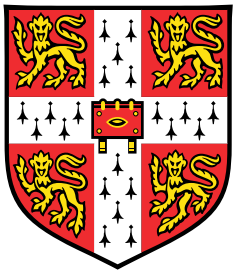
A Performance-Efficient and Practical Processor Error Recovery Framework



Jyothish Soman

Supervisor: Dr. Timothy Jones

Department of Computer Science and Technology
University of Cambridge

This dissertation is submitted for the degree of
Doctor of Philosophy

Wolfson College

July 2017

Abstract

Dissertation title: A Performance-Efficient and Practical Processor Error Recovery Framework

Author: Jyothish Soman

Continued reduction in the size of a transistor has affected the reliability of processors built using them. This is primarily due to factors such as inaccuracies while manufacturing, as well as non-ideal operating conditions, causing transistors to slow down consistently, eventually leading to permanent breakdown and erroneous operation of the processor. Permanent transistor breakdown, or faults, can occur at any point in time in the processor's lifetime. Errors are the discrepancies in the output of faulty circuits. This dissertation shows that the components containing faults can continue operating if the errors caused by them are within certain bounds. Further, the lifetime of a processor can be increased by adding supportive structures that start working once the processor develops these hard errors.

This dissertation has three major contributions, namely REPAIR, FaultSim and PreFix. REPAIR is a fault tolerant system with minimal changes to the processor design. It uses an external Instruction Re-execution Unit (IRU) to perform operations, which the faulty processor might have erroneously executed. Instructions that are found to use faulty hardware are then re-executed on the IRU. REPAIR shows that the performance overhead of such targeted re-execution is low for a limited number of faults.

FaultSim is a fast fault-simulator capable of simulating large circuits at the transistor level. It is developed in this dissertation to understand the effect of faults on different circuits. It performs digital logic based simulations, trading off analogue accuracy with speed, while still being able to support most fault models. A 32-bit addition takes under 15 micro-seconds, while simulating more than 1500 transistors. It can also be integrated into an architectural simulator, which added a performance overhead of 10

to 26 percent to a simulation. The results obtained show that single faults cause an error in an adder in less than 10 percent of the inputs.

PreFix brings together the fault models created using FaultSim and the design directions found using REPAIR. PreFix performs re-execution of instructions on a remote core, which pick up instructions to execute using a global instruction buffer. Error prediction and detection are used to reduce the number of re-executed instructions. PreFix has an area overhead of 3.5 percent in the setup used, and the performance overhead is within 5 percent of a fault-free case. This dissertation shows that faults in processors can be tolerated without explicitly switching off any component, and minimal redundancy is sufficient to achieve the same.

Declaration

It is hereby declared that the work shown in this work is my own work and includes only work done independently as part of this dissertation. This dissertation does not exceed the word limit of 60,000 words.

Acknowledgements

This thesis could not have come together without the guidance and support of many people, and especially many of my friends. I would like to thank my supervisor Dr Timothy Jones for being the source of inspiration and guidance for this work. His focus on the larger goals while pursuing short-term goals has been a revelation. Wolfson College has been a great source of support for me, especially my tutor, Dr. Kevin Greenbank, who provided me with the necessary help whenever needed, and gave me opportunities that have proved quite educational.

I would like to dedicate this thesis to my beloved friend, Negar, who is no longer with us. You were a source of strength and support during my thesis. My wife cannot go without mention in this thesis for her unending patience throughout my PhD. Finally, I would like to dedicate this thesis to my parents. Without their constant and continuing sacrifices and dedication towards my education, I would not have reached this point in life.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Transistor manufacturing has been steadily improving over time. One benefit from this is the scaling down of the transistor size by 2× nearly every four years. Moore's law [1] provides a rough guideline for this scaling. Chip fabrication since the 90s has followed Moore's Law with an error margin of ±1 years. Until the current technology node of 14nm, despite the difficult manufacturing process, transistor scaling has just managed to stay true to the law.

This scaling down brings expectations of proportional high performance, power and area improvements for the designs. Dennard scaling [2] provided estimates for voltage, power, delay and area for a transistor as its dimensions scale. Dennard suggested that the per-transistor delay, current and voltage will scale in direct proportion to the transistor scaling. Given that $Power \propto Voltage \times Current$, power consumed per transistor, would scale at a quadratic rate compared to the size. As the number of transistors for a given area also increases quadratically, the power density and total power for a given chip dimensions would be constant. Dennard postulated that as the technology scales, the delay and voltage would reduce, but the power density would remain the same, leading to faster, smaller processors which are power efficient as well. However, leakage current of transistors has started gaining relevance lately, which was not considered by Dennard. Leakage current increases the power dissipation of each transistor. Voltage scaling has not been achieved in the current technology nodes, especially threshold voltage has not reduced proportionally, leading to a higher power density. As the power dissipation of the chip increases, the temperature increases as well. Warmer chips age faster [3] and such an aggresive ageing would lead to fault forming earlier.

