original synchronisation steps.
  - A performance counter is programmed to trigger an exception when the chasing replica is approximate `N` branches behind the leading replica.
  - The execution of the chasing replica is resumed without setting an instruction breakpoint.
  - When a performance counter exception occurs, the exception handler in kernel mode sets the instruction breakpoint and resumes the original synchronisation steps.

As we can see, the additional steps allow the chasing replica to make more progress quickly, since a significant numbers of exceptions triggered by instruction breakpoints can be avoided. Therefore, the synchronisation protocol only needs to deal with the exceptions triggered by the remaining `N` branches. We need experiments to choose the value `N` properly for the following reasons: (1) The delivery of performance counter exception can be imprecise. (2) The performance counter can over-count or under-count. Furthermore, we need to confirm that programming an additional performance counter to trigger exceptions does not interfere with the accuracy of the existing counters for counting branches on x86. Note that both PMU-based and compiler-based CC-RCoE can be modified to adopt the optimisation.

8.2.6 Choosing Timeout Values for the Kernel-Mode Barriers

As we introduced before, choosing appropriate timeout values for kernel barriers is important for halting systems in a timely manner, and also it is hard since worst-case execution time (WCET) of the kernel and applications must be computed to obtain accurate timeout values. Therefore, the values are highly architecture- and application-specific. The computation of WCET is difficult for modern processors that extensively use caches, pipelines, speculative execution, and branch prediction. Various tools and research prototypes are surveyed by Wilhelm et al. [2008]. WCET analysis of seL4 is presented in [Blackham et al., 2011, 2012] on a TI DM3730 (ARM Cortex-A8) processor with L2 cache disabled. Following work further improves the analysis of kernel WCET [Sewell et al., 2016] as well as kernel WCET [Sewell et al., 2017]. Although the results cannot be directly used for processor models with L2 or L3 caches enabled, the work still provides guidelines and tools that can be used to estimate WCET of the seL4 kernel. However, the analysis is only applicable to LC-RCoE; setting and resuming instruction breakpoints in CC-RCoE render the WCET analysis highly dependable on which instruction is being monitored and if the monitored instruction is in a loop, when a replica is in the catching-up mode. Assuming that we have the following WCET results.

WCET_K represents the longest possible kernel execution path of the unmodified seL4 kernel (i.e., without support for redundant co-execution and error detection).

WCET_A represents the maximal execution time of user-mode instructions between any two successive system calls. To get this value, we need to have the applications that are planned to be deployed for analysis.

The WCET results can be used to calculate how long the leading replica needs to wait based on how many system calls the slowest replica still needs to execute.

8.2.7 Analysing and Hardening Shared Memory Regions

Several types of shared memory regions are described in the thesis:

(1) The kernel shared memory region for building the synchronisation protocol and kernel barriers (Section 4.3). The implementation uses several barriers to coordinate the replicas, and we use the timeouts of the barriers and logical checks to catch execution divergence caused by a data corruption in the region. For instance, a corrupted kernel barrier is mostly like

to cause a timeout of another barrier. Since this region is small, we can use information redundancy to protect the region if necessary.

(2) The kernel shared memory region for supporting device driver replication in CC-RCoE (Section 5.5) and I/O ports (Section 5.3). This region is used for storing temporary input data, which will be propagated to non-primary replicas. If the data in this region is corrupted, the replicas are likely to observe inconsistent input data and thus diverge. Subsequently, the corruption will be captured by barrier timeouts or fingerprint mismatches.

(3) The cross-replica shared memory region for supporting device driver replication in LC-RCoE (Section 5.4.1). Similarly, the region is used to support input data replication in user mode. Data corruptions in this region likely lead to similar results as in case (2).

The simple analysis above is incomplete, and we consider a complete analysis of all possible failures in these regions as future work. For (2) and (3), using redundant I/O devices and voting on input data from the redundant I/O devices can help to alleviate the issue.

8.2.8 Supporting Redundant I/O Devices for Device Driver Replication

The I/O access patterns (Section 5.2) are used to coordinate I/O operations for replicated device drivers when redundant I/O devices are not available. For x86 machines, supplementary I/O devices can be added to the expansion slots on motherboards. For example, it is trivial to integrate two or more network cards that are the same model to a single PC. Furthermore, the single root I/O virtualization and sharing specification [PCI-SIG, 2010] standardises I/O virtualization for PCIe-based devices, allowing efficient sharing a single I/O device by multiple virtual machines. A physical I/O device supporting the specification can provide multiple virtual I/O devices that can be directly assigned to virtual machines. Take network cards as an example; each VF (virtual function) is capable of transmitting/receiving data through DMA buffers and performing limited control operations (reset, configuring DMA descriptors and MAC addresses, etc.) [Int, 2011]. Therefore, it is worth investigating how redundant I/O devices can be used to improve performance and reliability of a system managed by RCoE. In the meanwhile, programmable NICs [Calvium, 2017; Mellanox, 2018; Netronome, 2018] provide on-board processing capability for network acceleration and offloads, reducing the pressure on general purpose CPUs. Potentially, such capability can be used to reduce the overhead of copying DMA buffers (Section 5.4.3 and Section 5.5.3) and thus to improve performance.

Bibliography

Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, pages 373–385, Berkeley, CA, USA, 2012.

J. R. Ahlbin, M. J. Gadlage, D. R. Ball, A. W. Witulski, B. L. Bhuva, R. A. Reed, G. Vizkelethy, and L. W. Massengill. The effect of layout topology on single-event transient pulse quenching in a 65 nm bulk CMOS process. *IEEE Transactions on Nuclear Science*, 57(6):3380–3385, December 2010.

Jim Alves-Foss, Paul W. Oman, Carol Taylor, and Scott Harrison. The MILS architecture for high-assurance embedded systems. *International Journal on Embedded Systems*, 2:239–247, 2006.

Hisashige Ando, Yuuji Yoshida, Aiichiro Inoue, Itsumi Sugiyama, Takeo Asakawa, Kuniki Morita, Toshiyuki Muta, Tsuyoshi Motokurumada, Seishi Okada, Hideo Yamashita, Yoshihiko Satsukawa, Akihiko Konmoto, Ryouichi Yamashita, and Hiroyuki Sugiyama. A 1.3 GHz fifth generation SPARC64 microprocessor. In *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.*, volume 1, pages 246–491, February 2003.

Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles. Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers*, 51(2):138–163, February 2002.

ARM. *ARM Cortex-R7 MPCore Technical Reference Manual*.

ARM. The ARM Cortex-A9 processors, 2009. URL https://web.archive.org/web/20120522214159/http://www.arm.com:80/files/pdf/ARMCortexA-9Processors.pdf.

ARM. ARM generic interrupt controller architecture specification. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.architecture/index.html, 2016.

Cortex-A9 Technical Reference Manual. ARM Ltd., r2p2 edition, 2010. ARM DDI 0388F.

ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. ARM Ltd., 2014. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html.

L. Artola, M. Gaillardin, G. Hubert, M. Raine, and P. Paillet. Modeling single event transients in advanced devices and ICs. *IEEE Transactions on Nuclear Science*, 62(4):1528–1539, August 2015.

Todd M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd ACM/IEE International Symposium on Microarchitecture*, pages 196–207, 1999.

M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 170–177, San Jose, CA, USA, 2003.

P. A. Barret, A. M. Hilborne, P. G. Bond, D. T. Seaton, P. Verissimo, L. Rodrigues, and N. A. Speirs. The Delta-4 extra performance architecture (XPA). In *[1990] Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*, pages 481–488, June 1990.

Robert Baumann. Technology scaling trends and accelerated testing for soft errors in commercial silicon devices. In *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.*, pages 4–, July 2003.

Robert Baumann. Soft errors in advanced computer systems. *IEEE Design and Test*, 22(3):258–266, May 2005a.

Robert Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, September 2005b.

Leonardo Bautista-Gomez, Ferad Zyulkyarov, Osman Unsal, and Simon McIntosh-Smith. Unprotected computing: A large-scale study of DRAM raw error rate on a supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, US, November 2016.

J. M. Benedetto, P. H. Eaton, D. G. Mavis, M. Gadlage, and T. Turflinger. Digital single event transient trends with technology node scaling. *IEEE Transactions on Nuclear Science*, 53(6): 3462–3465, December 2006.

Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 53–64, Pittsburgh, PA, USA, 2010.

Tom Bergan, Joseph Devietti, Nicholas Hunt, and Luis Ceze. The deterministic execution hammer: How well does it actually pound nails. In *The 2nd Workshop on Determinism and Correctness in Parallel Programming (WODET'11)*, 2011.

Richard Berger, David Artz, and Paul Kapcio. RAD750™ radiation hardened PowerPC™ microprocessor. October 2011.

Richard W. Berger, Devin Bayles, Ronald Brown, Scott Doyle, Abbas Kazemzadeh, Ken Knowles, David Moser, John Rodgers, Brian Saari, Dan Stanley, and Basil Grant. The RAD750™ - a radiation hardened PowerPC™ processor for high performance spaceborne applications. In *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*, volume 5, pages 2263–2272, March 2001.

David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. NonStop advanced architecture. In *Proceedings of the 35th International Conference on Dependable Systems and Networks (DSN)*, pages 12–21, Washington, DC, US, 2005.

Christian Bienia and Kai Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *IEEE Real-Time Systems Symposium*, pages 339–348, Vienna, Austria, November 2011.

Bernard Blackham, Yao Shi, and Gernot Heiser. Improving interrupt response time in a verifiable protected microkernel. In *EuroSys Conference*, pages 323–336, Bern, Switzerland, April 2012. doi: http://doi.acm.org/10.1145/2168836.2168869.

Bochs. The cross platform IA-32 emulator. `http://bochs.sourceforge.net/`, 2017.

Steve Bostian. Rachet up reliability for mission-critical applications. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/itanium-9500-reliability-mission-critical-applications-paper.pdf`, 2012.

Boundary Devices. BD-SL-i.MX6 development board. URL `https://boundarydevices.com/product/sabre-lite-imx6-sbc/`.

Boundary Devices. SABRE Lite hardware user manual, 2011. URL `https://boundarydevices.com/SABRE_Lite_Hardware_Manual_rev11.pdf`.

T. C. Bressoud. TFT: A software system for application-transparent fault tolerance. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, Munich, Germany, June 1998.

T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14:80–107, 1996.

D. Brière and P. Traverse. AIRBUS A320/A330/A340 electrical flight controls - A family of fault-tolerant systems. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pages 616–623, June 1993.

