Großer Beleg

Improving SWIFT Error Coverage by Fingerprint-Based State Comparison

Christian Menard

30. April 2014

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. Hermann Härtig
Betreuender Mitarbeiter:	Dipl.-Inf. Björn Döbel

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 30. April 2014

Christian Menard

Contents

List of Figures

List of Tables

List of Listings

Acronyms

CISC	complex instruction set computer.
COTS	commercial off-the-shelf.
CRC	cyclic redundancy check.
DUE	detected unrecoverable error.
ECC	error correcting code.
FC	Fletcher's checksum.
FPGA	field-programmable gate array.
HDL	hardware description language.
IPC	inter process communication.
ISA	instruction set architecture.
MSR	model-specific register.
RMT	redundant multithreading.
SDC	silent data corruption.
SEU	single event upset.
SMT	simultaneous multithreading.
SWIFT	software implemented fault tolerance.
UTCB	universal thread control block.

1 Introduction

Modern processors are built under the assumption that transistors always function correctly. In the past 30 years studies have shown that this assumption does not hold [Zie+96]. Transistors are exposed to various environmental influences. High energetic cosmic rays [ZL80], heat flux [Bor05], and brown-out effects [Ern+03] can cause temporal malfunctions. Transient errors caused by such a malfunctioning transistor are called soft errors. The probability for the occurrence of a soft error increases with each chip generation as hardware feature sizes shrink and the number of transistors integrated into a single chip increases. Modern chips are built under pessimistic assumptions to keep error probability low, which leads to inefficient designs. Therefore we have to drop the idea of a perfect transistor and develop fault tolerant designs.

Romain is a replication framework that provides transparent protection against soft-errors as an operating system service [DHE12]. Any application running on top of Romain is protected using *n*-way modular redundancy. As Romain is fully implemented in software and runs on top of an operating system, access to the processor's architectural state is limited. This leads to reduced error coverage and increased error detection latency. In contrast hardware-only solutions can provide much better error detection characteristics but are expensive as they rely on specialized circuitry. Therefore they will not become part of commercial off-the-shelf (COTS) systems in the near future.

I propose to use fingerprinting as a hardware extension to assist software implemented fault tolerance (SWIFT). A fingerprint summarizes all updates to the architectural state into one single checksum. This reduces the bandwidth needed for full state comparison of redundant units to a single value. Therefore fingerprinting is appealing for SWIFT designs where fingerprints can significantly improve error coverage. In this work I extend an existing COTS processor to support fingerprinting. I evaluate my design with Romain and show that fingerprinting can significantly improve the error detection characteristics of Romain.

Chapter 2 discusses causes for soft errors and previously proposed designs for fault tolerant systems. In this chapter I will describe Romain and fingerprinting in more detail and explain the gem5 simulator, on which my work is based. Chapter 3 discusses the design and implementation of the fingerprint extension. Additionally, I will explain the modifications to the software stack that are necessary to perform fingerprint comparison in Romain. Chapter 4 describes the evaluation of my design based on various experiments and shows the potential of fingerprinting. The final chapter concludes this work and gives an outlook on possible future use-cases.

2 State of the Art and Motivation

In this chapter I want to give an overview on soft errors and existing solutions for error detection and correction. First I explain possible causes for errors and define the terms that I will use in the rest of this work. Then I give a general overview on existing hardware and software solutions for error detection and recovery. Later I discuss fingerprinting as a hardware solution and the replication framework Romain as a software solution in more detail to motivate my work. In the final section I discuss the processor simulator gem5 on which my work is based.

2.1 Soft Errors

Back in 1979 May and Woods examined soft errors in DRAMs and identified alpha-particles as an error source [MW79]. Alpha-particles originate from small amounts of radioactive contaminants in the chip and packaging materials. While traveling through the semiconductor alpha-particles generate electron-hole pairs along their track. This may cause current pulses and hence disturb the device's normal operation. This kind of error is not permanent and can occur randomly distributed over time and space. Therefore it is called a soft error. Later on Ziegler and Lanford demonstrated that cosmic rays also can cause soft errors [ZL80]. At sea level 95% of the particles capable of causing soft errors are neutrons [Zie96].

Today's trend is to decrease hardware feature sizes to provide more functionality per equally sized chip. Therefore the vulnerability of modern processors against soft errors increases significantly [Shi+02]. Additionally this trend allows other error sources to come into effect. The sub-wavelength lithography, used in modern factoring processes, uses a wavelength much higher than the hardware feature size, leading to a large variance of gate switch delays [Nas10]. Temporary changes in transistor characteristics caused by heat flux may as well lead to soft errors [Bor05]. Finally undervolting and frequency scaling are causes for soft errors that are abet by decreasing supply voltages [Ern+03].

2.2 Terminology

Avizienis et al. define an error as “that part of the system state that may cause a subsequent failure”. A fault “is the adjudged or hypothesized cause of an error” [Avi+01]. I will use the term single event upset (SEU) to model radiation introduced soft errors. An SEU is a state change caused by ionization inside a register or memory cell due to radiation. Due to SEUs the content of a single “bit” can be modified causing a *bit-flip*.

Such SEUs can have different outcomes. Weaver et al. classify the possible outcomes into four categories. SEUs that do not affect the program outcome because the affected bits are not read or are masked by later instructions are called *benign faults*. If an SEU influences the program’s outcome and if the faulty bit is not error protected, a silent data corruption (SDC) occurs. SDCs can be avoided by adding simple error detection mechanisms like parity. This way we know when a program’s output is erroneous but we cannot recover. Such errors are called detected unrecoverable errors (DUEs). One approach to reduce the error rate of a system is not only to add error detection mechanisms but also recovery mechanisms. In a system that is protected in such a way an SEU that is detected and corrected causes no error. [Wea+04]

2.3 Error Detection and Fault Tolerance

Systems that are functioning correctly despite the existence of faults are called fault tolerant. A fault tolerant system not only detects errors, but also corrects them. In the past several different approaches for fault tolerance have evolved. In this section I give a short overview on previously proposed solutions for fault detection and recovery.

As especially memory is vulnerable for soft errors, techniques for memory protection where developed early. Today parity and error correcting code (ECC) are widely used for protection of memories and permanent storage devices [Muk08]. But besides memory also registers and logical blocks are affected by soft errors. One approach that is used in existing commercial fault tolerant systems is replication. Replication based systems compare the outputs of redundant components for error detection. Error recovery is performed by majority voting. The IBM G5 is an example for a replication based system [Sle+99]. It uses multiple replicated, lockstepped pipelines within a single core. Another example is the Tandem NonStop architecture that uses replicated, lockstepped processors [McE81]. The DIVA architecture extends an out-of-order pipeline by a simple checker that verifies and corrects instruction results [Aus99]. There are also several proposals for fault-tolerant simultaneous multithreading (SMT) processors. Rotenberg used an extended SMT processor to detect errors by full state comparison between two redundant threads [Rot99]. Vijaykumar, Pomeranz, and Cheng extended the redundant multithreading (RMT) approach to support recovery on SMT processors [VPC02]. Reinhardt and Mukherjee proposed the SRT processor that reduces the bandwidth needed for state comparison by only comparing stores across redundant threads [RM00].

Hardware implemented fault tolerance is attractive, as software running on such systems does not need to be fault aware. However, the design process for such systems is complex and manufacturing is more costly, as redundant components and additional logic are needed. Therefore it will take some time until fault tolerance techniques become part of COTS systems. Another drawback of hardware-only solutions is that they statically add redundancy. A possible scenario for a protected system would be to run applications with mixed criticality. Some applications might only need error detection or are not critical and need no protection at all. Therefore static redundancy increases run-time costs for uncritical applications.

Software-only designs allow for more flexibility and can run on top of any COTS processor. Oh, Shirvani, and McCluskey proposed control flow checking based on compiler generated signatures [OSM02b] and used duplicated instructions for detection of computational errors [OSM02a]. Reis et al. integrated both approaches into the SWIFT compiler [Rei+05]. Also arithmetic codes can be used instead of, or in addition to, duplicated instructions. For example ED4I [OMM02] is a compiler that is based on AN-Codes. However, ED4I has vulnerabilities that Fetzer, Schiffel, and Süsskraut addressed with their AN-Encoding compiler EC-AN [FSS09].

The discussed solutions have a major drawback. Either the software developer has to design his software in a fault aware way or a special compiler has to be used. Thus it is necessary to recompile the whole software stack to make unprotected applications fault tolerant. However, recompilation is impossible for proprietary third-party software.

2.4 Romain

Romain is a replication framework that is developed by the TU Dresden operating systems group [DHE12]. Romain eliminates the need for recompilation of applications by implementing a fault aware operating system service. This service allows unmodified applications to run in a fault tolerant context. Romain uses transparent redundant multithreading [RM00] to provide a fault tolerant environment that is transparent to the protected application.

To properly explain Romain, I need to talk about the underlying operating system and its kernel objects. Romain is based on operating system that consists of the Fiasco.OC microkernel [Dre12] and the L4 Runtime Environment (L4Re) [Dre14]. Both components are also developed by the TU Dresden OS group. Fiasco.OC is part of the L4 family of microkernels. All microkernels have in common that they reduce the kernel's functionality to a minimum. Additionally the kernel does not implement any policies, it only provides mechanisms. For a deeper understanding on microkernels please have a look at the literature [Lie95].

The fundamental philosophy of Fiasco.OC is that everything is an object. The main kernel objects that Fiasco.OC provides are tasks, threads, IRQs, IPC-gates, and factories [LW09]. A task is a protection domain and consists of a virtual memory address space and a capability table that holds references to other objects. A task can consist of multiple threads that represent a unit of execution and share the tasks resources. IRQs are senders of asynchronous inter process communication (IPC) messages that are bound to a specific thread. IRQs can be invoked by hardware interrupts or user actions. The IPC-Gate is a special kernel object that is used to establish synchronous communication channels between user-level objects. Finally a factory is an object that creates new kernel objects.