The scaling down of transistor sizes have hence caused a renewed interest and focus on reliability. Reliability is also becoming an issue due to the molecular scale sizes of the transistors. Currently, the size of a transistor is close to 40 atoms in width [4]. As mentioned earlier, operating voltage of a transistor has not scaled at the same rate as transistor dimensions. So there is a higher electric field across narrower widths which greatly increases both the energy requirements and the possibility of leakage of charges caused by parasitic resistances. This leads to the eventual breakdown of the transistor.

Another factor affecting reliability is the variability in the properties of various transistors within a chip. This is caused by the inability of the production environments to guarantee consistency, especially across multiple chips and also within a single chip. This has made current and upcoming generations of processors very susceptible to the occurrence of faults. Given the lowering dimensions, increasing electric fields and variation within the already small dimensions, the chances of a transistor breaking are substantially increased. Due to such effects, a variety of faults are formed during operation of a system.

The slowing down of processor voltage has been accompanied by a slowing down of the frequency of the processors as well. Since 2004, processor frequencies have found a ceiling of 5 Ghz. Most processors available now have their peak frequency below 4 Ghz. This clear throttling of application speeds on current and future generations of processors has to led to a situation where the current generation of processors can achieve acceptable levels of performance across multiple generations of processors. For example, the single-threaded performance of a processor released in 2010 (Intel Core i7–2700k) [5] is within 10–15% performance margin of a processor released in 2015 (Intel Core i7–6700), and within 22% of a processor released in 2017 (Intel Core i7–7700k). Moore's law would have suggested a 10X performance difference between the processors. Similarly, the top two processors on the single thread performance list are the Core i7–7700 and Core i7–4790, both having comparable performance despite the fact that they were released 3 years apart.

Despite the issues with reducing performance gains, the situation also presents an opportunity for cost savings to the end customer. A fault tolerant processor has a longer lifetime, reducing the cost of ownership for the end user. Manufacturers have clear advantages in terms of increased fabrication output, allowing immediate financial gains. Hence, a processor with fault tolerance and a longer lifetime has clear benefits.

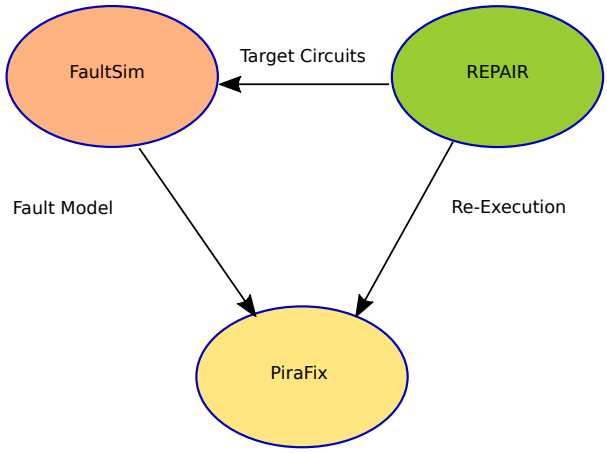


Fig. 1.1 Overview of dissertation structure

1.1 Hypothesis of this dissertation

This dissertation shows work done to handle hard faults in processors. Logic blocks internal to the Processor are the primary targets in this work. Processor peripherals or the memory system is not considered in this work. The methods are suited for hard faults but are useful for handling soft faults as well.

Hypothesis: *Given an out-of-order superscalar processor with hard faults in the logic components, a method can be developed, which can tolerate errors generated by these faults without switching off any faulty components.*

Definitions: In the above statement and rest of this dissertation, *faults* refer to actual hardware deterioration, *faulty component* refers to a circuit having a fault in it, and an *error* is a variation in the output of a faulty component from a fault-free case given the input. As can be inferred, the presence of a fault does not mean the occurrence of an error every time the faulty component is used.

1.2 Contributions of this dissertation

Given the hypothesis of this dissertation, this work presents REPAIR, FaultSim and PreFix as the three major contributions towards achieving it.

The first contribution is a fault-tolerant architectural design named REPAIR, where the faults are tolerated by re-executing instructions which use faulty components on a

remote accelerator. REPAIR allows instructions to be executed on the faulty component, and updates the result if the faults occur. The comparison is done post execution, the pipeline operations are halted pending the re-execution. On the availability of the results, the necessary registers are updated and the regular operation of the processor pipeline continues. REPAIR shows good performance in the presence of multiple faults especially in large structures. Further, it was seen that the performance of re-execution based fault tolerance is within acceptable levels.

FaultSim is the next contribution of this dissertation, which is a fast fault simulator capable of up to 300,000 32-bit addition operations per minute. FaultSim is also able to simulate numerous fault models. It is able to do so while providing transistor level simulations. FaultSim provides a simple interface for designing components, allowing easy simulation as well. Due to its simple interface, it can be easily integrated into architectural simulators.

PreFix is the final contribution of the dissertation; it has a conservative re-execution policy and the does not halt the pipeline. Both are clear improvements as compared to REPAIR. The performance is also significant, with median slowdown less than 5% compared to the fault-free case. Prediction, detection and permissive re-execution are used to handle faults. It is shown that even a conservative prediction of faults can reduce the overheads significantly. An interesting observation of the above mentioned fault tolerance systems is that, for multicores, on specific benchmark combinations, the performance of a fault tolerant system is better than a fault-free system.