D. Brière, C. Favre, and P. Traverse. A family of fault-tolerant systems: electrical flight controls, from Airbus A320/330/340 to future military transport aircraft. *Microprocessors and Microsystems*, 19(2):75–82, 1995.

S. Buchner, M. Baze, D. Brown, D. McMorrow, and J. Melinger. Comparison of error rates in combinational and sequential logic. *IEEE Transactions on Nuclear Science*, 44(6):2209–2216, December 1997.

Buildroot. Buildroot, 2018. URL `https://buildroot.org`.

BusyBox. BusyBox: the swiss army knife of embedded linux. https://busybox.net/, 2017.

Calvium. Calvium LiquidIO®II network appliance smart NIC, 2017. URL https://www.marvell.com/documents/konmn48108xfxalr96jk/.

Ethan H. Cannon, Daniel D. Reinhardt, Michael S. Gordon, and Paul S. Makowenskyj. SRAM SER in 90, 130 and 180 nm bulk and SOI technologies. In *2004 IEEE International Reliability Physics Symposium. Proceedings*, pages 300–304, April 2004.

João Carreira, Henrique Madeira, and João Gabriel Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, February 1998.

Daniel Chen, Gabriela Jacques-Silva, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Bruce Mealey. Error behavior comparison of multiple computing systems: A case study using Linux on Pentium, Solaris on SPARC, and AIX on POWER. In *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 339–346, December 2008.

S. Chen, J. Xu, Z. Kalbarczyk, R. K. Iyer, and K. Whisnant. Modeling and evaluating the security threats of transient errors in firewall software. *Performance Evaluation*, 56(1–4):53–72, March 2004.

Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *ACM Symposium on Operating Systems Principles*, pages 73–88, Lake Louise, Alta, CA, October 2001.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing*, pages 143–154, Indianapolis, IN, US, June 2010.

Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 105–120, Monterey, CA, US, 2015.

Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, 2008.

Dell. Enabling memory reliability, availability, and serviceability features on Dell PowerEdge servers. http://www.dell.com/downloads/global/power/ps3q05-20050176-Patel-OE.pdf, 2016.

Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, 1966.

Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. PQEMU: A parallel system emulator based on QEMU. In *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 276–283, Tainan, Taiwan, 2011.

Anand Dixit and Alan Wood. The impact of new technology on soft error rates. In *2011 International Reliability Physics Symposium*, pages 5B.4.1–5B.4.7, April 2011.

Anand Dixit, Raymond Heald, and Alan Wood. Trends from ten years of soft error experimentation. In *Workshop on System Effects of Logic Soft Errors*, 2009. URL http://www.selse.org/images/selse_2009/Papers/selse5_submission_29.pdf.

Björn Döbel and Hermann Härtig. Who watches the watchmen? Protecting operating system reliability mechanisms. In *Proceedings of the 8th Workshop on Hot Topics in System Dependability*, Hollywood, CA, US, October 2012.

Björn Döbel and Hermann Härtig. Can we put concurrency back into redundant multithreading? In *Proceedings of the 14th International Conference on Embedded Software*, New Delhi, IN, October 2014.

Björn Döbel, Hermann Härtig, and Michael Engel. Operating system support for redundant multithreading. In *Proceedings of the 12th International Conference on Embedded Software*, pages 83–92, Tampere, SF, October 2012.

P. E. Dodd, M. R. Shaneyfelt, J. R. Schwank, and J. A. Felix. Current and future challenges in radiation effects on CMOS electronics. *IEEE Transactions on Nuclear Science*, 57(4):1747–1763, August 2010.

Paul E. Dodd and Lloyd W. Massengill. Basic mechanisms and modeling of single-event upset in digital microelectronics. *IEEE Transactions on Nuclear Science*, 50(3):583–602, June 2003.

Paul E. Dodd, Marty R. Shaneyfelt, James A. Felix, and James R. Schwank. Production and propagation of single-event transients in high-speed digital logic ICs. *IEEE Transactions on Nuclear Science*, 51(6):3278–3284, December 2004.

Bruce Dubbs. GNU GRUB. https://www.gnu.org/software/grub/.

Adam Dunkels. Minimal TCP/IP implementation with proxy support. Technical Report T2001-20, SICS, February 2001. http://www.sics.se/~adam/thesis.pdf.

Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. Kernel data – first class citizens of the system. In *Proceedings of the 2nd International Workshop on Object Systems and Software Architectures*, pages 39–43, Victor Harbor, South Australia, Australia, January 2006.

K. Elphinstone, A. Zarrabi, A. Danis, Y. Shen, and G. Heiser. An evaluation of coarse-grained locking for multicore microkernels. *ArXiv e-prints*, September 2016.

Kevin Elphinstone and Gernot Heiser. From L3 to seL4 – what have we learnt in 20 years of L4 microkernels? In *ACM Symposium on Operating Systems Principles*, pages 133–150, Farmington, PA, USA, November 2013.

Kevin Elphinstone and Yanyan Shen. Improving the trustworthiness of commodity hardware with software. In *Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology (DCDV)*, page 6, Budapest, Hungary, June 2013.

Kurt Engel. The advanced mission computer and display system for naval aviation. In *20th DASC. 20th Digital Avionics Systems Conference (Cat. No.01CH37219)*, volume 1, pages 1A3/1–1A3/9, October 2001.

S. Esposito, C. Albanese, M. Alderighi, F. Casini, L. Giganti, M. L. Esposti, C. Monteleone, and M. Violante. COTS-based high-performance computing for space applications. *IEEE Transactions on Nuclear Science*, 62(6):2687–2694, December 2015.

Yi-Pin Fang and Anthony S. Oates. Neutron-induced charge collection simulation of bulk FinFET SRAMs compared with conventional planar SRAMs. *IEEE Transactions on Devices and Materials Reliability*, 11(4):551–554, December 2011.

Véronique Ferlet-Cavrois, Lloyd W. Massengill, and Pascale Gouker. Single event transients in digital CMOS – a review. *IEEE Transactions on Nuclear Science*, 60(3):1767–1790, June 2013.

Davides Ferraretto and Graziano Pravadelli. Efficient fault injection in QEMU. In *2015 16th Latin-American Test Symposium (LATS)*, pages 1–6, March 2015.

Kathleen Fisher. Using formal methods to enable more secure vehicles: DARPA's HACMS program. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 1–1, Gothenburg, Sweden, 2014.

John G. Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, 30(1):247–252, January 1982.

FSF. GCC internals. https://gcc.gnu.org/onlinedocs/gccint/.

Fujitsu. Memory mirror and memory patrol for improved system reliability. http://www.fujitsu.com/global/products/computing/servers/unix/sparc-enterprise/technology/availability/memory.html, 2016.

Matthew J. Gadlage, Jonathan R. Ahlbin, Balaji Narasimham, Bharat L. Bhuva, Lloyd W. Massengill, Robert A. Reed, Ronald D. Schrimpf, and Gyorgy Vizkelethy. Scaling trends in SET pulse widths in sub-100 nm bulk CMOS processes. *IEEE Transactions on Nuclear Science*, 57 (6):3336–3341, December 2010.

Jacques S. Gansler and William Lucyshyn. Commercial-off-the-shelf COTS: Doing it right. Technical Report UMD-AM-08-129, University of Maryland,School of Public Policy,Center for Public Policy and Private Enterprise, College Park,MD,20742, September 2008. URL http://www.dtic.mil/dtic/tr/fulltext/u2/a494143.pdf.

Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM Symposium on Operating Systems Principles*, pages 193–206, Bolton Landing, NY, US, October 2003.

Harvey L. Garner. Error codes for arithmetic operations. *IEEE Transactions on Electronic Computers*, EC-15(5):763–770, October 1966.

Filipe de Aguiar Geissler, Fernanda Lima Kastensmidt, and José Eduardo Pereira Souza. Soft error injection methodology based on QEMU software platform. In *2014 15th Latin American Test Workshop - LATW*, pages 1–5, March 2014.

Ran Ginosar. Survey of processors for space. In *Proceedings of DASIA 2012, data systems in aerospace*, May 2012.

S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, pages 154–165, 2003.

Jim Gray. Why do computers stop and what can be done about it?, 1985. URL http://jimgray.azurewebsites.net/papers/tandemtr85.7_whydocomputersstop.pdf.

Anna Gregorio and Federico Alimenti. CubeSats for future science and internet of space: Challenges and opportunities. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 169–172, December 2018.

Kush Gulati, Lloyd W. Massengill, and Ghasi R. Agrawal. Single event mirroring and DRAM sense amplifier designs for improved single-event-upset performance. *IEEE Transactions on Nuclear Science*, 41(6):2026–2034, December 1994.

Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the 9th EuroSys Conference*, Amsterdam, NL, January 2014.

Peter Hazucha and Christer Svensson. Impact of CMOS technology scaling on the atmospheric neutron soft error rate. *IEEE Transactions on Nuclear Science*, 47(6):2586–2594, December 2000.

Daniel Henderson. POWER8 processor-based systems RAS. https://www.slideshare.net/thinkasg/ibm-power8-processorbased-systems-ras.

Daniel Henderson, Jim Mitchell, and George Ahrens. POWER7 system RAS. http://www-07.ibm.com/tw/imc/seminar/download/2010/POWER7_RAS_Whitepaper.pdf.

Jorrit N. Herder. *Building a Dependable Operating System: Fault Tolerance in MINIX 3*. PhD thesis, Department of Computer Science, Vrije Universiteit Amsterdam, September 2010.

Sebastian Hiergeist and Florian Holzapfel. Fault-tolerant FCC architecture for future UAV systems based on COTS SoC. In *ARCS 2016; 29th International Conference on Architecture of Computing Systems*, pages 1–5, April 2016.

Rhett M. Hollander and Paul V. Bolotoff. RAMspeed, a cache and memory benchmarking tool. http://alasir.com/software/ramspeed/.

Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. 1st edition, 2003.

A. L. Hopkins Jr., T. B. Smith III, and J. H. Lala. FTMP—a highly reliable fault-tolerant multiprocess for aircraft. *Proceedings of the IEEE*, 66(10):1221–1239, 1978.

HP. Memory technology evolution: An overview of system memory technologies. http://h10032.www1.hp.com/ctg/Manual/c00256987.pdf, 2016.

Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997.

IBM. Chipkill memory. http://www-05.ibm.com/hu/termekismertetok/xseries/dn/chipkill.pdf, 2016.