In addition to the main objects Fiasco.OC also provides the virtual CPU (vCPU) object. A vCPU is a specialized supplemented thread and was introduced by Lackorzynski, Warg, and Peter to allow for more efficient operating system rehosting [LWP10]. However, the mechanisms vCPUs provide are also useful for Romain. A vCPU executes user-level code within an address space. Whenever it raises an exception, for instance a system call or a page fault, it is migrated to another process. This master process is also called "user-level kernel" because it is responsible for handling the exception and has full control over the vCPUs state. In particular the master is able to access and modify a vCPU's register values by accessing a simple data structure.

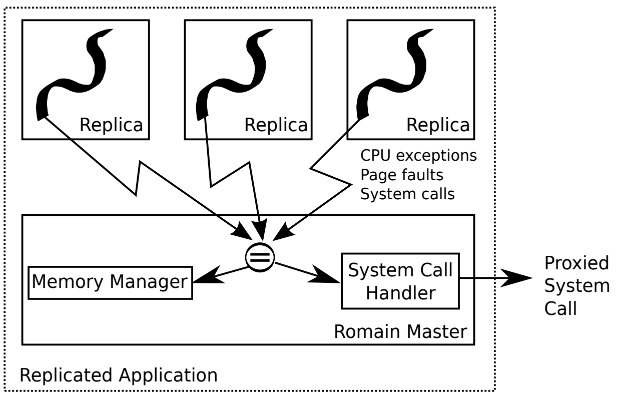


Figure 2.1: Replication in Romain [DHE12]

The fundamental principle of Romain is shown in Figure 2.1. Romain provides a master process that starts the application to be protected n times. The n replicas are vCPUs that execute isolated in separate address spaces. Whenever a replica wants to update structures outside of its domain, the master needs to verify these updates to ensure that no faulty state influences the operation of the rest of the system. A replica externalizes its state whenever a event like a system call, a page fault, or another CPU trap occurs. To ensure the correctness of such externalizing events, the master blocks a trapped replica until all replicas raised their next exception. Then the master checks if all replicas raised the same exception and compares their architectural state. If the replica states match, the master is responsible for handling the exception. Otherwise the master detected an error. If more than two replicas are used, the master will recover from that error using majority voting.

Romain's approach of software implemented replication has several advantages. Romain allows to protect unmodified applications on COTS hardware. Additionally Romain provides a level of flexibility that is only possible in software solutions. The system designer is able to decide which applications need to be protected and if they need error recovery or only error detection. A drawback of software replication is an increased error detection latency. If only externalized states are compared on events like system calls, an error in the architectural state will not become visible until an event occurs that externalizes this error. To reduce error detection latency and increase the error coverage, Romain also compares register states of its replicas and thus detects architectural errors earlier. But registers are just a small subset of the architectural state that also includes memory and caches. However, comparing whole memory regions would add a large overhead and is therefore not feasible. This lack of error coverage is a fundamental problem of Romain and needs to be resolved. In his bachelor thesis Kriegel proposed to perform state comparison in constant intervals by using a watchdog [Kri13]. This way the average error detection latency can be reduced as state comparisons are performed in regular intervals. However, errors still may reside unseen in memory as Romain's state comparisons do not cover the full architectural state. This may lead to situations where one ore more replicas contain undetected errors. As Romain performs majority voting for recovery, an undetected error has the potential to transfer to another replica. When this error externalizes, multiple replicas have the same erroneous state. If these replicas are the majority, Romain will consider them to be error free. At this point the error externalizes and the system fails. Therefore the aim of my work is to significantly increase error coverage in Romain.

2.5 Fingerprinting

Smolens et al. proposed to use fingerprints for state comparison in replication based hardware designs [Smo+04]. A fingerprint is a hash value that summarizes the architectural state of a processor. It is computed during program execution on the sequence of updates to the architectural state. Therefore the sequence can be characterized by its fingerprint and errors in the architectural state can be detected by comparing the fingerprints across multiple redundant cores. By compressing all updates to the architectural state into a single value, fingerprinting drastically reduces the bandwidth needed for full state comparison. According to Smolens et al. fingerprinting is the only error detection mechanism that allows for high error coverage while maintaining high input–output performance and reducing the detection bandwidth to a minimum.

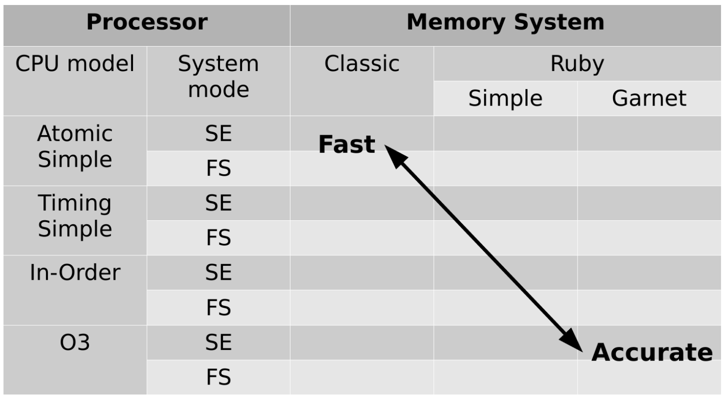
In the work of Smolens et al. error detection and recovery are completely implemented in hardware. Fingerprints are compared at certain points and if they match, a checkpoint is created. A checkpoint is a snapshot of all architectural registers and memory values, which contains only error-free data. If the system detects an error, it simply rolls back to the previous checkpoint and restarts execution. This approach significantly simplifies error detection but still relies on redundant cores and a special recovery mechanism.

Fingerprinting is a promising technique as it allows for fast state comparison. Therefore it is also interesting for fault detection and recovery implemented in software. I propose to use fingerprinting as a hardware extension that assists software implemented replication by providing easy and fast error detection. This way a small hardware addition can significantly increase the error detection characteristics of a software implemented fault-tolerance system like Romain. The necessary hardware addition is rather small as it only needs to compute fingerprints but does not need to implement any replication or recovery mechanisms. Therefore a fingerprint extension is more likely to become part of future COTS systems than a hardware-only fault-tolerance system.

2.6 gem5

In the previous section I proposed to use fingerprinting as a hardware extension to allow for high error coverage in software implemented replication systems. I need a flexible platform as a basis for my work to design, implement, and evaluate this extension. This platform needs to implement an x86 processor as Romain only runs on x86. As a real hardware implementation is out of reach, I based my work on the gem5 processor simulator [Bin+11].

At this point I need to clarify the term x86. x86 refers to a family of backward compatible instruction set architectures (ISAs) originating from the Intel 8086 processor. This work is more specifically based on the AMD64 ISA [Adv13] that provides a 64-bit extension and is also known as x86-64. I had to choose this ISA because gem5 only fully supports the 64-bit mode of x86. So when I refer to x86 in the rest of this work, I refer to AMD64 in particular.



Processor		**Memory System**		
CPU model	System mode	Classic	Ruby	
			Simple	Garnet
Atomic Simple	SE			
	FS			
Timing Simple	SE			
	FS			
In-Order	SE			
	FS			
O3	SE			
	FS			

Figure 2.2: gem5: speed vs. accuracy spectrum [Bin+11]

The gem5 simulation framework provides a flexible simulation environment that is capable of handling several ISAs including x86. By implementing various exchangeable CPU models, memory back-ends, and execution modes, gem5 can easily be adopted to a given task. The exchangeable components allow for a trade-off between simulation speed and accuracy. The use of detailed models provides an accurate simulation but also increases simulation time significantly. If simulation detail is less important, accuracy can be traded in for simulation speed by choosing simple models.

In total gem5 implements four CPU models, two execution modes, and two memory back-ends. The models are fully exchangeable and each one provides a unique level of simulation detail. Each combination of models marks a unique spot in the speed-vs.-accuracy spectrum as is shown in Figure 2.2. In the following I will briefly describe the characteristics of the different models.

CPU Model Currently gem5 implements four different CPU models. The *AtomicSimple* model is a simple CPU that executes a single instruction per cycle. It does not care about the timing of memory references at all. *TimingSimple* is a similar model, but it also simulates the timing of memory references. The *InOrder* and *O3* models are more detailed, pipelined CPUs. As the name implies *InOrder* is an in-order CPU. *O3* is a complex out-of-order, superscalar CPU model that uses advanced features like branch prediction, instruction and load/store queues, functional units, and memory dependence predictors.

Execution mode Each CPU model can execute code in either one of two modes. System-call emulation (SE) mode is used to execute simple Linux applications without the need of modeling devices and an operating system. Whenever an application performs a system call, it is handled by gem5. In most cases gem5 just calls the corresponding system call of the host system. Full-System (FS) mode is used to execute kernel level and user level code. In this mode all the devices and the operating system are modeled.

Memory Back-End There are two basic memory system models. The Classic model (derived from M5) is a fast model that is easy to handle but does not aim for a detailed simulation. In contrast the Ruby model (derived from GEMS) provides a flexible and highly configurable memory back end that allows for accurate simulation of a diversity of cache coherent memory systems.

All in all, gem5 is a good choice for the purpose of extending an ISA because of its flexible and modular design. However, it also has to be stated that there is no real alternative. I need a simulator that is capable of simulating an x86 CPU and running a full system on top of it. Alternatives to gem5 that are capable of simulating such a system are MARSS [Pat+11], Bochs [The13], and QEMU [Bel05]. MARSS is a simulator based on QEMU and provides a fast simulation environment that performs cycle-accurate simulation of superscalar multicore x86 processors. QEMU and Bochs are emulators that are designed with regard to the needs of operating system developers who need emulators to debug their systems. As MARSS is optimized for accuracy and QEMU as well as Bochs are optimized for speed and an easy to use debugging interface, their designs do not allow for extensibility. In contrast gem5 is designed for computer architecture research and therefore uses a modular and extensible design.