This dissertation brings together three methods, which are inter-related. Figure 1.1 presents an overview of the relationship between the three. REPAIR is the foundational work, the results from which are used to create further sub-problems related to a fault acceptive methodology. Some of these were analysed using FaultSim and the results were used by PreFix to create a fault-tolerant system.

1.3 Dissertation framework

In Chapter 1, the different concepts relevant to later discussion are briefly introduced, such as faults and processor types. Then Chapter 3 presents REPAIR, which is a preliminary work on profile based re-execution of erroneous instructions. Results which clearly show the advantage of having an external re-execution based system is

shown. Next, FaultSim Chapter 4 is presented, which is a fast fault simulator. It is the fastest Gem5 integrateable fault simulator available. Chapter 4 further presents the characterisation of faults and its effects on the output for a set of circuits. Chapter 5 discusses PreFix, which explicitly uses both prediction and detection to speed up the handling of hard faults and handling it as if it is a soft fault. The results clearly show that the results are faster than switching off the component. Chapter 6 concludes the discussion and presents a short discussion on the attempted solutions for the hypothesis of this dissertation.

Chapter 2

Background

Handling faults in processors is the central theme of this dissertation and this chapter gives a brief introduction to them and to the various solutions for this problem available in the literature. The discussion on the solutions is detailed using the different processor architectures they target. The next section details the different type of faults that can be found in a processor.

2.1 Types of faults

Faults in this context are any permanent or temporary variation from the expected behaviour of the circuit, caused by either electrical or structural anomalies in the circuit. The presence of such non-ideal variations can lead to errors in the output of that component. Faults in the internals of a processor can affect the larger system it is part of, if the errors generated permeate through to the next stages, which can lead to cascading failures. Hence, the understanding of both the types of faults and their effect on the system is of interest. The types of faults that can be present in a processor include transient errors, permanent errors and inconsistently occurring permanent errors, also known as Goldilocks errors [4].

2.1.1 Transient faults

Transient faults, also called as soft faults are temporary and are triggered by an event external to the point of effect [6]. They are generally caused by radiation as well as system internal events such as coupling and supply noise. Transient faults are temporary

and do not cause any permanent damage to the system. Despite the short-term nature of the faults, the possibility of one being caused at any point of time (temporal spread) and at any place on the circuit (spatial spread) causes substantial difficulty in dealing with them. Due to the unpredictability of the temporal and spatial spread of the fault, fault-tolerance using historical data is not a practical solution. Predictive methods are hence not a possibility, transient faults are handled post detection. Once the errors are detected, they are handled by correcting any side effects it would have caused. In processor-based systems, errors manifest as incorrect value in the affected component or the buffer that gets the value from the component. Fixing these errors involves data redundancy or execution redundancy either spatially or temporally.

2.1.2 Hard faults

Hard faults, on the other hand are permanent. Hard faults are caused by fabrication issues and usage related deterioration. They affect the output of the system from the Time of Formation (ToF). ToF is either the time of fabrication of the chip or later in the lifetime of the processor. Fabrication time issues are caused by imperfections in the fabrication process. Typically, the faulty chip will have substantial faults and portions of it are switched off if possible to deliver a working processor. In the case of minor fabrication issues such as process variation, the produced chips have transistors and other circuit elements with non-uniform electrical properties. This leads to the different parts of the chip deteriorating at a different rate and some will age much faster than the rest of the chip. Along with lifetime degradation issues, this can create hot-spot points, which quickly age and hard faults form.

In contrast to the probabilistic nature of transient faults, hard faults are deterministic and are caused by degradation over time. With sufficient early detection methods, hard faults can be predicted ahead of time if needed.

Another issue with hard faults is irreversibility. The formed faults will persist across the remaining lifetime of the processor. Despite this, hard faults are not terminal to the processor. Formation of error would depend on the location of the fault, usage of the faulty component and the set of inputs to the faulty component. Hence, every input would not lead to an error.

Compared to transient faults, hard faults have received less attention, largely due to the perception that hard faults are a start of life and end of life event and the operation

time cost of hard faults is negligible. If the number of chips that are marked as not fit for purpose post-fabrication are considered, a better economic sense of the issue can be developed. For example, a study [7] presented that 40 to 60% of chips on average are fabricated faulty. The large numbers as such point to economic losses for both the chip manufacturers and the processor companies. Fabrication related faults could be handled by increasing the inherent hard fault tolerance in the processor designs.

2.1.3 Physical causes of hard faults

Hard faults are mainly caused by one of the following causes:

1. **Process variation and manufacturing defects:** During the manufacturing process of processor chips, the possibility of significant variations in the fabrication process can lead to non-uniform chips being manufactured. This causes variation in circuit parameters including channel length, threshold voltage and wire spacing [8]. Such a variation in one extreme means that a very large portion of chips would be immediately made useless, while a large portion would have weak frequency and voltage responses indicating a short lifetime, and these would typically fail quality checks. The eventually available processors would have assymetry w.r.t. electrical properties across the chip. Hence, the time to failure of various parts of the processor would be different from the onset itself.