IEEE and The Open Group. The open group base specifications issue 7, 2018 edition, IEEE Std 1003.1-2017. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html.

Intel. Intel Xeon processor E7 family: Reliability, availability, and serviceability. http://www.intel.com.au/content/www/au/en/processors/xeon/xeon-e7-family-ras-server-paper.html, 2016a.

Intel. Intel software development emulator (Intel SDE). https://software.intel.com/en-us/articles/intel-software-development-emulator, 2016b.

PCI-SIG SR-IOV primer: An Introduction to SR-IOV Technology. Intel Corp., 2011. https://www.intel.com/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf.

Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. Intel Corp., 2016a. https://software.intel.com/en-us/articles/intel-sdm.

Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide. Intel Corp., 2016b. https://software.intel.com/en-us/articles/intel-sdm.

Srikanth Jagannathan, Matthew J. Gadlage, Bharat L. Bhuva, Ronald D. Schrimpf, Balaji Narasimham, Jugantor Chetia, Jonathan R. Ahlbin, and Lloyd W. Massengill. Independent measurement of SET pulse widths from N-hits and P-hits in 65-nm CMOS. *IEEE Transactions on Nuclear Science*, 57(6):3386–3391, December 2010.

Tahar Jarboui, Jean Arlat, Yves Crouzet, and Karama Kanoun. Experimental analysis of the errors induced into Linux by three fault injection techniques. In *Proceedings International Conference on Dependable Systems and Networks*, pages 331–336, May 2002.

JEDEC. Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices. http://www.jedec.org/sites/default/files/docs/jesd89a.pdf, 2016.

M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 237–250, Hollywood, CA, US, 2012.

Tanay Karnik, Peter Hazucha, and Jagdish Patel. Characterization of soft errors caused by single event upsets in CMOS processes. *IEEE Transactions on Dependable and Secure Computing*, 1 (2):128–143, April 2004.

Junsung Kim, Ragunathan (Raj) Rajkumar, and Markus Jochim. Towards dependable autonomous driving vehicles: A system-level approach. *SIGBED Rev.*, 10(1):29–32, February 2013.

Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceedings of the 41st International Symposium on Computer Architecture*, pages 361–372, June 2014.

M. P. King, R. A. Reed, R. A. Weller, M. H. Mendenhall, R. D. Schrimpf, B. D. Sierawski, A. L. Sternberg, B. Narasimham, J. K. Wang, E. Pitta, B. Bartz, D. Reed, C. Monzel, R. C. Baumann, X. Deng, J. A. Pellish, M. D. Berg, C. M. Seidleck, E. C. Auden, S. L. Weeden-Wright, N. J. Gaspard, C. X. Zhang, and D. M. Fleetwood. Electron-induced single-event upsets in static random access memory. *IEEE Transactions on Nuclear Science*, 60(6):4122–4129, December 2013.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009.

Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. San Francisco, CA, USA, 1st edition, 2007.

Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. HAFT: Hardware-assisted fault tolerance. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 25:1–25:17, 2016.

Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

Xin Li, Kai Shen, Michael C. Huang, and Lingkun Chu. A memory soft error measurement on production systems. In *Proceedings of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, US, 2007.

Jochen Liedtke. On μ-kernel construction. In *ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

Daniel Lipetz and Eric Schwarz. Self checking in current floating-point units. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 73–76, July 2011.

Tongping Liu, Charlie Curtsinger, and Emery D. Berger. DTHREADS: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 327–336, Cascais, Portugal, 2011.

LLVM. The LLVM compiler infrastructure. `http://llvm.org/`, 2016.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

J. A. Maharrey, R. C. Quinn, T. D. Loveless, J. S. Kauppila, S. Jagannathan, N. M. Atkinson, N. J. Gaspard, E. X. Zhang, M. L. Alles, B. L. Bhuva, W. T. Holman, and L. W. Massengill. Effect of device variants in 32 nm and 45 nm SOI on SET pulse distributions. *IEEE Transactions on Nuclear Science*, 60(6):4399–4404, December 2013.

N. N. Mahatme, S. Jagannathan, T. D. Loveless, L. W. Massengill, B. L. Bhuva, S. J. Wen, and R. Wong. Comparison of combinational and sequential error rates for a deep submicron process. *IEEE Transactions on Nuclear Science*, 58(6):2719–2725, December 2011.

Goerge Marsaglia. Random number generators. *Journal of Modern Applied Statistical Methods*, 2, 2003.

Lloyd W. Massengill. Cosmic and terrestrial single-event radiation effects in dynamic random access memories. *IEEE Transactions on Nuclear Science*, 43(2):576–593, April 1996.

Lloyd W. Massengill, Bharat L. Bhuva, W. Timothy Holman, Michael L. Alles, and T. Daniel Loveless. Technology scaling and soft error reliability. In *Proceedings of the 50th IEEE International Reliability Physics Symposium*, April 2012.

Timothy C. May and Murray. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, 1979.

Mellanox. Mellanox BlueField$^{\text{TM}}$ SmartNIC, 2018. URL `http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf`.

J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78–86, 1989.

Gokhan Memik, Masud H. Chowdhury, Arindam Mallik, and Yehea I. Ismail. Engineering overclocking: reliability-performance trade-offs for high-performance register files. In *Proceedings of the 35th International Conference on Dependable Systems and Networks (DSN)*, pages 770–779, June 2005.

Alan Messer, Philippe Bernadat, Guangrui Fu, Deqing Chen, Zoran Dimitrijevic, David Lie, Durga Devi Mannaru, Alma Riska, and Dejan Milojicic. Susceptibility of commodity systems and software to memory soft errors. *IEEE Transactions on Computers*, 53(12):1557–1568, December 2004.

Robert Meushaw and Donald Simard. NetTop commercial technology in high assurance applications, 2000. URL `http://cps-vo.org/node/6511`.

S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the number of fatal soft errors in Los Alamos national laboratory's ASC Q supercomputer. *IEEE Transactions on Devices and Materials Reliability*, 5(3):329–335, September 2005.

Todd K. Moon. *Error Correction Coding: Mathematical Methods and Algorithms*. 2005.

Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013. doi: 10.1109/SP.2013.35.

Hideyuki Nakamura, Taiki Uemura, Kan Takeuchi, Toshikazu Fukuda, Shigetaka Kumashiro, and Tohru Mogami. Scaling effect and circuit type dependence of neutron induced single event transient. In *2012 IEEE International Reliability Physics Symposium (IRPS)*, pages 3C.3.1–3C.3.7, April 2012.

Balaji Narasimham, Bharat L. Bhuva, Ronald D. Schrimpf, Lloyd W. Massengill, Matthew J. Gadlage, Oluwole A. Amusan, William Timothy Holman, Arthur F. Witulski, William H. Robinson, Jeffrey D. Black, Joseph M. Benedetto, and Paul H. Eaton. Characterization of digital single event transient pulse-widths in 130-nm and 90-nm CMOS technologies. *IEEE Transactions on Nuclear Science*, 54(6):2506–2511, December 2007.

NEC. Fault tolerant server white paper, Mar 2011. URL `http://www.nec.com/en/global/prod/express/collateral/whitepaper/ft_WhitePaper_E.pdf`.

Netronome. Agilio® FX 2x10GbE SmartNIC, 2018. URL `https://www.netronome.com/media/documents/PB_Agilio-FX.pdf`.

Michael Nicolaidis, Ricardo O. Duarte, Salvador Manich, and Joan Figueras. Fault-secure parity prediction arithmetic operators. *IEEE Design Test of Computers*, 14(2):60–71, April 1997.

Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proceedings of the 6th EuroSys Conference*, Salzburg, AT, April 2011.

i.MX 6Dual/6Quad Applications Processor Reference Manual. NXP, 2015. `http://www.nxp.com/assets/documents/data/en/reference-manuals/IMX6DQRM.pdf`.

Chip Errata for the i.MX 6Dual/6Quad and i.MX 6DualPlus/6QuadPlus, Rev. 6.1. NXP, 2016. `http://www.nxp.com/doc/IMX6DQCE`.

P. Oldiges, R. Dennard, D. Heidel, T. Ning, K. Rodbell, H. Tang, M. Gordon, and L. Wissel. Technologies to further reduce soft error susceptibility in SOI. In *2009 IEEE International Electron Devices Meeting (IEDM)*, pages 1–4, December 2009.

Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–108, Washington, DC, USA, 2009.

Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, pages 305–320, 2014.

Rich Painter. C converted Whetstone double precision benchmark. http://www.netlib.org/benchmark/whetstone.c, 1998.

Andrei Pavlov and Manoj Sanchdev. *CMOS SRAM Circuit Design and Parametric Test in Nano-Scaled Technologies: Process-Aware SRAM Design and Test*, chapter SRAM Circuit Design and Operation. 2008.

PCI-SIG. *Single Root I/O Virtualization and Sharing Specification Revision 1.1*, Jan 2010.

W Wesley Peterson and E J. Weldon. *Error-Correcting Codes, Second Edition*. Cambridge, MA, USA, 1972.

Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):413–421, 1974.

S. Potyra, V. Sieh, and M. Dal Cin. Evaluating fault-tolerant system designs using FAUmachine. In *Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems*, Dubrovnik, Croatia, 2007.

Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, February 2007.

RedisLabs. Redis, 2009. URL https://redis.io.

K. Reick, P. N. Sanda, S. Swaney, J. W. Kellington, M. Mack, M. Floyd, and D. Henderson. Fault-tolerant design of the IBM Power6 microprocessor. *IEEE Micro*, 28(2):30–38, March 2008.

Steven K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 25–36, June 2000.

George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd IEEE Symposium on Code Generation and Optimization*, pages 243–254, 2005.

Ronald Rivest. The MD5 message-digest algorithm. RFC 1654, April 1992. URL https://tools.ietf.org/html/rfc1321.

Kenneth P. Rodbell, David F. Heidel, Henry H. K. Tang, Michael S. Gordon, Phil Oldiges, and Conal E. Murray. Low-energy proton-induced single-event-upsets in 65 nm node, silicon-on-insulator, latches and memory cells. *IEEE Transactions on Nuclear Science*, 54(6):2474–2479, December 2007.

Rowhammer. Row hammer. http://en.wikipedia.org/wiki/Row_hammer.

John Rushby. Design and verification of secure systems. In *ACM Symposium on Operating Systems Principles*, pages 12–21, Pacific Grove, CA, USA, December 1981.