3 Design and Implementation

In Chapter 2 I gave an overview on existing techniques for error detection and recovery. I explained fingerprinting as a hardware solution and the replication framework Romain as a software solution in more detail. Fingerprinting allows for high error coverage and low error detection latency but also needs a complex hardware design when replication and error recovery are implemented in hardware. Romain on the other hand fully implements replication in software but has a lack of error coverage. My goal in this work is to integrate both approaches into a system that provides the flexibility of a software solution and the error coverage of a hardware mechanism.

In more detail my goal is to implement fingerprinting as an extension to the x86 ISA in gem5. Why I chose gem5 as the basis for my work is explained in 2.6. My design goal for the fingerprint extension is to reach a high error coverage and a high error detection probability. Furthermore I want this extension to be appealing for future COTS systems. Therefore I designed it with regard to possible implementations in real hardware. This means the extension should be realizable in an efficient way, or in other words: the amount of logic and registers needed for an implementation should be small while the possible throughput should be high. Additionally I want my design to be a backward compatible addition to the AMD64 ISA. This means instead of modifying any existing behavior I just want to add new functionality.

My target system is visualized in Figure 3.1. It consists of an extended version of gem5's *AtomicSimple* CPU that I call *AtomicSignatureCPU* and that supports fingerprinting. The microkernel Fiasco.OC runs on top of this CPU. The kernel manages the fingerprints computed by the CPU and associates them with threads. Fiasco.OC also provides an interface that allows user processes to read and write fingerprints. Romain runs on top of Fiasco.OC and uses this interface for state comparison across it replicas.

In the following sections I explain my design process and implementation. In Section 3.1 I discuss possible checksum algorithms based on considerations regarding hardware implementations. Then I explain the design of my fingerprint extension in Section 3.2. An explanation of my implementation in gem5 follows in Section 3.3. After that I describe how I modified the kernel in Section 3.4. In the final Section 3.5 I conclude by discussing fingerprint comparison in Romain.

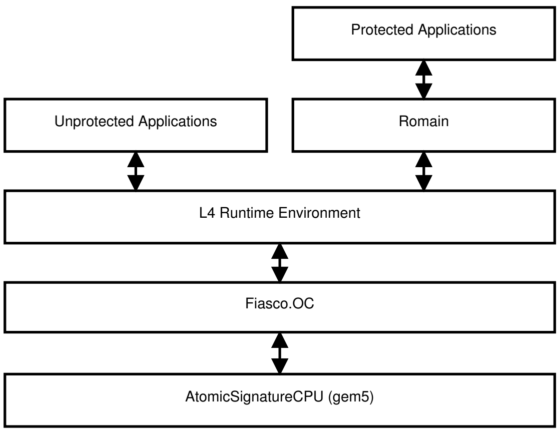


Figure 3.1: The target system

3.1 Choosing a Checksum Algorithm

Before I can discuss the actual design of the fingerprint extension, I need to explain my choice of a checksum algorithm. This algorithm is used for fingerprint computation and significantly influences the chracteristics of the final design. To comply with my design goals the algorithm should have a low computational complexity while providing good error detection characteristics. Namely the error detection probability should be high and the algorithm should be able to protect a block of sufficient size to allow for larger windows between state comparisons. The block size is important as the total block of protected data grows with every instruction executed. If state comparisons are only performed on events, the number of instructions executed between two state comparisons can easily expand to several millions.

Cryptographic hash functions like MD5 or SHA-1 are out of question because the computational complexity is too high. As a fingerprint does not need to fulfill any cryptographic requirements, the additional cost in computation is not justified. Algorithms that are widely used for the purpose of error detection are cyclic redundancy check (CRC) and Fletcher's checksum (FC). Now I will give a short overview on these algorithms, discuss there characteristics, and then explain my decision.

CRC is an error-detecting code that is widely used in network applications. Here the sender attaches a signature to the data block at transmission. The sender computes this signature by dividing a bit-stream message by a given generator polynomial. The remainder of this division is the signature that is sent alongside the data block. The generator polynomial can be chosen arbitrarily, but it has a big influence on the algorithm's error detection characteristics. The receiver performs the same computation on the received bit stream and compares its result to the signature received alongside the data block. If the two signatures match, the data can be assumed to be error-free. Otherwise the receiver detected an error and requests a new transmission. More details on CRC can be found in the literature [PB61] [RK06].

Fletcher's checksum is another algorithm commonly used for signature calculation [Fle82]. Fletcher devised this algorithm in the late 1970s. His objective was to approach an error detection probability similar to CRC while reducing the complexity by using summation techniques. To calculate Fletcher's checksum, an input stream is divided into n-bit blocks. A simple checksum is computed by calculating a n-bit modular sum of all blocks. This simple checksum is weak because it is insensitive to the order of blocks. In addition the set of possible checksum values is small (for small n) and hence the probability that two sequences have the same checksum is high. Fletcher addresses these two issues by computing a second n-bit modular sum of all simple checksums. The concatenation of the simple checksum and the sum of simple checksums forms the $2n$-bit Fletcher's checksum.

Satran and Sheinwald compared the performance of four 32-bit checksum algorithms: CRC-32, CRC-32C, Fletcher-32 and Adler-32. Adler-32 is a variation of Fletcher's checksum. It computes the sums using the modulo 65,521 that is the highest prime number smaller then 2^{-16}. This allows Adler-32 to detect some errors Fletcher-32 is not able to detect, but overall it has been found that Fletcher-32 has a slightly better error detection probability [Max06]. CRC-32 is used in the Ethernet protocol and is defined by IEEE 802.3 [IEE02]. It uses the generator polynomial 0x04C11DB7. CRC-32C uses the generator polynomial 0x1EDC6F41, which improves the error detection characteristics for big data blocks [CBH93]. Satran and Sheinwald showed that the probability of an undetected error (P_{ud}) of CRC-32 and CRC-32C is 10^5 times smaller than P_{ud} of Adler-32 and 10^4 times smaller than P_{ud} of Fletcher-32. Additionally CRC-32C can protect a much larger data block of 512MB while guaranteeing a Hamming distance of three. In contrast this guarantee holds only for block sizes up to 8KB for CRC-32 and 64KB for Adler-32 as well as Fletcher-32. A Hamming distance of three is important to ensure that a error is not only detectable but also correctable. [SS02]

The efficiency of an algorithm's hardware implementation is another important metric. When implemented in software, Fletcher's checksum is typically considered to be computationally cheaper while providing less robust error detection than CRC [MK09]. When implemented in hardware this statement does not hold. Caplan et al. developed and evaluated hardware implementations of CRC and Fletcher's checksum in FPGA (field-programmable gate array). Their results show that CRC is much more efficient than Fletcher's checksum in terms of throughput per register and in terms of throughput per ALUT. ALUT stands for adaptive lookup table and represents a certain amount of logic needed inside an FPGA. Even a 64-bit CRC achieves higher throughput per ALUT or per register than an 8-Bit Fletcher's checksum. [Cap+14]

The characteristics of the discussed algorithms are summarized in Table 3.1. P_{ud} is the probability of an undetected error. The block size is expressed in bits and is the maximum block size that can be protected while maintaining a hamming distance of three. Throughput, the amount of logic needed, and efficiency refer to a possible hardware implementation.

Table 3.1: Characteristics of selected checksum algorithms

Algorithm	P_{ud}	Block size	Throughput	Logic	Efficiency
CRC-32	10^{-40}	2^{16}	⊕	⊕⊕	⊕⊕
CRC-32C	10^{-40}	$2^{31}-1$	⊕	⊕	⊕
Fletcher32	10^{-36}	2^{19}	⊖	⊖	⊖
Adler32	10^{-35}	2^{19}	⊖	⊖⊖	⊖⊖

These values are difficult to compare, because Caplan et al. used different metrics than Satran and Sheinwald. Therefore the table shows representative indicators.

I chose to use CRC-32C for fingerprint computation, because it fulfills the requirements I discussed in the beginning of this section and is therefore a good basis for achieving my design goals. CRC-32C can protect large sized blocks while having the best error detection probability of the discussed algorithms. CRC-32C can be implemented much more efficiently than Fletcher-32 and Adler-32, but in total it needs slightly more logic than CRC-32. However, a slightly larger footprint does not violate my requirements as the total logic needed to implement CRC-32 or CRC-32C is negligible compared to the amount of logic needed to build a processor [SS02].

An alternative to hash functions was proposed by Sogomonyan et al. [Sog+01]. They used scan-chains to get a signature of the architectural state. Scan-chains are used for hardware debugging and therefore are already present on most chips. A scan-chain connects a chain of many flip-flops in a way that allows to shift their values through the chain. The resulting sequence of 1-bit outputs provides information on the architectural state and can be used for state comparison. A drawback of this solution is an increased error detection latency, because it may take some time until an error propagated through the whole scan-chain. Additionally this solution depends on the existence of scan-chains. If scan-chains are used and which parts of the architectural state they cover highly depends on the actual implementation.

3.2 General Design

So far I only talked about compressing the architectural state into one single fingerprint. However, I implemented two separate fingerprints: one for data and one for instructions. The data fingerprint summarizes all writes to registers and memory. Whereas the instruction fingerprint summarizes the instruction flow. The advantage of having a separate instruction fingerprint is that this fingerprint can be pre-calculated for certain parts of the program. For example this allows for hardware assistance in control-flow checking based on block signatures as proposed by Oh, Shirvani, and McCluskey [OSM02b]. Splitting the fingerprints is also wise with regard to possible hardware implementations. On a chip functional units are not always located close together. Collecting values from different units could introduce long electrical paths and therefore could limit the maximum clock frequency [Smo07]. It might be reasonable for future designs that are closer to the hardware to split the fingerprints even further, for instance to divide the data fingerprint into a register and a memory fingerprint.

As I design the fingerprint extension to be used in SWIFT systems like Romain, fingerprints will be associated to threads in those systems. A fingerprint associated to a thread shall summarize all updates to the architectural state that are committed in the thread's context. It is important that fingerprints of identical threads match on each instruction. Therefore a fingerprint may only depend on instructions executed in the thread's context and may not depend on any indeterministic results or on the occurrence of external events.