2. **Negative Bias temprature instability (NBTI):** When PMOS transistors are placed under the effect of negative bias, causing gate oxide breakdown [9]. This causes an increase in the threshold voltage and decreases the drain current.

3. **Hot Carrier Injection (HCI):** When electrons gain sufficient energy, they inject themselves into the dielectric region of a transistor, causing damage to the oxide [9]. This can lead to the degradation of electric properties, and is quite similar to radiation damage that is experienced in deep-space.

4. **Time dependent dielectric breakdown (TDDB):** The application of a low electric field over time causes the gate oxide of a transistor to break down causing a conducting path to form. This causes the transistor be stuck at the same logic level [10].

5. **Electromigration:** The movement of material due to the movement of charged particles through a medium causes electromigration. This affects the channels through which electrons move. Though this is an important phenomenon, it is considered a lesser issue than NBTI and HCI [10].

NBTI, HCI and TDDB cause the transistor parameters such as mobility and threshold voltage to change. This effectively increases the circuit level parameters, especially the delay and increases the threshold voltage. The increase in delay reduces the maximum frequency of operation. To keep an aged transistor active, the voltage would also need to be increased, or the frequency needs to be reduced. Both have processor-wide effects as each processor will be within a frequency and voltage island [11], and hence any changes to voltage or frequency would be applicable to every element on that island. Hence, the frequency of the processor and its voltage requirements would be driven by such aged transistors.

2.1.4 Goldilocks Errors

An intermediate state between hard faults and soft faults happens when the operating conditions are not suitable for the circuit due to manufacturing issues [4]. The hardware would be seen as capable of accurate operation under certain constraints, but would fail otherwise. It is notable that the process of aging can also cause similar faults when the circuit begins to reach its end of life.

2.2 Comparison of Hard and Soft faults

Hard faults are caused by the wearout of the circuit due to usage, whereas soft faults are caused by environmental reasons. Assuming that the environmental events are random and are spread uniformly in space, the probability of a soft fault, $p_{sf} \propto Area$. The environmental conditions are independent of the operation of the processor, hence the probability of a soft fault is independent of the past state of the processor. The probability of error being caused by a fault is also independent of any previous faults. Errors may propagate through the system, but faults act as independent events. Soft faults are hence spatially and temporally independent. This is useful for n-modular redundancy schemes where single-event-upset [12] has a low probability of affecting a majority of the redundant components. Additionally, soft faults gained importance

from technology node of 90nm [13]. At this size, the energy of a 1.5MeV alpha particle is less than the inverter switching energy. An alpha particle strike would not change the value held in an inverter built at that technology node. All technology nodes smaller than 90nm have the possibility of the value changing due to an alpha particle strike.

Hard faults on the contrary are dependent on the time of operation of the processor, and hence cannot be treated as independent events. Using the NBTI and HCI equations, the deterioration of a component is $\propto f(\sqrt{T})$ [9], where T is the duration of usage of a component. Any two components with similar usages will develop faults at times close to one another. Additionally, such faults are permanent. Hard faults hence have temporal dependency. N-modular redundancy schemes that are active for the same duration of time would have similar chances of hard-faults manifesting in them. In a perfectly symmetrical system, where the initial state of the system (electrical and physical properties) are identical and the same application is run on the replicas with the same operating conditions would lead to the same flaws occurring in the replicas, and hence the faults would not cause the behaviour of the replicas to deviate. This would allow errors to pass through as the replicas would give out the same erroneous result. Hard faults are hence dependent on the usage of a given component, and if two identical components have run the same application, they will, with very high probability, form hard faults at matching locations.

Given that usage can be correlated in both space and time for any given pair of components, hard faults are neither spatially nor temporally independent. Hence, solutions for soft faults which are built on the premise of spatial and temporal independence of faults would not be an immediate fit for handling hard faults.

The direct consequence of this is on voting based schemes such as Triple Modular Redundancy (TMR). TMR relies on spatial and temporal independences, given that the errors caused by hard faults are dependent, active fault tolerance methods are not always ideal solutions for handling hard faults. A similar critique can be made for always active hardware redundancy schemes such as DIVA, where the possibility of fault formation increases over time quite similarly to the ones in the processor. Cold sparing is hence a more feasible mechanism for hard fault tolerance. REPAIR is one such strategy, where a cold spare in a separate location on the chip is provided that is used only when needed. Also, two processors running different workloads over time would develop different kind of faults, hence coupling processors can lead to better

fault tolerance solutions. One such method is explored in PreFix. REPAIR presents a way of using minimal spares, whereas PreFix focusses on reusing faulty processors. In summary, soft-fault tolerant solutions focus on reducing the window of space and time in which a transient fault can cause an error in the processor. On the contrary, hard fault tolerance focusses on providing viable spares or by reusing faulty components.

2.3 Overview of Fault tolerance strategies

Fault tolerance is not restricted to hardware-level solutions. It is a well studied problem in scales varying from large clusters, distributed systems and software, to processors. The error causing mechanisms and the semantics of their effect on the system are varied, but the fundamental methods of tolerating the faults have several common features. In this section, the various generic strategies devised to handle faults are discussed.