G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer. An experimental study of soft errors in microprocessors. *IEEE Micro*, 25(6):30–39, November 2005.

P. N. Sanda, J. W. Kellington, P. Kudva, R. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. R. Jones. Soft-error resilience of the IBM POWER6 processor. *IBM Journal of Research and Development*, 52(3), May 2008.

D. J. Scales, M. Nelson, and G. Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *ACM Operating Systems Review*, 44(4):30–39, December 2010.

Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *2015 11th European Dependable Computing Conference (EDCC)*, pages 245–255, September 2015.

Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A large-scale field study. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems*, pages 193–204, Seattle, WA, US, 2009.

Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. https://googleprojectzero.blogspot.com.au/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.

N. Seifert, P. Slankard, M. Kirsch, B. Narasimham, V. Zia, C. Brookreson, A. Vo, S. Mitra, B. Gill, and J. Maiz. Radiation-induced soft error rates of advanced CMOS bulk devices. In *Proceedings of the 44th IEEE International Reliability Physics Symposium*, pages 217–225, San Jose, CA, US, March 2006.

seL4. seL4 reference manual with asynchronous endpiont binding extensions API version 1.3. http://www.cse.unsw.edu.au/~cs9242/16/project/sel4-manual.pdf.

seL4. seL4 synchronisation library. https://github.com/seL4/seL4_libs/tree/master/libsel4sync, 2014.

Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In *International Conference on Interactive Theorem Proving*, pages 325–340, Nijmegen, The Netherlands, August 2011. doi: 10.1007/978-3-642-22863-6_24.

Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 471–481, Seattle, Washington, USA, June 2013.

Thomas Sewell, Felix Kam, and Gernot Heiser. Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Vienna, Austria, April 2016.

Thomas Sewell, Felix Kam, and Gernot Heiser. High-assurance timing analysis for a high-assurance real-time OS. *Real-Time Systems*, 53:812–853, September 2017. doi: 10.1007/s11241-017-9286-3.

Tom Shanley and Don Anderson. *PCI System Architecture*. 1999.

S. Z. Shazli, M. Abdul-Aziz, M. B. Tahoori, and D. R. Kaeli. A field analysis of system-level effects of soft errors occurring in microprocessors used in information systems. In *2008 IEEE International Test Conference*, pages 1–10, October 2008.

Yanyan Shen and Kevin Elphinstone. Microkernel mechanisms for improving the trustworthiness of commodity hardware. In *European Dependable Computing Conference*, page 12, Paris, France, September 2015.

Yanyan Shen, Gernot Heiser, and Kevin Elphinstone. Fault tolerance through redundant execution on COTS multicores: Performance, functionality, and dependability trade-offs. In *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN)*, June 2019.

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 32nd International Conference on Dependable Systems and Networks (DSN)*, Bethesda, MD, US, June 2002.

Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A. Connors. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(2):135–148, April 2009. doi: 10.1109/TDSC.2008.62.

Brian D. Sierawski, Marcus H. Mendenhall, Robert A. Reed, Michael A. Clemens, Robert A. Weller, Ronald D. Schrimpf, Ewart W. Blackmore, Michael Trinczek, Bassam Hitti, Jonathan A. Pellish, Robert C. Baumann, Shi-Jie Wen, Rick Wong, and Nelson Tam. Muon-induced single event upsets in deep-submicron technology. *IEEE Transactions on Nuclear Science*, 57(6): 3273–3278, December 2010.

Charles Slayman. Soft error trends and mitigation techniques in memory devices. In *2011 Proceedings - Annual Reliability and Maintainability Symposium*, pages 1–5, January 2011.

T.J. Slegel, III Averill, R.M., M.A Check, B.C. Giamei, B.W. Krumm, C.A Krygowski, W.H. Li, J.S. Liptay, J.D. MacDougall, T.J. McPherson, J.A Navarro, E.M. Schwarz, K. Shum, and C.F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, March 1999. doi: 10.1109/40.755464.

J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, pages 250–259, June 1996.

Baruch Solomon, Avi Mendelson, Ronny Ronen, Doron Orenstien, and Yoav Almog. Micro-operation cache: A power aware frontend for variable instruction length ISA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(5):801–811, October 2003.

L. Spainhower and T. A. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 43(5):863–873, September 1999.

Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. Feng shui of supercomputer memory: Positional effects in DRAM and SRAM faults. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Denver, CO, US, 2013.

Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–310, Istanbul, Turkey, 2015.

Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. 2005.

Armen Toorian, Ken Diaz, and Simon Lee. The CubeSat approach to space access. In *2008 IEEE Aerospace Conference*, pages 1–14, March 2008.

UBoot. The universal boot loader. http://www.denx.de/wiki/U-Boot.

Paulo E. Veríssimo, Nuno F. Neves, Christian Cachin, Jonathan Poritz, David Powell, Yves Deswarte, Robert Stroud, and Ian Welch. Intrusion-tolerant middleware: the road to automatic security. *IEEE Security Privacy*, 4(4):54–62, July 2006.

VIRTIO-v1.0. Virtual I/O device (VIRTIO) version 1.0 03 december 2013. committee specification draft 01/public review draft 01. http://docs.oasis-open.org/virtio/virtio/v1.0/csprd01/virtio-v1.0-csprd01.html, 2013.

VMware. What's New in the VMware vSphere 6.0 Platform. http://www.vmware.com/files/pdf/vsphere/VMW-WP-vSPHR-Whats-New-6-0-PLTFRM.pdf, 2015.

C. Wang, H. S. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proceedings of the 5th International Symposium on Code Generation and Optimization*, pages 244–258, 2007.

David Tawei Wang. *Modern DRAM Memory Systems: Performance Analysis and Scheduling Algorithm*. PhD thesis, College Park, MD, USA, 2005.

Feng Wang, Yuan Xie, Kerry Bernstein, and Yan Luo. Dependability analysis of nano-scale FinFET circuits. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, March 2006.

Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. COREMU: A scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 213–222, San Antonio, TX, USA, 2011.

Christopher Weaver, Joel Emer, Shubhendu S. Mukherjee, and Steven K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 264–, 2004.

V. Weaver, D. Terpstra, and S. Moore. Non-determinism and overcount on modern hardware performance counter implementations. April 2013.

Reinhold P. Weicker. Dhrystone benchmark: Rationale for version 2 and measurement rules. *SIGPLAN Notices*, 23(8):49–62, August 1988.

Reinhold P. Weicker. A detailed look at some popular benchmarks. *Parallel Computing*, 17(10): 1153–1172, 1991. Benchmarking of high performance supercomputers.

J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, 1978.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, S. Margherita Ligure, IT, 1995.

J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer. An experimental study of security vulnerabilities caused by errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 421–430, 2001.

Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 122–135, San Diego, CA, US, 2003.

Y. C. Yeh. Triple-triple redundant 777 primary flight computer. In *1996 IEEE Aerospace Applications Conference. Proceedings*, volume 1, pages 293–307, February 1996.

Haissam Ziade, Rafic Ayoubi, and Raoul Velazco. A survey on fault injection techniques. *International Arab Journal of Information Technology*, 1(2):171–186, July 2004. URL https://hal.archives-ouvertes.fr/hal-00105562.

J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420): 776–788, 1979. URL http://science.sciencemag.org/content/206/4420/776.

Appendix A

Terminology

- Transient Hardware Faults: anomalies that change hardware state but do not cause damage permanently. The affected state can be corrected by rewriting correct values. Bit flips induced by alpha particles or neutrons are examples of transient faults.
- Intermittent Hardware Faults: anomalies that appear and disappear multiple times without a clear pattern.
- Permanent Hardware Faults: anomalies that cause permanent damage to hardware and remain indefinitely. The impaired component needs treatment to restore its functions. Oxide wearout is an example of a permanent fault.
- Singe Event Upset (SEU): a non-destructive change of state in a storage element. An SEU is caused by ionizing particle striking a sensitive node of the device, and it can manifest as a single-bit upset or a multiple-bit upset.
- Multiple Cell Upset (MCU): a single event upset that causes several bits in an integrated circuitry to fail.
- Multiple Bit Upset (MBU): a multiple-cell upset (MCU) in which two or more error bits occur in the same word.
- Soft Errors: incorrect data or signals that caused by transient hardware faults and can be observed through architectural interfaces.
- Failure In Time (FIT): the number of errors occurred in one billion hours.
- Soft Error Rate (SER): the observed or predicated frequency of a device experiencing soft errors, usually measured in FIT.
- Silent Data Corruption (SDC): erroneous outputs produced by a system that is affected by a fault. Usually, systems not equipped with error detection mechanisms or using only simple mechanisms are vulnerable to hardware faults.
- Detected Unrecoverable Error (DUE): errors that can be detected by an error detection mechanism but cannot be corrected by the mechanism. For example, SEC-DED ECC can detect a double-bit error but cannot correct the error.

- Error-Correcting Code (ECC): coding schemes that embed redundant information to the original data in order to correct errors in the coded data.

- Single-Error-Correction Double-Error-Detection ECC (SEC-DED ECC): a coding scheme that is able to correct single-bit errors and detect double-bit errors.

- Fault Detection Latency: the delay between the occurrence of a fault and the detection of the fault.

- Dual Modular Redundancy (DMR): an engineering approach that duplicates the components of a system twice for spatial redundancy. The duplicated components work in parallel, and an error can be identified by comparing outputs.

- Triple Modular Redundancy (TMR): an engineering approach that duplicates the components of a system three times for spatial redundancy. An error can be detected and corrected since the faulty component can be identified by majority voting.

- Replica (node): the basic unit for replication, redundant execution, and comparison. A DMR system has two replicas, and a TMR system has three replicas.

- Error Coverage: the percentage of errors that can be detected by an error detection mechanism.

- Microkernel: a kernel that is designed to provide only minimal sets of mechanisms to build an operating system. Compared with monolithic kernels which include almost all system services (e.g., file systems, device drivers, memory manager, and network stack) in privileged mode, a microkernel implements such services as user-mode processes, reducing the trusted computing base.

- Hypervisor: a piece of system software that enables creating, scheduling, servicing, and isolating multiple virtual machines (VMs) on a single physical host machine. Each VM executes its operating system on hardware resources allocated by a hypervisor.

- Barrier (synchronisation): a synchronisation primitive that allows a group of threads or processes to pass the barrier only when all the members of the group arrive at it.