Hardware interrupts are external events that may occur randomly and have the potential to influence fingerprints. When an interrupt occurs, the CPU halts and loads an interrupt handler that is executed in kernel mode (ring 0 in x86). This handler terminates when the interrupt return instruction *IRET* is executed. This instruction switches back to user mode (ring 3 in x86) and restores the thread's state. This process updates the architectural state and therefore also modifies fingerprints. To ensure that a thread has always the same fingerprint on reexecution, the fingerprints have to be preserved on interrupts. This could be implemented by modifying the behavior of the interrupt entry and the IRET instruction. On interrupt entry the current fingerprint could be stored alongside the current program counter and stack pointer. Then it could be restored on execution of the IRET instruction. This mechanism would break backwards compatibility because most operating systems rely on the order and amount of values written to the stack on interrupt entry. Instead I decided to split the fingerprints into user mode and kernel mode fingerprints. Whenever instructions are executed in user mode, updates to the architectural state are compressed into the user mode fingerprints. Kernel fingerprints are updated whenever instructions are executed in kernel mode. This ensures that interrupt handlers or other code executed in kernel mode do not effect user mode fingerprints. Additionally the kernel fingerprints could be used to protect the kernel itself in future systems. However, then the kernel would need find a way to protect fingerprint non interrupts to ensure integrity.

So far I discussed how the fingerprints are computed and protected. But to be of any use, they also need to be accessible from within an application. The x86 ISA provides model-specific registers (MSRs) for the purpose of controlling hardware additions and optional features. MSRs are readable and writable by the special instructions *rdmsr* and *wrmsr*, respectively. These instructions are privileged and therefore must be executed by a kernel. I use four MSRs to store the current fingerprint values. The mapping between fingerprint values and MSR-addresses is shown in table 3.2. The kernel may modify fingerprints as it likes by using theses MSRs. In particular the kernel may store and restore fingerprints on task switches and provides an interface for user level applications.

Table 3.2: Mapping between fingerprints and MSR addresses

Fingerprint Value	MSR-Address
kernel instruction	0xC0020000
kernel data	0xC0020001
user instruction	0xC0020002
user data	0xC0020003

3.3 Implementing Fingerprinting in gem5

As discussed in section 2.6, gem5 provides CPU and memory back-end as fully exchangeable modules. First I needed to choose the CPU model on which my extensions are based. The timing of memory references is neither important for implementation nor for evaluation of fingerprinting. Therefore *TimingSimple* does not provide any advantage over the *Atomic-Simple* CPU. The *InOrder* CPU is out of question because it does not work in combination with the x86 ISA [Vai14]. *O3* implements a superscalar CPU with out-of-order execution. Such a design needs special care in the context of fingerprinting as the order of instruction results committed depends on different factors. Dealing with the problems of a super-scalar system is beyond the scope of this work. A discussion of the occurring problems and possible solutions can be found in the literature [Smo07]. To ease the design process, I based my work on the *AtomicSimple* CPU. The level of detail this model provides is sufficient for my design as my goal is to evaluate the idea of fingerprinting as a basis for error detection in SWIFT systems.

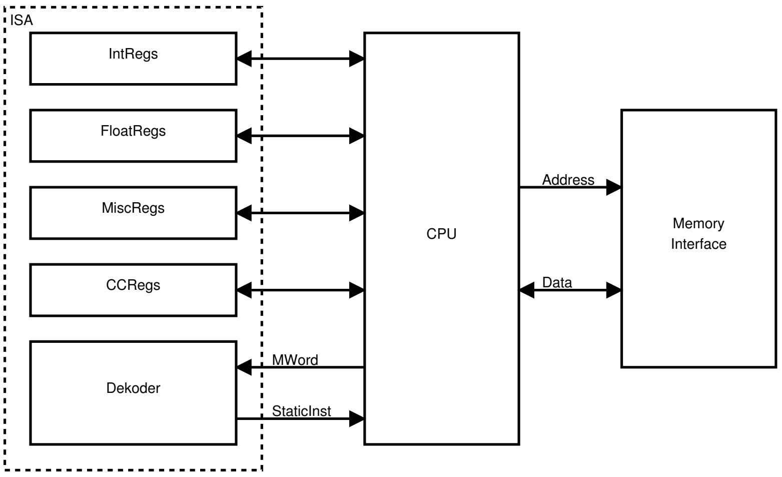


Figure 3.2: Functional principle of gem5.

Figure 3.2 gives an overview on how gem5 simulates a system. CPU and ISA are closely coupled but are implemented in separate modules. Although the CPU performs all the work, the CPU itself does not depend on any ISA specific structures. The CPU accesses everything that is ISA specific, for example registers or the decoder, through standard interfaces. The architectural state of a CPU is represented by a thread object. This object contains all architectural registers, the instruction pointer, and references to other simulation objects like decoder and memory ports. In gem5 architectural registers are divided into four groups: integer registers (*IntReg*), floating point registers (*FloatReg*), miscellaneous registers like control registers (*MiscReg*), and condition code registers (*CCReg*) that contain CPU flags.

The decoder is not part of the CPU as it is highly dependent on the the ISA. However, the decoding process is fully controlled by the CPU. The CPU is responsible for fetching new bytes and initiates the decoding process. Because instructions have no fixed size in x86 and instructions are not aligned to full words, the decoder may request additional fetches. When the CPU has fetched all instruction bytes, the decoder returns an object of type *StaticInst*. A static instruction contains all information that is needed by the CPU to execute the instruction. For instance it contains register operand indeces, memory addresses, and immediate values. A *StaticInst* does not provide any dynamic information like the current instruction pointer or register values. As x86 is a complex instruction set computer (CISC) architecture, most instructions are microcoded. If the current instruction is a microcoded instruction, the corresponding *StaticInst* is a macro-operation. Such a macro-operation consists of several micro-operations that are also represented by *StaticInst* objects. The whole process of interaction with the decoder is implemented in the method `preExecute` of the *AtomicSimple* CPU.

AtomicSimple CPU executes the current instruction simply by calling `execute` on the current *StaticInst* object. The CPU passes a reference to itself as an argument. This allows `execute` to use the methods the CPU provides for accessing memories and registers. *AtomicSimple* CPU implements the methods `setIntRegOperand` and `readIntRegOperand` for setting and reading integer registers, respectively. Similar methods exist for writing and reading *FloatRegs*, *MiscRegs*, and *CCRegs*. The term *Operand* in the method names implies that the registers are not indexed directly. Instead the index just refers to the ordering of operands within the instruction. The operand indeces are then translated into global register indeces based on the context that the current static instruction provides. These indirect register references are especially important for the support of windowed registers in SPARC [SPA92]. Besides methods for register access the CPU also provides methods for memory access. Namely these methods are `readMem` and `writeMem`.

I implemented fingerprinting in gem5 by introducing a new CPU model that I call *AtomicSignature*. This model is a modification of *AtomicSimple*. The general structure of the new model is shown in the UMD diagram in Figure 3.3. The classes are structured in the same way as for the simple CPU. The UML diagram only shows methods I referenced in the previous paragraphs, the ones I modified are printed bold. As `TimingSimpleCPU` and `AtomicSimpleCPU` share most of their functionality, it is also possible to implement a *TimingSignature* CPU based on my modifications.

As my goal is to detect errors within the CPU, I need to compress all its outputs into the data fingerprint. These outputs are new register values on register writes, new memory

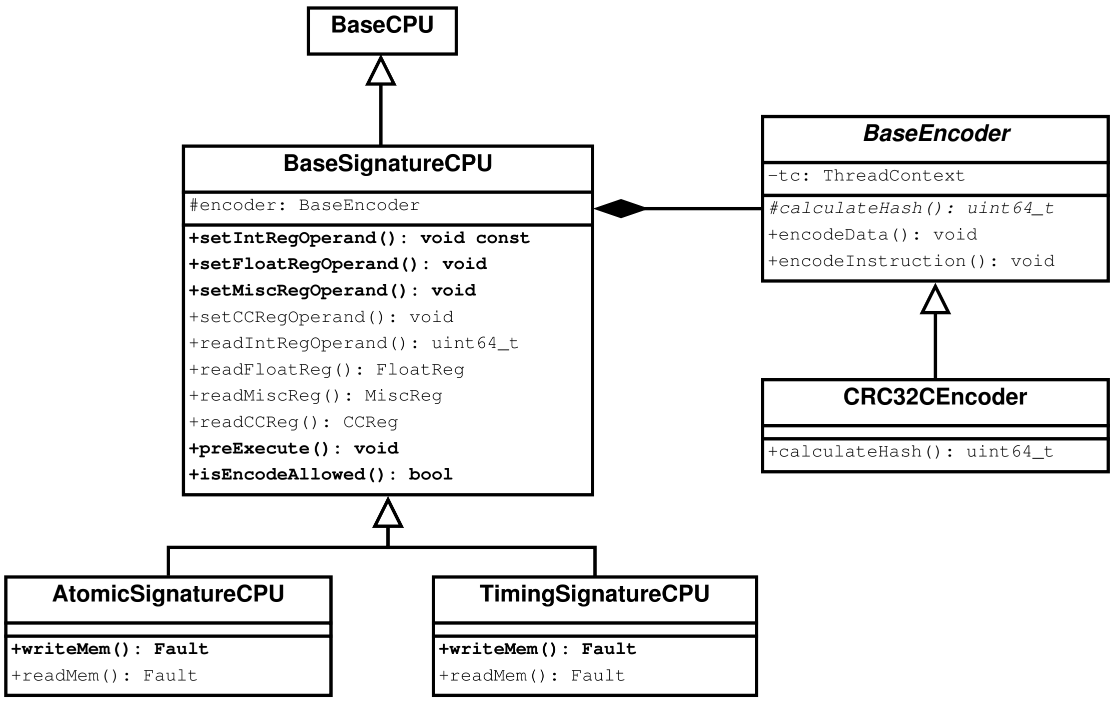


Figure 3.3: UML diagram showing the new `AtomicSignatureCPU`

values as well as virtual addresses on stores, and virtual addresses on loads. However, I can exclude virtual addresses on loads from fingerprint computation, because reading from memory does not change the architectural state. If a load is performed on a erroneous address and a wrong value is read, the error will become visible eventually, for instance when the value is written to a register. At this point I have to note that a load from an erroneous I/O-address may have side effects and therefore should be detected. However, detection based on fingerprints does not provide any benefit as the side-effect cannot be undone. Instead an additional solution that intercepts erroneous I/O-address accesses needs to be found. However, this problem is beyond the scope of this work and is therefore not considered in my design.