Fault-tolerance strategies can either be built for a hardware system or a software system. Hardware systems are either a cluster of multiple machines or a single machine with multiple or a single processor. Software systems have an analogy to the hardware systems, wherein, they are either a multi-machine software or a single-system software.

The solutions that will be discussed below target a system whose size or granularity would be either that of a multi-machine cluster, or of a single machine at its various software/hardware hierarchy of Operating system, application-level, compiler or multi-processor level or at a single processor level. Any solution which addresses all the above would be called a full-closure method. Though both hardware and software fault-tolerance systems take care of multiple points of failure such as network, power distribution network, storage etc. In this discussion, we would focus only on faults in the processor and the methods used to handle them. A detailed discussion on this is available in [14]. A short summary of the methods follow here:

- Redundancy: Redundancy has different implementations for hardware and software. Hardware redundancy [15, 16] is either Dual or Triple redundancy, the generalisation of which is N-number of component redundancy. Software redundancy on the other hand is replication of operation and verification of the results. The support for such can be from the user-level, compiler level or the Operating-system level [17].

- Recovery: Recovery is the process of dealing with an error once it occurs. There are two methods of doing so, namely checkpoint recovery or by using forward error correction. Checkpoints are either micro-checkpoints at the level of process registers, or at process level, where the checkpoints reflect the status of the process at a time. The largest checkpointing mechanism is a system level checkpoint, where the entire state of the system is stored. Micro-checkpoints are characteristic of processor-level fault tolerance systems, whereas process level checkpointing needs software-level interference at either one of OS, compiler or application level. System-level checkpointing [18] has the highest requirements in terms of complexity and storage requirements. Typically, it is present in systems where clear system snapshoting is possible. Though typically not used for running-systems due to various complexities, system-level checkpointing can be found in architectural simulator like Gem5 [19] or any database [20].

- Migration: Migration is the process of shifting operation of a running system from a faulty system to a currently functional system. For clusters, this means shifting the responsibilities of a machine to another one in the cluster, and in multi-processor systems, this means switching a task from one processor to another one. In either scenarios, the migration policy would be triggered on the event of a failure or the future possibility of one. For a multi-processor machine, a workload which would not accurately run on a faulty machine can be migrated to one in which it can run failure free; or a newly faulty processor is retired by migrating tasks away from it. Migration is a pre-emptive action on a running node. Hence, the node will be active while migration is taking place.

- Failure Masking: Failure-masking allows failures to happen and the system would then react to the event and re-configures to hide the effects of the failure. Failure masking is a reactive method, operating after a failure occurs. This method requires clear indications that the failure has occurred requiring error detection. One of the common application scenarios is when the failure is fatal and the failed machine cannot recover from it. In large scale systems, this is supported by repeating the tasks running on the failed machine, on a different machine. An example can be found in Hadoop clusters, where individual tasks are provided to a single machine and on failure, other machines take up tasks assigned to the failed machine and complete it. Compared to recovery, failure-masking assumes

a system that cannot be immediately brought back into full functionality without intervention.

2.4 Cost metrics of Fault Tolerance

The cost metrics of fault-tolerance are: reliability, area, power, performance and price. Reliability is measured by calculating the Mean time to failure (MTTF). MTTF is the expected time for a system to fail in the first instance. Once a failure happens, reliability is defined by Mean time between failures (MTBF). Given that this work focusses on handling hard faults, hence we focus on the MTBF of a processor. The overhead in terms of area covers both the area on chip and any additional resources that need to be added. For example, in system level checkpointing, the space needed to store the checkpoints can be significantly large and would likely be stored on a hard-disk drive. This adds an area overhead which may not be accounted for in a chip-level area computation. Power relates to the additional electric power required by the system to perform its functions. Power consumption increases the running cost of the system. Performance quantifies the increase in the time taken by a program running on the hardware. In this dissertation, it is measured as the percentile increase compared to a fault-free system. The monetary price of fault-tolerance consists of both design costs and run-time costs. Due to design costs being driven by market factors and run-time costs being driven by the operating environment, a discussion on costs will be beyond the scope of this thesis.

2.5 Processor architecture and Fault-tolerance

Fault-tolerance as discussed earlier presents a higher level view of the strategies that are usable for fault-tolerance. A discussion on the different fault-tolerance strategies based on the type of processor architecture is discussed in this section.

2.5.1 Fault tolerance for processors

Hard-fault tolerance strategies greatly vary depending on the different sections of the processor and for different processor types. From a performance perspective, the best suited method for fault-tolerance is the availability of spares. Such redundancies

can either be inter-core (spare is present in the core) or cross-core (spare is either shared across cores or is available on another core). Spares can be hot-spares (already active, and can be switched off when faulty) or cold spares (activated when an active components turn faulty). The recovery mechanism in the presence of faults depends on the redundancy available. Typically, once a fault is found, the faulty component is switched off and the redundancy is used. The mechanism of usage depends on the spare is intra-core or cross-core. PreFix presents a cross-core design, where the faulty component is not switched off. This is contrary to the fault tolerance schemes currently available.