- Memory Barrier (hardware): architecture-specific instructions that prevent hardware from reordering memory accesses specified in program order and ensure the observation of the effects of memory accesses. Architectures implementing weak memory models may reorder memory accesses to hide memory access latency.

- MMU: memory management unit.

Appendix B

The SPIN Model for the Synchronisation Protocol

```
/* The number of replicas */
#define NKERNEL       2

#ifndef NKERNEL
#define NKERNEL       2
#endif


/* Deterministic events. These events are handled by each *
 * replica locally without triggering synchronisations.   *
 * The original seL4 system calls are such events.        */
#define DE_NORMAL        0
/* Deterministic synchronous events. These events need    *
 * to trigger synchronisations. For instance, the         *
 * fingerprint comparisons invoked by seL4_FT_Add_Event   *
 * system call are such events.                           */
#define DE_SYNC          1

#define DE_TYPE_BITS     1

/* Nondeterministic events (I/O device interrupts). We     *
 * need to synchronise the replicas before allowing them   *
 * to observe these events.                                */
#define ND_EVT      2

#define N_BITS    2

#if NKERNEL == 2
#define DE_LIMIT        7
#define DE_LIMIT_BITS   3
#endif
```

```
#if NKERNEL == 3
#define DE_LIMIT        3
#define DE_LIMIT_BITS   2
#endif

typedef de_t {
    unsigned type : DE_TYPE_BITS;
};

/* The array of deterministic events to be consumed */
de_t des[DE_LIMIT];

/* The per-replica data structure for building the protocol */
typedef data_t {
    bool        catchup;
    bool        entry_de;
    bool        entry_nde;
    bool        sync;
    bool        sync_req;
    bool        out_flag;
    unsigned    de_type     : DE_TYPE_BITS;
    unsigned    de_count    : DE_LIMIT_BITS;
    unsigned    sync_abort  : N_BITS;
    unsigned    sync_abort_notify : N_BITS;
    unsigned    lead_rep   : N_BITS;
};

data_t data[NKERNEL];

/* The data structure for kernel barriers */
typedef bar_t {
    unsigned  cnt : N_BITS;
    bool      gflag;
    bool      flag[NKERNEL];
    bool      backoff[NKERNEL];
};

/* sbar represents the kernel S-barrier used by the protocol */
bar_t sbar;

/* fbar represents the kernel F-barrirs used by the newly  *
 * added system calls (seL4_FT_Add_Event, seL4_FT_Mem_Rep) *
 * and the kernel barriers used for coordinating kernel    *
 * accesses to I/O devices.                                */
bar_t fbar;

unsigned de_type : DE_TYPE_BITS;
unsigned de_index : DE_LIMIT_BITS;

/* Generate random deterministic events */
inline gen_des() {
    atomic {
        for (de_index : 0 .. (DE_LIMIT - 1)) {
            select (de_type : DE_NORMAL .. DE_SYNC);
            des[de_index].type = de_type;
        }
    }
}

/* Get the next deterministic event */
inline get_de(kid) {
    atomic {
        data[kid].de_type = des[data[kid].de_count].type;
    }
}

/* The common part of the barriers */
inline bar_common(kid, b) {
    b.flag[kid] = b.gflag;
/* The atomic steps are protected by a spin lock or  *
 * implemented with atomic instructions              */
    atomic {
        b.cnt++;
        if
        :: (b.cnt == NKERNEL) ->  {
            b.cnt = 0;
            b.gflag = 1 - b.gflag;
        }
        :: else -> skip;
        fi;
    }
}

inline bar(kid, b) {
    bar_common(kid, b);
    (b.flag[kid] != b.gflag);
}

/* The conditional kernel barrier. */
inline bar_cond(kid, b) {
    bar_common(kid, b);
    if
    :: (b.flag[kid] != b.gflag) -> {
        skip;
    }
```

```
/* When the condition below becomes true, the conditinal   *
 * barrier aborts; and the replicas waiting on the barrier *
 * proceed.                                                 */
    :: (data[kid].sync_abort != data[kid].sync_abort_notify) -> {
        assert(b.flag[kid] == b.gflag);
        b.backoff[kid] = 1;
        b.cnt = 0;
    }
    fi
}

/* This function triggers synchronisations by marking the   *
 * sync and sync_req as true. In the C implementation, the  *
 * function sends IPIs to other replicas to notify that     *
 * a round of synchronisation is required.                  */
inline trigger_action(kid) {
    for (_for_i : 0 .. (NKERNEL - 1)) {
        data[_for_i].sync = true;
        data[_for_i].sync_req = true;
    }
}

inline do_sync(kid) {
   if
   :: data[kid].de_count == data[data[kid].lead_rep].de_count -> {
/* The one with the highest event counter needs to wait others *
 * to catch up.                                                 */
       data[kid].catchup = 0;
       bar(kid, sbar);

/* After passing the second barrier, all kernel instances      *
 * finished the same number of deterministic events, so they   *
 * can proceed to handle some actions.                          */
       assert(data[kid].de_count <= DE_LIMIT);
       assert(data[kid].de_count ==
          data[data[kid].lead_rep].de_count);
/* The barrier is required for the assertions above */
       bar(kid, sbar);

       /* Reset states */
       data[kid].catchup = 0;
       data[kid].lead_rep = 0;
       data[kid].sync = 0;
       data[kid].out_flag = false;
       bar(kid, sbar);
   }
   :: else -> {
/* The replicas need to catch up until their de_count variables *
```

```
 * are the same as the leading replica's de_count. The catchup   *
 * is set to true to indicate that a replica is in catching-up   *
 * mode.                                                        */
       assert(data[kid].de_count <
          data[data[kid].lead_rep].de_count);
       data[kid].catchup = true;
   }
   fi
}

/* The inline synchronises the replicas when sync == true. */
inline handle_action(kid) {
    if
    :: data[kid].sync == true -> {
        if
        :: data[kid].catchup == true -> {
            do_sync(kid);
        }
        :: else -> {
            data[kid].sync_abort = data[kid].sync_abort_notify;
            /* Check if other replicas are ahead */
            for (_for_i : 0 .. (NKERNEL - 1)) {
                if
                :: data[kid].de_count < data[_for_i].de_count -> {
                    data[kid].out_flag = true;
                }
                :: else -> skip;
                fi
            }
            if
            :: data[kid].out_flag == false -> {
                /* Wait on the conditional barrier */
                bar_cond(kid, sbar);
                if
                :: sbar.backoff[kid] == true -> {
                    sbar.backoff[kid] = false;
                }
                :: else -> {
    /* All the replicas arrive here, so we start to vote the *
     * leading replica.                                      */
                    for (_for_i : 0 .. (NKERNEL - 1)) {
                        if
                        :: data[_for_i].de_count >
                           data[data[kid].lead_rep].de_count -> {
                            data[kid].lead_rep = _for_i;
                        }
                        :: else -> skip;
                        fi
                    }
   /* The leading replica has been voted, and we start to *
    * synchronise the replicas.                           */
                    do_sync(kid);
                }
                fi
            }
            :: else -> { data[kid].out_flag = false;}
            fi
        }
        fi
    }
    :: else -> skip;
    fi
}

/* Notify the replicas waiting on the conditional barrier *
 *  to exit.                                              */
inline bar_sync_abort_notify(kid) {
    for (_for_i: 0 .. (NKERNEL - 1)) {
        data[_for_i].sync_abort_notify++;
    }
}


/* This function is used to represent several kernel barriers     *
 * used by synchronous events.  It aborts the conditional barrier *
 * if required so that the replicas waiting on the conditional    *
 * barrier can proceed and arrive at this barrier, avoiding       *
 * deadlocks.                                                     */
inline sync_event_bar(kid) {
    trigger_action(kid);
    bar_sync_abort_notify(kid);
    bar(kid, fbar);
    for (_for_i : 0 .. (NKERNEL - 1)) {
        assert(data[_for_i].de_count == data[kid].de_count);
        skip;
    }
    bar(kid, fbar);
}

inline de_handler(kid) {
    if
    :: data[kid].de_type == DE_NORMAL -> {
        /* Do nothing for normal syscalls; just count them */
        data[kid].de_count++;
    }
        /* Sync events need to synchronise the replicas */
    :: data[kid].de_type == DE_SYNC-> {
        data[kid].de_count++;
        sync_event_bar(kid);
    }
    fi

/* Reset counters and generate the next batch of deterministic *
 * events if necessary. This is because we only have limited   *
 * bits for the de_count to avoid state explosion.             */
    if
    :: data[kid].de_count == DE_LIMIT ->  {
        bar_sync_abort_notify(kid);
        bar(kid, fbar);
        data[kid].de_count = 0;
        if
        :: kid == 0 -> gen_des();
        :: else -> skip;
        fi
        bar(kid, fbar);
    }
    :: else -> skip;
    fi

    /* Check if any actions pending */
    handle_action(kid);
}

inline nd_handler(kid) {
    assert(kid == 0);
    trigger_action(kid);
    handle_action(kid);
}

/* The primary replica */
inline kernel_pri(kid) {
    unsigned _for_i : N_BITS;
    assert(kid == 0);
/* Nondeterministic choices between deterministic events and *
 * nondeterministic events.                                  */
    do
    :: data[kid].entry_de = true -> {
/* Get the type of the current deterministic event. DE_NORMAL *
 * or DE_SYNC.                                                */
        get_de(kid);
        de_handler(kid);
        data[kid].entry_de = false;
    }
    /* Nondeterministic interrupts */
    :: data[kid].entry_nde = true -> {
        nd_handler(kid);
        data[kid].entry_nde = false;
    }
    od
}

/* The non-primary replica(s) */
proctype kernel_np(unsigned kid : N_BITS) {
    unsigned _for_i : N_BITS;
    assert(kid != 0);
    do
/* The non-primary replicas only observe deterministic    *
 * events when the sync_req is false. When the sync_req   *
 * is true, it represents that a round of synchronisation *
 * is triggered by the primary replica in order to handle *
 * nondeterministic interrupts.                           */
    :: data[kid].entry_de = true ->
    {
        get_de(kid);
        de_handler(kid);
        data[kid].entry_de = 0;
    }
    /* Sync notification IPI */
    :: data[kid].sync_req == true -> {
        handle_action(kid);
        data[kid].sync_req = false;
    }
    od
}

/* The execution starts here */
init {
    gen_des();
    /* Bring up other replicas */
    run kernel_np(1);
#if NKERNEL == 3
    run kernel_np(2);
#endif
    /* The init proc becomes the primary replica */
    kernel_pri(0);
}
```