Computation of fingerprints is performed by a new encoder object to which the CPU refers. This object is an instance of the new abstract class `BaseEncoder` that is also shown in Figure 3.3. This new class is responsible for loading of the old fingerprint from the corresponding MSR, computation of the new fingerprint, and storing the new value in the MSR. As the method `calculateHash` is virtual, the base class can be implemented in different ways for supporting various hash algorithms. I implemented the class `CRC32CEncoder` to support CRC-32C. For CRC computation I used the Boost CRC library [Wal01].

This design allows for easy computation of fingerprints. Calculation of the data fingerprint is as easy as adding a call to `encodeData` in every method of the CPU that updates the architectural state. These methods are `writeMem` and the four methods that are used for setting registers. Unfortunately, it is not as simple as that, because there are some cases where the fingerprint should not be updated. As discussed in section 3.2, the fingerprints are protected on interrupts by splitting them into user mode and kernel mode interrupts. However, as x86 is a microcoded architecture, switches into the kernel and back to the user are not atomic. Instead switches are performed in several micro-operations. Therefore a micro-operation that is performed before the actual switch to kernel mode may modify the fingerprints. The same applies for operations executed after the switch back to user mode in the *IRET* instruction. My solution for this problem is to disable fingerprint computation when an interrupt entry is performed or the *IRET* instruction is executed. The necessary check is performed by the newly added method `isEncodeAllowed`. This way I can ensure the integrity of the fingerprints on interrupts. However, errors occurring during an interrupt entry or interrupt return are not covered directly by fingerprinting. But most of these errors should be detected in later execution as the control flow changed or an erroneous stack pointer was loaded.

As the x86 ISA is microcoded, it provides additional micro-architectural registers that are not visible on the instruction level. In gem5 the micro-machine uses these additional registers for internal computations but does not distinguish between registers that are part of the ISA and registers that are part of the micro-architecture. Both register sets are accessed using the same set of methods. Although computing fingerprints on the micro-architectural state would increase error coverage, I only included basic AMD64 registers into fingerprint computation. Therefore I added an index check before calling `encodeData` in each of the four methods for setting registers. I excluded the micro-architectural state from fingerprint computation, because the behavior of the micro-machine is not defined generally and could vary significantly in different x86 implementations. Especially gem5's micro-architecture caused several problems because some intermediate values, in particular some of the internally used flags, may differ on reexecution.

The instruction fingerprint is computed within the method `preExecute` that is responsible for managing the decoding process. Fingerprint computation is performed on each new instruction word that is fetched from memory. The instruction pointer is not part of fingerprint calculation, as an erroneous instruction pointer will become visible indirectly by fetching different values from memory. It would be nice to calculate the instruction fingerprint based on the decoder output, because this way also errors induced by the decoder would be covered. Unfortunately, this behavior is rather difficult to implement in gem5, as the decoder output is an abstract object and not a set of electrical signals. Errors induced by the decoder will become visible eventually, as register or memory values will change when a wrong instruction is executed.

3.4 Managing Fingerprint Registers in Fiasco.OC

So far I discussed how I implemented fingerprinting as a hardware extension. Now I will explain my changes to the software running on top of it. First I needed to modify Fiasco.OC so that it is aware of the fingerprint MSRs and associates them to threads. The kernel has to ensure that it restores the fingerprint MSRs whenever it resumes a task that was preempted before. Therefore the kernel needs to associate fingerprints with threads and store the fingerprints on each context switch. At this point it becomes clear why it is important to provide write access to fingerprint as discussed in Section 3.2.

To implement fingerprint preservation on context switches, I simply added two new attributes to the `Context` class in Fiasco.OC. They hold the values of data and instruction fingerprints. The kernel performs context switches by calling the method `switchin_context` on the current context object and passing a pointer to the new context as an argument. I extended this method, so that it simply stores the current fingerprints in the current context object before it reinitializes the fingerprints by loading the values from the new context object into the MSRs.

Besides storing and restoring fingerprints on context switches, the vCPU object also needs to be aware of fingerprints. When a vCPU raises an exception, it is migrated to another process that handles the exception. The user fingerprints need to be stored in the vCPU object on such a migration, so that the handler process can read and modify them. When the handler process finishes, it tells the kernel to resume the vCPU. On that switch the kernel has to load the fingerprints from the vCPU object and write them to the MSRs. This whole process is fundamental for error detection and recovery in Romain. The master process not only needs to be able to read and compare the fingerprints of its replicas, but it also needs to be able to restore the fingerprints on recovery.

I implemented this process by adding `fp_data` and `fp_inst` as new attributes to the structure representing the architectural state of a vCPU. A switch from a vCPU to its handler process is performed by the method `vcpu_pv_switch_to_kernel` that is part of the context object. I extended this method, so that it reads the user fingerprint MSRs and stores them within the vCPU structure. The switch back to the vCPU is performed by the method `vcpu_pv_switch_to_user`. Here I added code that reads the fingerprint values from the vCPU structure and writes them back into the MSRs.

3.5 Fingerprint Comparison in Romain

The modifications discussed in the previous section provide user level access to fingerprints. Now I have the basis for discussing fingerprint comparison in Romain but first I need to explain my changes in L4. The structure representing the architectural state of a vCPU is defined in the package *l4sys*. I extended this structure by two attributes for user fingerprints, as I did in the corresponding kernel structure. The package *libvcpu* wraps vCPU functions into an easy to use user level library. I had to add a method `arch_state` that returns a pointer to the structure defined in *l4sys* to allow for access to the fingerprints.

My modifications to Romain are rather straightforward. I need to ensure that each replica starts with the same initial fingerprint. Therefore I initialize the fingerprint values of the vCPU object with zero. Romain compares the replicas architectural states by summing up all integer registers and then comparing the resulting checksums. This checksum calculation is done in the method `csum_state` of the class `App_thread` that represents a replica. I simply replaced the summation of integer registers by a summation of the replica's data and instruction fingerprints. Furthermore I added code that restores the fingerprints on recovery, so that all replicas use the same initial fingerprints on resume. With these additions Romain is able to perform fingerprint comparison for error detection.

4 Evaluation

In this chapter I evaluate the design I discussed in Chapter 3. I analyze the error coverage of the fingerprint extension and of the system in its whole. Therefore I need to inject faults into the system and observe the resulting behavior. Based on these observations I can draw conclusions on the system's error coverage. As fault injection is a technique commonly used for evaluation of fault tolerant systems, several approaches have been developed. A survey on fault injection techniques can be found in the literature [ZAV03].

Fault injection techniques can be divided into software implemented and hardware implemented techniques. Software implemented techniques inject faults either by modifying the examined program at compile-time or by injecting faults through runtime services on certain events. Both techniques rely on modifications to the system and the examined application and therefore may have an influence on the system itself. As my design is based on the gem5 processor simulator, I have direct access to the architectural state. This allows for direct fault injection in the architectural state without modifying the system itself. As gem5 is not available as a back-end for fault injection frameworks like GOOFI [SBK10] and Fail* [Sch+12], I built my own solution as described in Section 4.1.

To get reliable results, it is important to base any conclusions on a sufficient amount of experiments. However, at this point it has to be stated that I only have limited resources. Therefore my experiments are based on small applications or on a small subset of larger applications.

4.1 Fault Injection in gem5

gem5 itself does not support fault injection and does not provide any interface that allows to modify the architectural state from outside the simulator. Therefore I extended gem5 to allow for fault injection. I added three command line options that can be used to flip bits in integer registers and memory cells. These options are shown and explained in Listing 4.1. With this extension the faults are injected on startup before the first instruction is executed. Fault injection at arbitrary points is possible in combination with gem5's checkpointing feature. In gem5 a checkpoint stores the whole state of the simulated machine and may be created at any time. gem5 is able to load a checkpoint and resume execution even when the configuration changed. Based on such a checkpoint, I can start new simulations with modified architectural states. I automated this process of checkpoint creation and fault injection in a set of Perl scripts to be able to perform a large amount of fault injections.

```
--fault-reg-idx=FAULT_REG_IDX      inject fault into the specified
                                   register
--fault-mem-addr=FAULT_MEM_ADDR    inject fault at the specified
                                   memory address
--fault-xor-mask=FAULT_XOR_MASK    apply this XOR-mask on the
                                   specified location
```

Listing 4.1: New command line options added for fault injection in gem5

4.2 Experiment 1: Full Register Test

I designed the first experiment to validate the error detection characteristics of the fingerprint extension itself. Therefor this first experiment does not use replication based on Romain. This experiment injects a fault into each bit of each of the 16 integer registers for each examined instruction. As an integer register has 64 bits, 1,024 fault injections need to be performed for each instruction. Because the total amount of injections that I am able to perform is limited, this experiment is based on a small application. The examined application is an implementation of the CRC64 algorithm. The source code of this application is shown in Listing 4.2.

gem5 provides functions that allow for control of the simulator from within an application running on that simulator. This functionality is implemented by the use of pseudo-operations, which are reserved by the ISA but trigger functions in gem5. `m5_checkpoint(0,0)` immediately creates a checkpoint. I use this checkpoint as a starting point for my experiment. To be able to print the current fingerprints at a certain point in a program, I added the pseudo instruction `m5_printsignatures`. By comparing the printed fingerprints, I can determine if a fault that was injected before `m5_printsignatures()` resulted in a fingerprint mismatch. Therefore the part of the program that is covered by my experiment is everything in between the lines 8 and 22 in Listing 4.2. For the argument "Bickerstaffe" a total of 1,274 instructions is covered. I excluded the call to `printf` from my examination as otherwise the total amount of instructions would easily exceed 10,000. However, `printf` can be excluded, because it is just used to display the result and is not part of the computation itself. Additionally, I compared the fingerprints again after `printf` to analyze differences in the distribution of results.