For systems where spares are not possible, different mechanisms for handling faults are employed. One situation is when a processor is running an application and a component in it develops an error. In such a situation, just the presence of cold-spares (and the absence of an aggresive fault detection mechanism) would not protect the system from having errors forming in it. Aggresive fault detection is also critical in such cases. Safety-critical devices such as medical devices, autonomous vehicles, navigational and guidance systems cannot tolerate such errors. Another scenario is when the area overheads are critical, in such cases, explicit spares would increase the total area, this is true especially in embedded and low power systems. In either of these scenarios, the provision of explicit spares would not be either beneficial or sufficient.

For registers, the ease of substitution is high, as local provision of spare registers with multiplexors to redirect reads and writes allow cheap sparing. For functional units such as an ALU or FPU, the space taken by an individual unit is a non-trivial portion of an individual core. In situations where the space taken by spares is of concern, the ALU and the FPU are shared across processors if needed. This work takes a similar approach, where REPAIR has a common pool of resources shared across different processors. In PreFix, similarly, the resources of a different chip are shared between processors.

It can be noted from this discussion that fault detection is an essential part of fault-tolerance. Additionally, NBT and HCI induced ageing and degradation consistently reduces the performance of the processor. Eventually, the number of faults created due to ageing would be significantly higher than the maximum performance guaranteed by any fault-tolerance scheme. At this point, the processor would need to be retired. Hard fault handling methods focusses on providing some performance guarantees between the

first hard fault and the time of retirement. Specific solutions for hard fault-tolerance are discussed in the later chapters.

2.6 Summary

Faults of different types affect processors. The strategies to handle them are dependent on the application, processor design and costs involved. In the coming chapters, methods to deal with hard faults in out-of-order super-scalar processors are presented. These methods are also useful in handling faults in in-order processors. The effect of speculation and memory operations on these methods are also discussed.

Chapter 3

REPAIR: Accelerator based fault tolerance

This chapter presents REPAIR, a system that tackles faults in the processor core without the need for explicit spares. REPAIR is a minimal re-execution framework which will allow faulty components to do some amount of meaningful work without explicit duplication. REPAIR allows for remote re-execution using shared cold-spare components. Cold-sparing allows better protection from hard faults as the spare components have longer life left from the time of first fault as compared to hot spares [21]. Further, REPAIR also acts as a base from which key components are identified in the pipeline, faults in which can be catastrophic. The rest of the dissertation understands as well as mitigate effects of faults on the processor. Hence, REPAIR is the first building block, in tackling hard faults in hardware, of this dissertation.

Traditionally, methods to handle faults involved explicitly switching off components, or keeping them away from the critical aspects of the processor's functioning. REPAIR takes a different approach. Instead of swapping out faulty parts of the hardware, or keeping it running only to provide hints to other cores, the faulty cores are kept running and performing work. To do this the system is augmented with a small amount of logic containing only functional units and buffers that is termed here as an Instruction Re-execution Unit (IRU). Additionally, each core has a fault map, which store the fault state of all the tracked components (e.g., registers or ROB entries). It is either initialized by power-on self-test or periodic built-in self-test, to record faulty components within the core. REPAIR needs a dynamic periodic testing mechanism to

deal with evolving faults as REPAIR cannot perform fault detection. Also, faults are generated at an exponential rate in the later phases of a processor's lifecycle [22]. Each instruction is monitored as it traverses through the core's pipeline and whenever it touches a component marked as faulty, its execution is aborted, re-executed on the IRU, and the results passed back to the original core. This requires minimal modification to a standard out-of-order super-scalar pipeline and, additionally, imposes no performance penalty unless and until a hard error occurs. In this way multiple errors are tolerated in different components within each faulty core and the core generates correct results. This technique is named as REPAIR because it provides "Recovery from Errors in Processors by Allowing Instruction Re-execution".

REPAIR is evaluated as an addition to a single-core system, and as a shared resource within a multicore chip, showing that the expected performance for a single core is 0.81× peak performance and for multicore with four errors in every core is 0.68×.

The rest of this chapter is organized as follows. Section 3.2 discusses related work from the literature. Section 3.3 gives an overview of REPAIR and presents the IRU for re-executing instructions; how it can take a group of instructions whose execution faulted on the main core and re-execute them. Section 3.4 explains how the IRU is connected to the main processor cores, explaining our specific use case which integrates the IRU with an out-of-order super-scalar pipeline with checking for faults at both the dispatch and commit stages. Section 3.5 details our experimental setup before Section 3.6 presents the results from single core and multicore deployments of REPAIR.

3.1 Motivation

As discussed in Chapter 2, hard-fault tolerance is necessary for processors operating in circumstances where reliability is necessary. Also, there is a need to provide spares which would have a higher lifetime compared to the processor it is protecting. Cold-spares is one such method, where the lifetime of the spare is significantly higher than that of the processor it supports. Additionally, there exist use cases where providing extensive spares is not possible. In all such cases, the processor would need to have in-built fault-tolerance mechanisms which are light-weight in both area and power. Given that the cold spares are not active in a fault-free scenario, they do not incur any

faults during that phase. Hence, any additional steps to protect them from faults, such as increasing their geometry is needed. To protect REPAIR from fabrication faults, it would be useful to fabricate it in a higher geometry, but not necessary. REPAIR is designed to provide graceful end of life for a processor without the addition of substantial area or performance overheads.