Listing B.1: The SPIN model for the synchronisation protocol

Appendix C

Background

C.1 DRAM

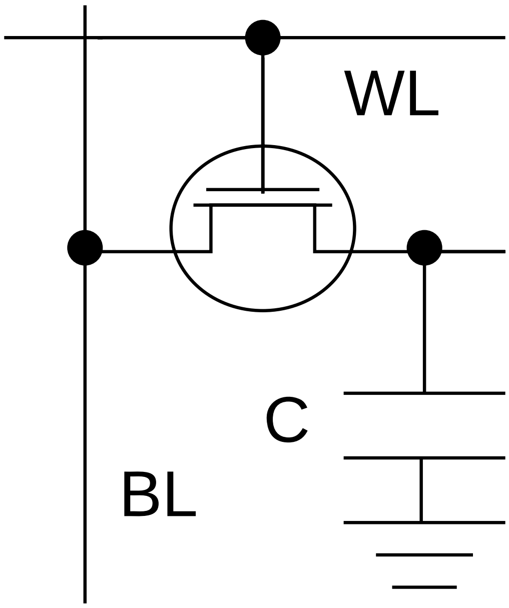

Figure C.1: 1-T-1-C DRAM cell

Figure C.1 is the logical circuit diagram for a modern DRAM cell; the cell is made of one transistor and one capacitor, representing one bit [Wang, 2005]. The *WL* (word line) controls accesses to the capacitor; and the *BL* (bit lines), which actually consists two lines, are used to read out the data or to overwrite the cell with new data. One of the bit lines directly connects to the transistor, and the other serves as a reference voltage for read operations. When the capacitor is charged to a specified voltage level, the cell stands for a "1"; a "0" is stored when the capacitor is discharged to 0 voltage. Due to the leakage effect, the capacitor must be refreshed periodically to retain the correct data. For a read operation, the two bit lines are pre-charged to the median value of logic "0" and "1". The lines still maintain the voltage for a short period after the pre-charge circuit turns off. After that, the access transistor is turned on. If the capacitor contains a "1", it discharges and transfers the charge to the bit line connected to the transistor. Thus, the voltage of the bit line is slightly higher than the other line served as the reference. If a "0" is stored, the connected bit line transfers charge to the capacitor so that its voltage decreases slightly. The voltage difference between the bit lines is picked up by the *sense amplifier* and interpreted as a "1" or "0". Obviously, a read operation destroys the value stored in a cell; so the read value is written back to the cell again. For a write operation, the bit line is set to high or low voltage according to the value, and the corresponding WL is selected, forcing the capacitor to charge or discharge.

C.2 Static Random Access Memory (SRAM)

The logic layout of the widely used 6-transistor SRAM cell is illustrated in Figure C.2 [Pavlov and Sanchdev, 2008]. The data is stored in a pair of cross-coupled inverters (N1 and P1, N2 and P2); so as long as the power is on, the data is retained by the inverters. Transistors N3 and N4 control access to the inverters, and they are enabled/disabled by the WL (word line). Assuming that both N3 and N4 are off, a "1" is stored when transistors N1 and P2 are off and transistors N2 and P1 are on; a "0" is stored in the opposite case.

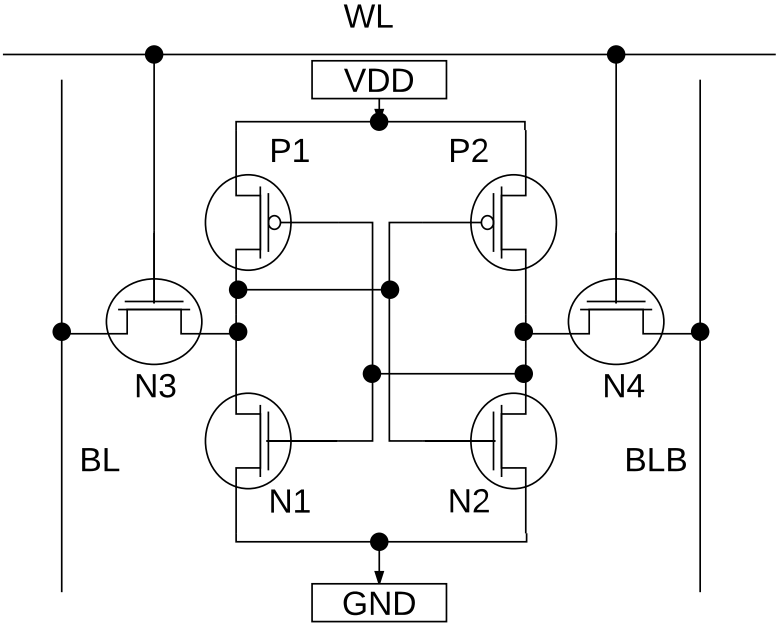

Figure C.2: 6-T SRAM cell

We briefly show how a read request is fulfilled and assume that the cell contains a "1". Firstly, both bit lines BL and BLB are pre-charged to an intermediate voltage $V_{dd}/2$. Then, N3 and N4 are turned on by applying V_{dd} to the word line. Because N4 and N2 are on, the bit line BLB is connected to the ground and starts to discharge. Also, the bit line BL is connected to V_{dd} and being charged towards V_{dd} since P1 and N3 are on and N1 is off. The differential voltage bewteen the bit lines BL and BLB is picked up and amplified to the voltage of logic "1" by the sense amplifier (not shown in the figure) which speeds up the read process. To write a value, the bit lines BL and BLB are set to the corresponding voltages for the value, and then, the word line WL is enabled to overwrite the inverters with the new value. Assuming that the previous value is 0 (N1 is on; N2 is off), writing a "1" requires pulling the bit line BL up to V_{dd} and pulling the bit line BLB down to 0. When the WL is enabled, the transistor N1 is turned off since the bit line BLB is 0 and the transistor N4 is on. N2 is connected to the bit line BL (V_{dd}) through N3 and is switch on. The transistors N1 and N2 flip their states, so do the transistors P1 and P2.

Now, let us consider how a transient fault changes the value stored in a SRAM cell. The sensitive nodes are the drain of the OFF-NMOS transistor and the drain of the OFF-PMOS transistor. Supposing that the original value is "0" (N2 is off; N1 is on) and that the transistor N2 is impacted by a transient fault, the transient current passes through the stuck transistor N2; and the transistor

P2 tries to counteract the effect so that there is also current flowing through P2. The end result is a voltage drop at the drain of P2 [Dodd and Massengill, 2003], changing the status of the inverter on the left (N1 → off; P1 → on). The output for the left inverter becomes 1 which is fed back to the right inverter, reinforcing the wrong value by changing the transistor N2 to on and the transistor P2 to off. Thus, both inverters flip their states so that the cell now represents an incorrect value "1".

Appendix D

Implementation Details

D.1 The Conditional Barrier

A simple implementation of the conditional barrier is shown in Listing D.1 and Listing D.2 for the DMR mode. The condition to abort blocking on the barrier is that the local variable sync_abort read at line 2 is different from the current value of ft_data[node_id].sync_abort. For the barriers used in other parts of the kernel (e.g., the barrier used by seL4_FT_Mem_Rep), the kernel does the following two steps before blocking itself on the barriers: (1) increment the per-replica event counting variable (et_count) and (2) increase the sync_abort variables of all other replicas.

```
1  /* save a local copy of the per-replica sync_abort */
2  int sync_abort = ft_data[cur_node_id].sync_abort;
3  /* dmb: a memory barrier is needed for ARM */
4  dmb();
5  /* if the other replica is ahead, we just return to catch up */
6  if (ft_data[1 - cur_node_id].et_count >
       ft_data[cur_node_id].et_count)
7    return;
8  /* the conditional barrier checks the sync_abort variable */
9  while (required_number_not_achieved) {
10   if (sync_abort != ft_data[cur_node_id].sync_abort) {
11     abort_cond_barrier();
12     return;
13   }
14 }
```

Listing D.1: Conditional Barrier (waiting side)

```
1  /* increae the event counter */
2  ft_data[cur_node_id].et_count++;
3  /* dmb: a memory barrier is needed for ARM */
```

```
4 dmb();
5 ft_data[1 - cur_node_id].sync_abort++;
6 /* now we can block on another barrier */
7 kbar_wait(&another_bar);
```

Listing D.2: Conditional Barrier (aborting side)

The key mechanism is that if a replica is waiting on another barrier ahead, either the et_count comparison at line 6 or the sync_abort comparison at line 10 will cause the synchronisation initiator to abort the current round, catch up, and meet the other replica at the barrier ahead. On ARM, we need the memory barriers (dmb()) to avoid reordering of the two reads at lines 2 and 6 in Listing D.1, and two writes at lines 2 and 5 in Listing D.2. Keeping the orders is vital to the correctness of the algorithm. Now let us consider all the possible interleaving of the two reads and two writes, as shown in listings D.3 to D.8, assuming the replica 0 waits on the conditional barrier and the replica 1 is waits on another barrier.. The *r* and *w* represent read and write accesses, and the *0* and *1* indicate the replica that issued the read/write. For the interleaving A, the conditional barrier is aborted by the write to the ft_data[0].abort_sync by the replica 1. For the interleavings B to F, the et_count comparison cancels the current round of synchronisation since the et_count of the replica 1 is increased before being read by the replica 0. Should the reads or writes be reordered by the hardware, the algorithm fails to fulfil its purpose.

```
/* aborting cond_barrier */
r0 ft_data[0].abort_sync;
r0 ft_data[1].et_count;
w1 ft_data[1].et_count;
w1 ft_data[0].abort_sync;
```

Listing D.3: Interleaving A

```
/* et_count comparison */
r0 ft_data[0].abort_sync;
w1 ft_data[1].et_count;
r0 ft_data[1].et_count;
w1 ft_data[0].abort_sync;
```

Listing D.4: Interleaving B

```
/* et comparison */
r0 ft_data[0].abort_sync;
w1 ft_data[1].et_count;
w1 ft_data[0].abort_sync;
r0 ft_data[1].et_count;
```

Listing D.5: Interleaving C

```
/* et_count comparison */
w1 ft_data[1].et_count;
w1 ft_data[0].abort_sync;
r0 ft_data[0].abort_sync;
r0 ft_data[1].et_count;
```

Listing D.6: Interleaving D

```
/* et_count comparison */
w1 ft_data[1].et_count;
r0 ft_data[0].abort_sync;
w1 ft_data[0].abort_sync;
r0 ft_data[1].et_count;
```

Listing D.7: Interleaving E

```
/* et_count comparison */
w1 ft_data[1].et_count;
r0 ft_data[0].abort_sync;
r0 ft_data[1].et_count;
w1 ft_data[0].abort_sync;
```

Listing D.8: Interleaving F

D.2 Sending Notifications

If the primary replica needs to start a round of synchronisation, it sets the `sync_evt` variable of each replica to indicate the purpose (handling interrupts, rescheduling, and/or fingerprint comparison) of the next synchronisation and then notifies all other replicas by sending IPIs (inter-processor interrupts). On ARM, the IPIs are generated by writing to the software generated interrupt register of GICD (generic interrupt controller distributor) [ARM, 2016] with target core ID and interrupt ID. On x86, interrupt command registers of LAPIC (local advanced programmable interrupt controller) [Int, 2016b] are programmed with corresponding core APIC ID and interrupt number. Although the `sync_evt` can be observed by the replicas when they handle deterministic events (each replica checks the variable before it returns to user mode from kernel mode), we need the IPIs to reduce interrupt latency and deal with "unfriendly" programs (e.g., programs use `while (true)` loops for polling).