In total the experiment consists of 1,304,576 fault injections. I classified their outcomes in four categories by analyzing the program output and the fingerprint values. The results are visualized in Figure 4.1. As discussed in Section 2.2, an injected fault is a benign fault if no error is detected and the program output does not change. If fingerprints are compared before the call to `printf`, about 67% of all injected faults are benign faults. This result is

```c
#define CRC64MASK 0x42F0E1EBA9EA3693 // CRC-64-ECMA Bitmask

int main(int argc, char **argv) {
  // check if argument is valid

  m5_checkpoint(0,0); // create

  unsigned long crc64 = 0; // shift register
  char* str = argv[1];
  char c = *str;

  // simple implementation of CRC64
  while(c) {
    for (int i = 0; i < 8; i++) {
      if (((crc64 & (1 << 63)) ? 1 : 0) != ((c & (1 << i)) >> i))
          crc64 = (crc64 << 1) ^ CRC64MASK;
      else
          crc64 <<= 1;
    }
    str++;
    c = *str;
  }

  m5_printsignatures(); // (1)   print signatures before

  printf("0x%16lX\n", crc64); // the result is printed

  m5_printsignatures(); // (2)   and again afterwards

  m5_exit(0); // abort simulation

  return 0;
}
```

Listing 4.2: CRC64 implementation used for examination in Experiment 1 and 2

not surprising as the examined algorithm only works on a subset of the available registers and therefore many faulty bits are never read. About 24% of all injected faults resulted in a fingerprint mismatch and are therefore detected errors. 65% of the detected errors actually resulted in an erroneous program output. A smaller fraction of 6.5% of the injected faults resulted in runtime errors. Runtime errors are for instance page faults caused by a memory read from an erroneous address. This error class cannot be prevented by fingerprinting itself but in combination with Romain runtime errors are detected as Romain intercepts all events. The remaining 2.6% of all injected faults resulted in a much longer execution time and where intercepted by a timeout. A longer execution time may be caused by an erroneous iterator variable or by a incorrect jump into an infinite loop. On timeouts and runtime errors the `m5_printsignatures` instruction was not reached and therefore no conclusion on the error coverage can be made. Overall not a single SDC occurred. Every error that resulted in an erroneous program output was detected by the first fingerprint comparison.

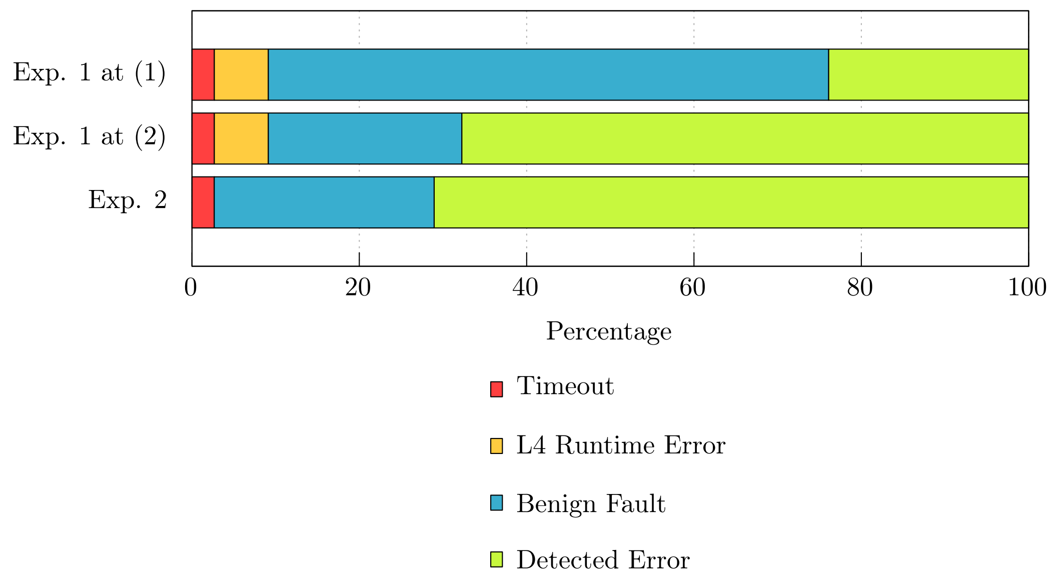


Figure 4.1: Results of Experiment 1 and 2 in comparison

For comparison I also evaluated fingerprints after the call to `printf`. At this point the distribution of results varies significantly. About 68% of the injected faults result in a fingerprint mismatch and are therefore detected errors. Only 23% are benign faults. This difference, compared to the first fingerprint comparison, occurs because `printf` is a more complex algorithm then my CRC64 implementation and therefore uses a larger register set. Faults that remained unseen before may become visible when `printf` reads erroneous registers. Of course `printf` does not load any registers it did not write before, but it may use registers only partially to store values shorter than 64 bits. Although the most significant bits are not used by the program, the CPU always works on whole registers. Therefore faults in unused register parts become part of the fingerprint.

The results of this experiment show that fingerprints can reliably detect errors that influence the program outcome. Additionally fingerprinting detects a large set of errors that do not influence the program outcome and therefore could be classified as benign faults. However, it is important to detect these errors as they have the potential to reside in the system and externalize eventually. As explained in section 2.4 such an undetected error has the potential to transfer to other replicas and may cause a failure of the whole system. That fingerprinting is able to detect this particular class of errors is shown in Experiment 4.

4.3 Experiment 2: Full Register Test with Romain

My second experiment is a repetition of Experiment 1, but this time the examined application runs on top of Romain. Thereby Romain uses fingerprint comparison for error detection. Therefore this experiment allows for estimation of the error coverage of the complete system. For comparability I used the same algorithm with the same argument as in Experiment 1 and also performed a full test. In this experiment I observed three different outcomes. For comparison the distribution of outcomes is also visualized in Figure 4.1. The biggest group are detected and recovered errors. This group encompasses over 71% of all injected faults. 26.3% of all injected faults are benign faults. The remaining 2.7% resulted in a timeout. In this experiment no runtime error occurred because Romain intercepts all page faults or other events that may occur and performs a state comparison. Not a single silent data corruption occurred during the experiment. The results show that Romain using fingerprint based state comparison reaches a similar error coverage as when fingerprints are compared manually. Overall a high error coverage is reached. Additionally the occurrence of timeouts shows that Romain is not able to detect errors that lead into a very long execution time. This is a general problem of Romain as state comparisons are only performed on events but the period of time between two events may be arbitrary long.

4.4 Experiment 3: Full Memory Test with Romain

The composition of Experiment 3 is similar to Experiment 2 but now faults are injected into memory. Because the CRC64 algorithm, that I used in the previous experiments, uses only a limited amount of memory references, it is not suitable for memory fault injection. Instead I examined the benchmark *qsort* from the MiBench suite [Gut+01]. This experiment injects a fault into each memory bit that *qsort* reads. Therefore I created a profile, based on gem5's instruction trace. This profile contains a list of all instructions that read from memory. To keep the total amount of fault injections manageable, I reduced the program input to a total of 20 strings with a maximum length of three characters. Additionally my examination is limited to the actual algorithm, similar as in Experiment 1 and 2. The parts of the program responsible for displaying the results and loading the input file are not part of my examination.

In total I performed 104,080 fault injections. With 98.8% almost all injected faults where detected by Romain. This behavior is not surprising because on most instructions the value read from memory is directly written to a register or another memory location. Therefore a value read from memory almost always becomes part of the data fingerprint. An exception is the return instruction RET. gem5's implementation of RET reads the return address from memory and then truncates the address as internally only 48-bit addresses are used. Therefore an fault injected into the return address may get masked and results in a benign fault. This happened for a total of 1,141 injected faults. 57 injected faults resulted in a crash of gem5. This crash is a side effect that occurs when an I/O-address is accessed and gem5 is not able map this address. Actually, a direct access to an I/O-address performed by the application should result in a page fault. However, as no page fault occurs this probably points to a security problem in the operating system and needs further investigation. Overall the experiment showed that fingerprinting is able to detect almost all faults injected into memory. Side effects caused by I/O-address accesses are a remaining problem and need to be resolved eventually.

4.5 Experiment 4: Random Fault Injection

The previous experiment showed the general potential of fingerprint based state comparison. My final experiment is designed to compare the error coverage of Romain without any modifications and Romain with fingerprint based state comparison. The benchmarks I used for this experiment are *qsort* and *bitcount* from the MiBench suite [Gut+01]. In this experiment I did no full test, instead I injected faults at randomly selected instructions. Therefore I created a profile containing a list of all reads from integer registers and from memory. For each injection I randomly selected a read and flipped a random bit in the selected register or at the selected memory address. By only injecting faults into registers or memory cells that are read, I ensure that the injected fault has the potential to lead into an error. Otherwise many simulations would result in a benign fault and therefore can't be used to compare error coverage. However, as I select the faults to be injected randomly from a list, this experiment does not reflect reality. In reality the distribution of faults is equally distributed over time and space. A value that is stored for a long time is more likely to become erroneous than a value that is stored for the duration of a few instructions. However, for the purpose of comparing error detection characteristics based on the same data set this experiment is sufficient.

For this experiment I used a slightly modified version of Romain. This version compares the state of its replicas based on fingerprinting and based on register state comparison, which Romain uses per default. Romain prints a message when one of the methods detected a state mismatch but does not perform recovery. By observing Romain's output I can determine which method detected an error and which one detected the error first.