3.2 Related work

Significant work has addressed hard faults and methods for continuing to use erroneous components despite their errors. The first issue to focus on is that of fault detection. DIVA [23] provides a solution for dynamic detection of errors by incorporating a functional checker at the commit stage of a superscalar pipeline. Results from computations on the main core are sent to the DIVA checker for comparison with the values generated there, and differences flagged up as an exception that can be handled by flushing and restarting. Although similar in some aspects to REPAIR, DIVA requires full execution of all instructions to detect and correct errors, whereas REPAIR only re-execute instructions that may have incurred an error, meaning there is no performance impact for correct instructions.

BlackJack [24] is an architectural scheme to detect hard errors by running redundant threads on a single SMT core. By providing duplicate execution of a program, application state can be compared at key points and differences discovered. In contrast Rescue [25] presents a micro-architecture that identifies and avoids hard errors. Using scan chains and automated test pattern generation, a Rescue core can be quickly locate faulty components which are then isolated and mapped out from the core, exploiting the natural redundancy available. Finally, Constantinides et al. [26] extend the ISA with new instructions that leverage the scan chains to probe any microarchitectural component, thereby isolating faulty structures that are usually unavailable to the software. These schemes do not directly approach fault tolerance, but REPAIR can use these approaches to periodically probe for errors and fill in the fault maps that are used to identify whenever an instruction requires re-execution.

Once errors have been detected they must be corrected, usually through the use of redundancy. Srinivasan et al. [15] add extra redundant resources for key microarchitectural structures which can be used in the event of a fault in original.

Graceful performance degradation makes use of the pre-existing spares in other logic to turn off parts of a structure that have errors, maintaining functionality at the expense of performance. Architectural core salvaging [27] uses the natural redundancy across cores to avoid structures with defects. When an error occurs, the executing thread can be migrated to a new core or swapped with a thread that will not make use of the buggy hardware. Architectural redundancy is used in StageNet [28] and StageWeb [16] by providing a network of pipeline stages. The network can be configured to work around faulty stages, creating logical cores that are distributed about the chip. Similarly Romanescu and Sorin [29] present a scheme to cannibalize cores at pipeline granularity, arranging cores into groups so that the cannibalized cores can donate spare pipeline stages to nearby cores. Cobra [30] has a distributed execution model where instructions are batched and executed based on resource availability. All these schemes rely on turning off parts of the core or chip that are faulty. REPAIR, in contrast, allows continued use of the faulty hardware but re-executes instructions that have touched these faulty resources.

Necromancer [31] uses faulty cores to enhance the performance of others by providing hints to another animator core. Although the animator has a lower specification to the other cores, by using hints from the faulty core its performance can be close to that of the fully functioning cores.

ReCycle is a method for tolerating process variation within the processor's pipeline [32] via cycle time stealing and the addition of donor pipeline stages. With these alterations the pipeline can be clocked with a period close to the average stage delay, rather than being dictated by the slowest stage. There has also been research to take advantage of micro-architectural redundancy to improve the performance average yield [33].

Other schemes have used software to avoid hard errors. A coprocessor was presented by Rajendiran et al. [34] to execute instructions that cannot be run correctly on the main core. Hard faults on the base processor are known by the compiler and appropriate calls are placed in the software to execute those instructions on the co-processor. Detouring [35] is a compiler method to translate software so that it doesn't use faulty components. ALU operations can be converted to logical counterparts and errors in the register file, bypass network and instruction cache can also be dealt with. Rosy [17] uses the operating system to execute software on unreliable cores. Thread relocation [36] uses a hypervisor-based system to handle errors by mapping software to appropriate

cores for execution. REPAIR, on the other hand, focuses on providing hard error tolerance in hardware only, avoiding the need to recompile software for a faulty core.

A performance analysis of different architectural and non-architectural arrays in the presence of errors across different technology nodes has been presented by Hardy et al. [37]. Their analytical model considers the performance impact of faulty cache cells being turned off, reducing capacity and leading to additional misses, and faulty predictor entries causing increased mis-predictions. They show that at 22nm the impact of SRAM cell failures are low, but that performance degrades noticeably at lower technology nodes. A discussion on performance-based reliability measures for computing systems has also been presented by Beaudry [38].

Finally recent work has considered approximate compute where errors are allowed to occur and tolerated rather than corrected [39–41]. Although this is suitable for a certain class of application, the majority of programs require exact computation and cannot survive errors in the underlying fabric. REPAIR is a general technique that provides error recovery for a processor, and is not targeted towards any class of applications.