D.3 Implementation Details of Hardware-Assisted CC-RCoE for x86

The width of the performance counters for Haswell and Skylake microarchitectures is 48-bit, and each core has 8 general purpose counters that (`IA32_PMCx`) can be programmed to count certain events by writing to their associated control registers (`IA32_PERFEVTSELx`). The values of the `IA32_PMCx` can be read by the `rdpmc` instruction, and the `IA32_PERFEVTSELx` are accessed by the `rdmsr` instruction. We reserve two counters and instruct them to include only user-mode events, so it is safe to read the counter values in kernel mode with two `rdpmc` instructions since the counters stop counting in kernel mode.

The x86 architecture has 8 debug registers (`DR0` to `DR7`). However, the `DR4` and `DR5` are obsolete synonyms for `DR6` and `DR7`. `DR0` to `DR3` contain the linear addresses we want to monitor, so we can monitor up to 4 addresses at a time. Being the control register, `DR7` specifies the trigger conditions (instruction execution, read, read and write, or I/O read and write) and the levels (local or global) for the 4 debug address registers. The generation of a debug exception, when the condition for a debug address register is met, is also controlled by `DR7`. `DR6` is the debug status register, indicating which debug address register triggers a debug exception. The CPU never clears the status register, so the software must reset it after handling a debug exception to avoid confusions.

We reserve one debug address register, `DR0`, for the precise preemption algorithm. When we need to set an instruction breakpoint, DR0 is updated with the leading replica's IP; DR7 is also changed to specify that an exception must be triggered right before executing the instruction pointed by the address in DR0 and that the address must be matched globally (i.e., all threads in a system). The global matching is required: When we vote the leading replica, the current running threads of the system replicas can be different (note that IPCs do context switches frequently and that we do not invoke the synchronisation protocol for such deterministic events).

When a chasing kernel replica receives a debug exception, it consults DR6 first to determine if the exception is triggered by DR0. The leading replica's branch counter value and IP (specifically, the IP of the user-mode thread) are stored in the shared kernel data region, and they do not change since the leading replica is stopping at the kernel barrier. When the chasing replica finds that it needs to make progress, the *resume flag* (bit 16) in the FLAGS register has to be set. Remember that DR0 still has the address of the next instruction to be executed, and the instruction breakpoint is still enabled. Without enabling the resume flag, another breakpoint exception will be triggered immediately since the processor is trying to execute the instruction again. The resume flag temporarily disables the breakpoint so we can execute the next instruction without generating a debug exception. However, the instruction breakpoint is still enabled, and the resume flag is cleared by the processor automatically after the breakpoint is skipped; so we can still receive an exception the next time when the instruction is about to be executed.

The movs (move string), cmps (compare string), scas (scan string), lods (load string), and stos (store string) instructions are *string instructions* [Int, 2016a]. These instructions operate on individual elements in a string, and each element can be a byte, word, or doubleword. The register ESI specifies the source element, and the destination element is identified by the EDI register. The string instructions can be used with rep, repe, repz, repne, and repnz prefixes to construct *repeated string operations*; and the value in the register ECX controls the how many times the instructions are repeated. The EDI and ESI registers are automatically incremented or decremented after each iteration so that they point to the next element. Recent processors further optimise the operations initiated by movs and stos if implementation-specific initial conditions are met. The optimised operations are called *fast-string operations*, and they operate in groups (each group may include multiple elements); interrupts or data breakpoints are recognised only on the boundaries of the groups. The 64-bit string operations are similar to the 32-bit ones. But RCX, RDI, and RSI registers are used instead.

The string operations bring us an issue: if we set an instruction breakpoint at a string instruction with one of the rep-like prefixes, for each iteration, a debug exception is generated. Thus, if the value in the ECX register is substantial, the debug exceptions introduce significant overhead. Also, these instructions do not increase the performance counters that we use for counting branches. We avoid setting breakpoints on repeated and fast string operations by inspecting the content of the address pointed by the leading replica's instruction pointer. If the next instruction is one of the repeated string operations, we set the breakpoint to the address following the repeated string instruction for all the replicas. The leading replica does not wait on the barrier in this case, but keeps running until the breakpoint triggers the debug exception; and then, it waits on a special barrier dedicated for the case. When all replicas arrive at the special kernel barrier, the synchronisation protocol restarts from the beginning. We restart the synchronisation process because, although very unlikely, a repeated string instruction may be followed by another repeated string instruction. Restarting the process can handle this case correctly. Nevertheless, thoughtlessly examining the content according to the address provided by a user-mode thread is unwise: the code section of the address to be examined can be paged out so that the examining the content in kernel mode will trigger a kernel-mode page fault which results in a kernel panic. Currently, we disallow paging out code sections.

D.4 An Introduction to GCC RTL

The GCC's front-ends transform source code of all supported languages (C, C++, Objective-C, etc.) to an intermediate, three-address representation called GIMPLE. Complex control structures are lowered into conditional jumps. Target- and language-independent optimisations, for example, inlining, dead code elimination, constant propagation, redundancy elimination, etc., work on the GIMPLE representation. The option `-fdump-tree-all-raw` can be passed to GCC to produce GIMPLE tuples. For example, a simple assignment `a = 5;` is transformed to `gimple_assign <integer_cst, a, 5, NULL, NULL>`. When the target-independent optimisations are done, GIMPLE is expanded to a low-level intermediate representation called RTL (register transfer language), which is used by low-level optimisers (loop optimisation, common sub-expression elimination, modulo scheduling, etc.) and back-ends to generate architecture-specific assembly code. Essentially, RTL is an architecture-independent assembly language for an abstract machine with infinite registers. When various RTL passes finish, a list of RTL instructions will be matched against RTL templates to produce final assembler code.

```
char if_test(void) {
  char c = getchar();
  if (c == 66) return c;
  if (c == 55) c++;
  return c;
}
```

Listing D.9: An example of RTL: source code

RTL employs 5 object types: `expressions`, `integers`, `wide integers`, `strings`, and `vectors`. We focus on expressions since they include branch instructions. An RTL expression is called RTX for short. A function in source code is represented by a double-linked list of RTL objects called `insns`. Each `insn` must be one of the following six expression codes (excerpts from file `rtl.def` in the GCC source code):

- `insn`: an instruction that cannot jump.
- `jump_insn`: an instruction that can possibly jump.
- `call_insn`: an instruction that can possibly call a subroutine but which will not change which instruction comes next in the current function.

- barrier: a marker that indicates that control will not flow through.

- code_label: holds a label that is followed by instructions.

- note: says where in the code a source line starts, for symbol table's sake.

Obviously, we are interested in jump_insn and call_insn for the purpose of counting branches. Note that return instructions are covered by the code jump_insn. Now let us have a look at the code in Listing D.9 and the corresponding insns in Listing D.10 with some unimportant details removed for clarity. The lines guarded by the BEGIN and END pairs are the inserted code for counting branches, and they can be ignored at the moment. The call_insn at line 8 is the function call to getchar(); line 18 *insn* assigns the result of the function call to register r0. Line 21 compares the value of r0 with a constant 66 and sets the condition register reg:CC to 0 if the values are equal. At line 33, the jump_insn returns if the reg:CC equals to 0. The inst at line 44 compares the value in r0 with a constant 55 and again, sets the reg:CC accordingly. The insn at line 48 conditionally sets the r0 to 56 according to the comparison result above. The last jump_insn at line 62 returns from the function.

```
1   /* BEGIN increase branch counter */
2   (insn 72 8 9 (set (reg:SI 9 r9)
3           (plus:SI (reg:SI 9 r9)
4               (const_int 1 [0x1]))) -1
5        (nil))
6   /* END increase branch counter */
7
8   (call_insn:TI 9 72 11 (parallel [
9               (set (reg:SI 0 r0)
10                  (call (mem:SI (symbol_ref:SI ("_IO_getc"))
11                      (const_int 0 [0])))
12              (use (const_int 0 [0]))
13              (clobber (reg:SI 14 lr))
14          ]) /usr/include/bits/stdio.h:46 251 {*call_value_symbol}
15       (nil)
16      (expr_list:REG_CFA_WINDOW_SAVE (use (reg:SI 0 r0))
17          (nil)))
18  (insn:TI 11 9 12 (set (reg/v:SI 0 r0 [orig:110 c ] [110])
19          (zero_extend:SI (reg:QI 0 r0))) loop.c:7 171
               {*arm_zero_extendqisi2_v6}
20       (nil))
21  (insn:TI 12 11 67 (set (reg:CC 100 cc)
22          (compare:CC (reg/v:SI 0 r0 [orig:110 c ] [110])
23              (const_int 66 [0x42]))) loop.c:8 217
                   {*arm_cmpsi_insn}
24       (nil))
25  (insn 67 12 73 (use (reg/i:SI 0 r0)) -1
26       (nil))
```

```
/* BEGIN increase branch counter */
(insn 73 67 13 (set (reg:SI 9 r9)
        (plus:SI (reg:SI 9 r9)
            (const_int 1 [0x1]))) -1
     (nil))
/* END increase branch counter */
(jump_insn:TI 13 73 14 (set (pc)
        (if_then_else (eq (reg:CC 100 cc)
                (const_int 0 [0]))
            (return)
            (pc))) loop.c:8 257 {*cond_return}
     (expr_list:REG_DEAD (reg:CC 100 cc)
        (expr_list:REG_BR_PROB (const_int 1991 [0x7c7])
            (nil)))
 -> return)
(note 14 13 55 [bb 3] NOTE_INSN_BASIC_BLOCK)
(note 55 14 63 NOTE_INSN_DELETED)
(insn:TI 63 55 64 (set (reg:CC 100 cc)
        (compare:CC (reg/v:SI 0 r0 [orig:110 c ] [110])
            (const_int 55 [0x37]))) loop.c:11 217
               {*arm_cmpsi_insn}
     (nil))
(insn:TI 64 63 27 (cond_exec (eq (reg:CC 100 cc)
            (const_int 0 [0]))
        (set (reg/v:SI 0 r0 [orig:110 c ] [110])
            (const_int 56 [0x38]))) loop.c:11 3211 {*p
               *arm_movsi_insn}
     (expr_list:REG_DEAD (reg:CC 100 cc)
        (nil)))
(insn 27 64 74 (use (reg/i:SI 0 r0)) loop.c:24 -1
     (nil))
/* BEGIN increase branch counter */
(insn 74 27 69 (set (reg:SI 9 r9)
        (plus:SI (reg:SI 9 r9)
            (const_int 1 [0x1]))) -1
     (nil))
/* END increase branch counter */
(jump_insn 69 74 68 (return) loop.c:24 256 {*arm_return}
     (nil)
 -> return)
```