In total I performed about 25,000 fault injections for each benchmark. For qsort 22.7% of all faults where injected into memory. For *bitcount* only 2.9% where injected into memory. This difference reflects the fact that *qsort* uses significantly more memory references than *bitcount*. The results of the experiment are visualized in Figure 4.2.

75.7% of all faults injected into the *qsort* benchmark are detected as an error by fingerprint comparison. Register state comparison only detects 43.3% of all injected faults as errors. It is remarkable that Romain with register state comparison is not able to detect a total of 113 errors that effectively lead to an erroneous program output. This is possible because Romain

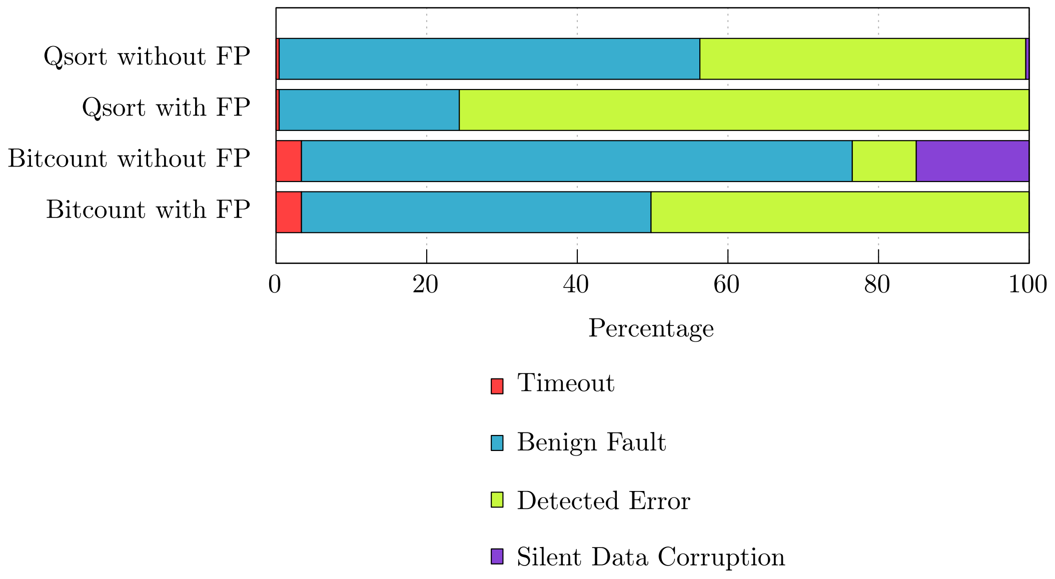


Figure 4.2: Results of Experiment 4 in comparison

in its current version only compares the register state and no universal thread control block (UTCB) contents. However, on a write to the terminal the UTCB contains the values that are printed and therefore an erroneous UTCB may influence the program outcome although the register state contains no errors. In contrast with fingerprint comparison every error that resulted in an erroneous program output was detected. In addition to the discussed results a total of 102 timeouts occurred.

Fault injection based on the *bitcount* benchmark shows different results. Again fingerprint comparison detects every error that results in an erroneous program output. In total 50.2% of all injected faults are errors detected by fingerprint comparison. Register state comparison misses a large set of faults that lead to silent data corruptions. These are 15.0% of all faults. In contrast register state comparison only detects 8.5% of all injected faults as errors. Two injected faults led to a crash of gem5 because of accesses at erroneous I/O-addresses. 3.4% of all injected faults led to a timeout and are therefore not detected.

In addition to the results discussed so far, I measured the number of state comparisons that are performed until an error is detected. Based on fingerprints every error was detected immediately on the next state comparison. For the *qsort* benchmark register state comparison detected only 95.0% of all detected errors immediately. For the *bitcount* benchmark Romain using register state comparison detected every detected error immediately. However, as only 8.5% of all faults where errors detected by register state comparison, I cannot draw a conclusion on error detection latency based on the *bitcount* benchmark.

Overall the experiment shows that Romain based on fingerprint comparison has a significantly higher error coverage than Romain based on register state comparison. In particular it was shown that fingerprinting is able to detect errors that lead to silent data corruptions when register state comparison is used. In addition I showed that fingerprinting reduces the error detection latency as every error is detected immediately on the next state comparison.

5 Conclusion And Outlook

This work discussed the design of a fingerprint extension for x86 based on the gem5 processor simulator. In Section 3.1 I started with preconsiderations that allow for the requirements of possible hardware implementations. This way I ensure that my extension has the potential to become part of future COTS systems. Additionally, a design with regard to real hardware implementation allows for realistic results. Reaching high error coverage in a simulation does not provide any benefit when the design is not feasible for hardware implementations in the near future. In Section 3.3 I described my implementation in gem5 and explained ISA specific problems that occurred during implementation. Although I implemented fingerprinting specifically for x86, the extension should be easily portable to other ISAs. As my design is based on gem's ISA independent CPU model, the general design of the fingerprint extension is also ISA independent. However, the method used for fingerprint access and solutions for ISA specific problems need to be implemented specifically for each ISA. In the final part of Chapter 3 I discussed my modifications to the software stack. This modifications provide an user level interface for fingerprint access and allow for fingerprint comparison in Romain.

In Chapter 4 I evaluated my design based on the Romain replication framework by a variety of fault injection experiments. This evaluation showed that the fingerprint extension is able to achieve a high error coverage. Romain detects significantly more errors when using fingerprints for state comparison as when using register state comparison. Overall not a single silent data corruption occurred. In all experiments errors where detected through a fingerprint mismatch immediately on the next state comparison issued by Romain. Therefore fingerprinting can be used to set a upper bound for error detection latency.

Despite an increased error coverage, evaluation also showed that there are remaining problems. As Romain only performs state comparison on interrupting events, errors are not detected when a replica is trapped in an infinite loop. However, in combination with a watchdog timer as proposed by Kriegel [Kri13] an upper bound for the error detection latency could be set. Furthermore the experiments showed that access to I/O-addresses may cause side-effects. In particular reads from undefined I/O-addresses lead to crashes in gem5. As a side-effect of an erroneous access to an I/O-addresses is not recoverable, the access needs to be intercepted immediately. Erroneous accesses to I/O-addresses are a general problem of Romain and need further investigation. However, fingerprinting alone is not able to prevent erroneous accesses to I/O-addresses as it is only able to detect errors after an operation on erroneous data was performed.

Although I designed the fingerprint extension with regard to real hardware implementations, it is difficult to estimate costs for an implementation in a modern processor. As my work is based on a simple model, it does not allow for modern architectures that use use super-scalar out-of-order pipelines. The complex design of modern processors is a challenge for fingerprinting as the order of commits of instruction results may vary on reexecution. To get a deeper understanding on the problems of a super-scalar design and the design costs for possible solutions, a future design could be based on gem5's *O3* model.

The use of an abstract model does also not allow for a precise estimation of hardware costs. Compared to hardware-only solutions I reduced the hardware cost drastically, as only a detection mechanism is needed. However, based on my model I cannot estimate the total cost. The complexity of a hardware implementation for the most part is determined by the complexity of the checksum algorithm. The amount of logic that is necessary for an implementation of CRC-32C is negligible compared to the amount of logic a processor needs [SS02]. However, more interesting are the timing characteristics of a hardware implementation and how they might effect the timing of the whole processor. To estimate the hardware costs more precisely, one could implement the fingerprint extension using a hardware description language (HDL). The resulting model could be simulated based on input generated by a processor simulator. Based on this HDL model one could determine the total amount of logic that is necessary for a hardware implementation and would be able to analyze the timing characteristics.

Despite Romain also other applications benefit from a fingerprint extension in hardware. Control-flow checking as proposed by Oh, Shirvani, and McCluskey [OSM02b] could use precalculated instruction fingerprints for signatures associated to nodes. By comparing precalculated and actual fingerprints at certain points in a program, errors in the control-flow can be detected. This technique is also interesting in the context of security. Fingerprinting implemented in hardware not only can detect soft errors but also can detect anomalies caused by control-flow attacks.

Bibliography

[Adv13] Advanced Micro Devices, Inc. (AMD). *AMD64 Architecture Programmers Manual*. May 2013. URL: `http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/` (visited on Apr. 16, 2014).

[Aus99] T.M. Austin. "DIVA: a reliable substrate for deep submicron microarchitecture design." In: *32nd Annual International Symposium on Microarchitecture, 1999. MICRO-32. Proceedings*. 1999, pp. 196–207. DOI: `10.1109/MICRO.1999.809458`.

[Avi+01] Algirdas Avizienis et al. *Fundamental Concepts of Dependability*. University of Newcastle upon Tyne, Computing Science, 2001. 20 pp.

[Bel05] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, 4141.

[Bin+11] Nathan Binkert et al. "The Gem5 Simulator." In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), 17. ISSN: 0163-5964. DOI: `10.1145/2024716.2024718`.

[Bor05] S. Borkar. "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation." In: *IEEE Micro* 25.6 (Nov. 2005), pp. 10–16. ISSN: 0272-1732. DOI: `10.1109/MM.2005.110`.

[Cap+14] Johna Caplan et al. "Trade-offs in Execution Signature Compression for Reliable Processor Systems." In: *DATE '14:Proceedings of the Conference on Design, Automation and Test in Europe*. DATE 14. 2014.

[CBH93] G. Castagnoli, S. Brauer, and M. Herrmann. "Optimization of cyclic redundancy-check codes with 24 and 32 parity bits." In: *IEEE Transactions on Communications* 41.6 (June 1993), pp. 883–892. ISSN: 0090-6778. DOI: `10.1109/26.231911`.

[DHE12] Björn Döbel, Hermann Härtig, and Michael Engel. "Operating System Support for Redundant Multithreading." In: *Proceedings of the Tenth ACM International Conference on Embedded Software*. EMSOFT '12. New York, USA: ACM, 2012, 8392. ISBN: 978-1-4503-1425-1. DOI: `10.1145/2380356.2380375`.