Listing D.10: An example of RTL: RTL expressions

D.5 Implementation Details of Compiler-Assisted CC-RCoE for ARM

We implement the GCC plugin based on GCC version 4.8.4. Older or newer GCC versions may change internal functions, structures, and macros, so the plugin may or may not be compatible with other versions.

On ARM Cortex-A9 processors, the x86 resume flag equivalent feature is missing; the feature is especially useful if an instruction point is set inside a loop. Without this feature, the kernel cannot disable the breakpoint temporarily to resume the interrupted instruction without generating another exception. Instead, the kernel has to remove the breakpoint and enter single-step mode by setting the debug register to generate an exception when any instruction other than the previously interrupted instruction is the next-to-run instruction; having executed the previously-interrupted instruction, the processor generates a prefect exception because of single-stepping. Finally, the kernel re-enables the breakpoint and disables single-stepping, so the processor can generate another exception the next time when it hits the breakpoint.

The general purpose registers are 32-bit; if we assume that the processor runs at 1 GHz and that the frequency of branch instruction is one out of ten, the 32-bit registers overflow approximately every 42.95 seconds. We need to reset the R9 register to 0 after each synchronisation; since the replicas need to synchronise for kernel preemption timer interrupts, which are usually 10 to 100 times per second, the overflow does not present an issue.

Lastly, we mention that all source code must be recompiled with the plugin, including libraries. Actually, the library `libgcc.a` is linked to the binary implicitly. GCC assumes that it can safely call functions in `libgcc.a` during code generation as it sees fit. The functions included in `libgcc.a` depend on the target architecture, the configuration when building GCC, and command-line options passed to the compiler. On ARM, the library implements arithmetic functions (div, mod, divmod, etc.), floating-point functions (single and double precisions), and other support functions. We obtain the source files from GCC source code, modify functions that are implemented in assembly and used by GCC during compilation, and compile these files with applications. We also use the option `-nostdlib` to instruct GCC not to link the `libgcc.a` by default so that our modified version is used.

D.6 An Introduction to the x86 Hardware Virtualisation Technology

VMX instructions, `vmxon`, `vmxoff`, `vmlaunch`, `vmresume`, `vmread`, `vmwrite`, `vmclear`, `vmptrst`, and `vmptrld` are added in the VMX root operation and are only available to the kernel in ring 0. The `vmxon` instruction enables VMX operation; `vmlaunch` switches from root operation to non-root operation and starts to execute a virtual machine. When certain events specified by the VMM happen, the execution of a virtual machine is interrupted, and a transition is made from non-root operation to root operation. The VMM handles the events and restarts the VM with the `vmresume` instruction. Transitions from root operation to non-root operation are called *VM entries*; transitions from non-root operation to root operation are called *VM exits*.

An seL4 *VCPU* kernel object contains the VMCS (virtual machine control structure) memory region defined by Intel, and the VMCS controls the behaviour of the corresponding VM. The data in VMCS can be divided into 6 groups: guest-state area, host-state area, VM-execution control fields, VM-exit control fields, VM-entry control fields, and VM-exit information fields. The guest-state area is used to save the processor state when VM exits happen, and the state is restored from the area on VM entries. The data of host-state area is restored to the current processor on VM exits.

The VM-execution control fields govern the processor behaviour in non-root operation. They define under what conditions VM exits should be triggered. For example, whether an interrupt, a `rdtsc` instruction, or a `cr3` reloading triggers a VM exit is controlled by these fields. The second-level address translation is also managed by one of the fields. The VM-exit control fields determine the behaviour of VM exits. For instance, saving the debug control registers or not is specified by one of these fields. VM-entry control fields prescribe the actions to be performed during VM entries. One of most frequently used fields is the VM-entry interruption-information field that controls event injection (e.g., interrupts, debug exceptions, etc.). Lastly, the VM-exit information fields describe the reason for the most recent VM exit, including the cause (interrupts, accesses to control registers, privileged instructions, etc.) and additional qualifications.

Memory virtualisation is achieved by the two-level address translation mechanism: The first level is managed by the guest Linux kernel, and it translates *guest virtual addresses* to *guest physical addresses*. The second level is controlled by a VMM through *EPT* (extended page table) objects, and it translates *guest physical addresses* to *host physical addresses*. The VMM determines physical memory allocated to the VM by constructing and populating the second-level translation tables, restricting the physical addresses that can be accessed by the VM.

The microkernel does not handle a VM exit directly; a VM exit is redirected to the user-mode VMM through an endpoint. The VMM inspects the VM exit reason and qualifications and decides if the exit is valid or not according to policies. For example, for a VM exit triggered by an access to a physical memory region that is not mapped in second-level address translation tables, the VMM may decide to terminate the VM by destroying the VCPU object and EPT objects if allocating more physical memory to the VM is not allowed. If the policy does not allow the VM to use the `rdtsc` instruction to read absolute values from the CPU cycle counter, the VMM can set corresponding VM-execution control field to generate a VM exit each time the VM executes the `rdtsc` instruction; so the VMM can emulate the instruction and then resume the VM by replying to the endpoint.

Appendix E

Supplementary Code

E.1 Memory Copy Functions

```
asm volatile (
  "push   {r3-r10}      \n\t"
  "1:                   \n\t"
  "pld    [%1, #0]      \n\t"
  "pld    [%1, #32]     \n\t"
  "pld    [%1, #64]     \n\t"
  "pld    [%1, #96]     \n\t"
  "pld    [%1, #128]    \n\t"
  "pld    [%1, #160]    \n\t"
  "pld    [%1, #192]    \n\t"
  "pld    [%1, #224]    \n\t"
  "pld    [%1, #256]    \n\t"
  "pld    [%1, #288]    \n\t"
  "pld    [%1, #320]    \n\t"
  "pld    [%1, #512]    \n\t"
  "pld    [%1, #640]    \n\t"
  "pld    [%1, #768]    \n\t"
  "pld    [%1, #1024]   \n\t"
  "ldmia  %1!, {r3-r10}\n\t"
  "stmia  %0!, {r3-r10}\n\t"
  "ldmia  %1!, {r3-r10}\n\t"
  "stmia  %0!, {r3-r10}\n\t"
  "subs   %2, %2, #0x40\n\t"
  "bge    1b            \n\t"
  "pop    {r3-r10}      \n"
  : "+r"(dest), "+r"(src), "+r"(size)
  :
  : "memory"
);
```

Listing E.1: The memcpy function for ARM

```
asm volatile (
  "1:                        \n\t"
  "movups 0(%%rsi),   %%xmm0 \n\t"
  "movups 16(%%rsi),  %%xmm1 \n\t"
  "movups 32(%%rsi),  %%xmm2 \n\t"
  "movups 48(%%rsi),  %%xmm3 \n\t"
  "movntdq %%xmm0,  0(%%rdi) \n\t"
  "movntdq %%xmm1,  16(%%rdi)\n\t"
  "movntdq %%xmm2,  32(%%rdi)\n\t"
  "movntdq %%xmm3,  48(%%rdi)\n\t"
  "movups 64(%%rsi),  %%xmm4 \n\t"
  "movups 80(%%rsi),  %%xmm5 \n\t"
  "movups 96(%%rsi),  %%xmm6 \n\t"
  "movups 112(%%rsi), %%xmm7 \n\t"
  "movntdq %%xmm4, 64(%%rdi) \n\t"
  "movntdq %%xmm5, 80(%%rdi) \n\t"
  "movntdq %%xmm6, 96(%%rdi) \n\t"
  "movntdq %%xmm7, 112(%%rdi)\n\t"
  "movups 128(%%rsi), %%xmm0 \n\t"
  "movups 144(%%rsi), %%xmm1 \n\t"
  "movups 160(%%rsi), %%xmm2 \n\t"
  "movups 176(%%rsi), %%xmm3 \n\t"
  "movntdq %%xmm0, 128(%%rdi)\n\t"
  "movntdq %%xmm1, 144(%%rdi)\n\t"
  "movntdq %%xmm2, 160(%%rdi)\n\t"
  "movntdq %%xmm3, 176(%%rdi)\n\t"
  "movups 192(%%rsi), %%xmm4 \n\t"
  "movups 208(%%rsi), %%xmm5 \n\t"
  "movups 224(%%rsi), %%xmm6 \n\t"
  "movups 240(%%rsi), %%xmm7 \n\t"
  "movntdq %%xmm4, 192(%%rdi)\n\t"
  "movntdq %%xmm5, 208(%%rdi)\n\t"
  "movntdq %%xmm6, 224(%%rdi)\n\t"
  "movntdq %%xmm7, 240(%%rdi)\n\t"
  "addq  $256, %%rsi         \n\t"
  "addq  $256, %%rdi         \n\t"
  "subq  $256, %%rdx         \n\t"
  "jnz   1b                  \n\t"
  "sfence"
  : : : "rsi", "rdi", "rdx"
);
```

Listing E.2: The `memcpy` function for x86-64