[Dre12] Dresden Operating Systems Group. *The Fiasco microkernel - Overview*. Nov. 30, 2012. URL: `http://os.inf.tu-dresden.de/fiasco/overview.html` (visited on Apr. 1, 2014).

[Dre14] Dresden Operating Systems Group. *L4Re – The L4 Runtime Environment*. Feb. 2, 2014. URL: http://os.inf.tu-dresden.de/L4Re/ (visited on Apr. 1, 2014).

[Ern+03] D. Ernst et al. "Razor: a low-power pipeline based on circuit-level timing speculation." In: *36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36. Proceedings*. Dec. 2003, pp. 7–18. DOI: 10.1109/MICRO.2003.1253179.

[Fle82] John G. Fletcher. "An Arithmetic Checksum for Serial Transmissions." In: *IEEE Transactions on Communications* 30.1 (1982), pp. 247–252. ISSN: 0090-6778. DOI: 10.1109/TCOM.1982.1095369.

[FSS09] Christof Fetzer, Ute Schiffel, and Martin Süsskraut. "AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware." In: *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*. SAFECOMP '09. Berlin, Heidelberg: Springer-Verlag, 2009, 283296. ISBN: 978-3-642-04467-0. DOI: 10.1007/978-3-642-04468-7_23.

[Gut+01] M. R. Guthaus et al. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite." In: *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, 314. ISBN: 0-7803-7315-4. DOI: 10.1109/WWC.2001.15.

[IEE02] IEEE Computer Society. *IEEE Sandard 802.3*. 2002.

[Kri13] Martin Kriegel. *Bounding Error Detection Latencies for Replicated Execution*. June 21, 2013. URL: http://os.inf.tu-dresden.de/papers_ps/kriegel-bak.pdf (visited on Apr. 23, 2014).

[Lie95] J. Liedtke. "On Micro-kernel Construction." In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. 00822. New York, NY, USA: ACM, 1995, 237250. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224075.

[LW09] Adam Lackorzynski and Alexander Warg. "Taming Subsystems: Capabilities As Universal Resource Access Control in L4." In: *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*. IIES '09. New York, NY, USA: ACM, 2009, 2530. ISBN: 978-1-60558-464-5. DOI: 10.1145/1519130.1519135.

[LWP10] Adam Lackorzynski, Alexander Warg, and Michael Peter. "Virtual Processors as Kernel Interface." In: *Twelfth Real-Time Linux Workshop 2010* (2010).

[Max06] Theresa Maxino. *Revisiting Fletcher and Adler Checksums*. 2006.

[McE81] Dennis McEvoy. "The Architecture of Tandem's NonStop System." In: *Proceedings of the ACM '81 Conference*. ACM '81. 00026. New York, NY, USA: ACM, 1981, 245. ISBN: 0-89791-049-4. DOI: 10.1145/800175.809886.

[MK09] T.C. Maxino and P.J. Koopman. “The Effectiveness of Checksums for Embedded Control Networks.” In: *IEEE Transactions on Dependable and Secure Computing* 6.1 (2009), pp. 59–72. ISSN: 1545-5971. DOI: 10.1109/TDSC.2007.70216.

[Muk08] Shubu Mukherjee. *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 9780080558325, 9780123695291.

[MW79] T.C. May and Murray H. Woods. “Alpha-particle-induced soft errors in dynamic memories.” In: *IEEE Transactions on Electron Devices* 26.1 (1979), pp. 2–9. ISSN: 0018-9383. DOI: 10.1109/T-ED.1979.19370.

[Nas10] S.R. Nassif. “The light at the end of the CMOS tunnel.” In: *2010 21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*. July 2010, pp. 4–9. DOI: 10.1109/ASAP.2010.5540756.

[OMM02] Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey. “ED4I: Error Detection by Diverse Data and Duplicated Instructions.” In: *IEEE Trans. Comput.* 51.2 (Feb. 2002), 180199. ISSN: 0018-9340. DOI: 10.1109/12.980007.

[OSM02a] N. Oh, P.P. Shirvani, and E.J. McCluskey. “Error detection by duplicated instructions in super-scalar processors.” In: *IEEE Transactions on Reliability* 51.1 (Mar. 2002), pp. 63–75. ISSN: 0018-9529. DOI: 10.1109/24.994913.

[OSM02b] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. “Control-flow checking by software signatures.” In: *IEEE TRANSACTIONS ON RELIABILITY* 51 (2002), 111122.

[Pat+11] A. Patel et al. “MARSS: A full system simulator for multicore x86 CPUs.” In: *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC). June 2011, pp. 1050–1055.

[PB61] W.W. Peterson and D.T. Brown. “Cyclic Codes for Error Detection.” In: *Proceedings of the IRE* 49.1 (1961), pp. 228–235. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1961.287814.

[Rei+05] G.A. Reis et al. “SWIFT: software implemented fault tolerance.” In: *International Symposium on Code Generation and Optimization, 2005. CGO 2005*. Mar. 2005, pp. 243–254. DOI: 10.1109/CGO.2005.34.

[RK06] Justin Ray and Philip Koopman. “Efficient High Hamming Distance CRCs for Embedded Networks.” In: *Proceedings of the International Conference on Dependable Systems and Networks*. DSN ’06. Washington, DC, USA: IEEE Computer Society, 2006, 312. ISBN: 0-7695-2607-1. DOI: 10.1109/DSN.2006.30.

[RM00] Steven K. Reinhardt and Shubhendu S. Mukherjee. “Transient Fault Detection via Simultaneous Multithreading.” In: *27th Annual International Symposium on Computer Architecture, 2000. Proceedings*. 00508. ACM Press, 2000, 2536.

[Rot99] E. Rotenberg. “AR-SMT: a microarchitectural approach to fault tolerance in microprocessors.” In: *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, 1999. Digest of Papers*. June 1999, pp. 84–91. DOI: 10.1109/FTCS.1999.781037.

[SBK10] D. Skarin, R. Barbosa, and J. Karlsson. “GOOFI-2: A tool for experimental dependability assessment.” In: *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. June 2010, pp. 557–562. DOI: 10.1109/DSN.2010.5544265.

[Sch+12] H. Schirmeier et al. “Fail*: Towards a versatile fault-injection experiment framework.” In: *ARCS Workshops (ARCS), 2012*. Feb. 2012, pp. 1–5.

[Shi+02] P. Shivakumar et al. “Modeling the effect of technology trends on the soft error rate of combinational logic.” In: *International Conference on Dependable Systems and Networks, 2002. Proceedings*. 2002, pp. 389–398. DOI: 10.1109/DSN.2002.1028924.

[Sle+99] T.J. Slegel et al. “IBM’s S/390 G5 microprocessor design.” In: *IEEE Micro* 19.2 (Mar. 1999), pp. 12–23. ISSN: 0272-1732. DOI: 10.1109/40.755464.

[Smo+04] J.C. Smolens et al. “Fingerprinting: bounding soft-error-detection latency and bandwidth.” In: *IEEE Micro* 24.6 (Nov. 2004), pp. 22–29. ISSN: 0272-1732. DOI: 10.1109/MM.2004.72.

[Smo07] Jared C. Smolens. *Fingerprinting: Hash-based Error Detection in Microprocessors*. ProQuest, 2007. 162 pp. ISBN: 9780549402909.

[Sog+01] E.S. Sogomonyan et al. “Early error detection in systems-on-chip for fault-tolerance and at-speed debugging.” In: *VLSI Test Symposium, 19th IEEE Proceedings on. VTS 2001*. 00003. 2001, pp. 184–189. DOI: 10.1109/VTS.2001.923437.

[SPA92] SPARC International, Inc. *The SPARC Architecture Manual*. 1992.

[SS02] Julian Satran and Dafna Sheinwald. *Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations*. Sept. 2002. URL: https://tools.ietf.org/html/rfc3385 (visited on Mar. 21, 2014).

[The13] The Bochs Project. *Bochs: The Open Source IA-32 Emulation Project*. May 26, 2013. URL: http://bochs.sourceforge.net/ (visited on Apr. 28, 2014).

[Vai14] Nilay Vaish. *Status Matrix - gem5*. Mar. 6, 2014. URL: http://www.m5sim.org/Status_Matrix (visited on Apr. 15, 2014).

[VPC02] T.N. Vijaykumar, I. Pomeranz, and K. Cheng. “Transient-fault recovery using simultaneous multithreading.” In: *29th Annual International Symposium on Computer Architecture, 2002. Proceedings*. 2002, pp. 87–98. DOI: 10.1109/ISCA.2002.1003565.

[Wal01] Daryle Walker. *Boost CRC Library - 1.55.0*. May 14, 2001. URL: http://www.boost.org/doc/libs/1_55_0/libs/crc/ (visited on Apr. 14, 2014).

[Wea+04] C. Weaver et al. “Techniques to reduce the soft error rate of a high-performance microprocessor.” In: *31st Annual International Symposium on Computer Architecture, 2004. Proceedings*. 00224. June 2004, pp. 264–275. DOI: 10.1109/ISCA.2004.1310780.

[ZAV03] Haissam Ziade, Rafic Ayoubi, and Raoul Velazco. *A Survey on Fault Injection Techniques*. 2003.

[Zie+96] J.F. Ziegler et al. “IBM experiments in soft fails in computer electronics (1978-1994).” In: *IBM Journal of Research and Development* 40.1 (1996), pp. 3–18. ISSN: 0018-8646. DOI: 10.1147/rd.401.0003.

[Zie96] J.F. Ziegler. “Terrestrial cosmic rays.” In: *IBM Journal of Research and Development* 40.1 (1996), pp. 19–39. ISSN: 0018-8646. DOI: 10.1147/rd.401.0019.

[ZL80] J.F. Ziegler and W. Lanford. “The effect of sea level cosmic rays on electronic devices.” In: *Solid-State Circuits Conference. Digest of Technical Papers. 1980 IEEE International*. Vol. XXIII. Feb. 1980, pp. 70–71. DOI: 10.1109/ISSCC.1980.1156060.