3.3 REPAIR

The REPAIR architecture [1] allows a faulty core to continue correct execution by rerunning potentially-erroneous instructions on a new, specialized instruction re-execution unit (IRU). This unit provides external redundancy for instruction execution and can be shared by a number of cores in the system. Compared to existing methods, which rely on either forcing internal redundancy or using the faulty core to provide hints to the error-free cores, our approach focuses on getting useful work done on the faulty core, assisted by external logic. The design of IRU is discussed in this section; Section 3.4 describes how the instructions that need to be re-executed are identified.

3.3.1 Overview of the REPAIR architecture

The REPAIR architecture consists of standard cores, with minor extra logic to support REPAIR, and an IRU, which is used to rerun instructions that are faulty. It contains a mechanism to identify instructions that have, or may have, used erroneous hardware

[1] This work has been published at DFTS 2015 [42]

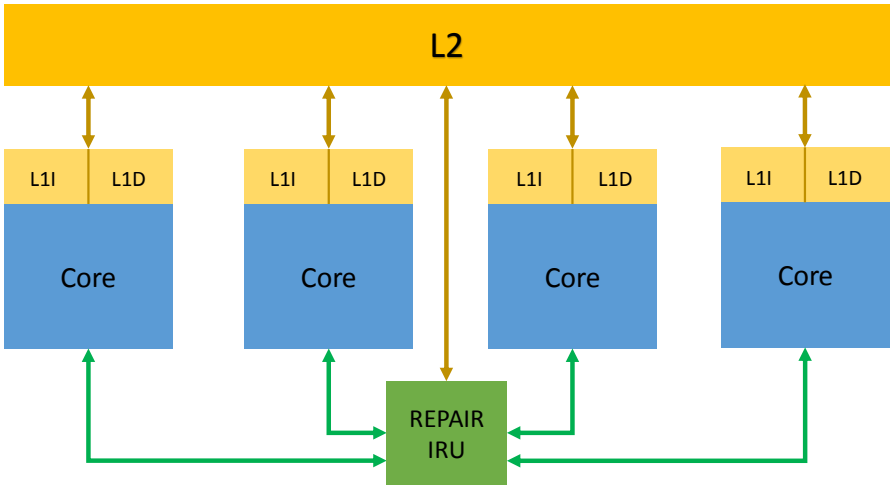


Fig. 3.1 REPAIR overview.

and transfer them to the IRU, writing the results back to the faulty core afterward. The IRU sits alongside the core that it supports, and in a multicore system can be shared between a number of cores. An overview is shown in Figure 3.1.

The REPAIR interface to the cores can be split into two parts: one for fault detection and the other for re-execution. The first part monitors instructions within the pipeline, checking each one to ensure that no logic with hard errors has been used while processing. The instruction resource usage are tracked at two points within a superscalar pipeline (dispatch and commit) to quickly catch instructions using faulty hardware with minimal intrusion into a core's internal logic. The second part re-executes an instruction that has used a faulty structure by squashing it and all subsequent instructions (if necessary), then passing it on to the IRU. Once the result is returned, execution continues with the first instruction after the fault.

It is important to note that REPAIR doesn't solely intercept instructions that definitely have an error, but in fact replays all instructions that *may* have an error. This keeps our monitoring circuitry simple, yet still guarantees that all faulty instructions will be caught.

3.3.2 The REPAIR Re-Execution unit

The instruction re-execution unit used in REPAIR is a simple circuit that executes one instruction, or a group of consecutive instructions from the same core, using operands and data read from that core. The IRU does not need all the components of a standard

processor; in fact it only requires buffers to hold instructions and their results, along with functional units to execute them. The basic IRU is shown in Figure 3.2 and consists of an interface to the main cores, input and output buffers, execution units, an operand manager, to deal with dependences between instructions, and a memory manager, to perform loads and stores.

The core interface is responsible for managing groups of instructions coming from the main cores. The IRU only executes instructions from a single core at any point of time; where more than one core wishes to use the IRU, the core interface arbitrates for access in a round-robin fashion.

Once a core has been chosen to use the IRU, it starts streaming the instructions and data to be operated on (i.e., register values) into the IRU where they are placed in the input buffer. Instructions wait here until execution units are available to execute them, and can start while further instructions continue to be streamed in. Instructions are executed from the input buffer in order and their results are placed in the output buffer, along with the destination architectural register ID. The output buffer keeps a mapping between architectural registers and the values that are generated within the IRU, coalescing multiple writes to a single register to reduce the data that needs to be sent back to the core. The execute unit of the IRU contains one copy of each type of functional unit appearing in a standard core. Although the IRU can execute any number of instructions from a core, REPAIR only uses it with sequences of micro-ops from a single macro-instruction at any one time. Accordingly, the IRU contains a simple pipeline and operand manager to handle such cases and avoid round-trips between the core and the IRU for each instruction within a group from the same core.

REPAIR also maintains precise exceptions by marking any instruction that causes an exception within the output buffer. Once this instruction has been transferred back to the core, exception handling deals with the issue as normal.

3.3.3 Source operands

Sitting alongside the execution units and buffers is the operand manager. Its job is to identify the location of the input operands for the execution units. The majority of data values are sent over with each instruction from the core. However, for a group of instructions there are two other sources. First is the bypass network between the functional units, which is included to provide back-to-back execution of dependent

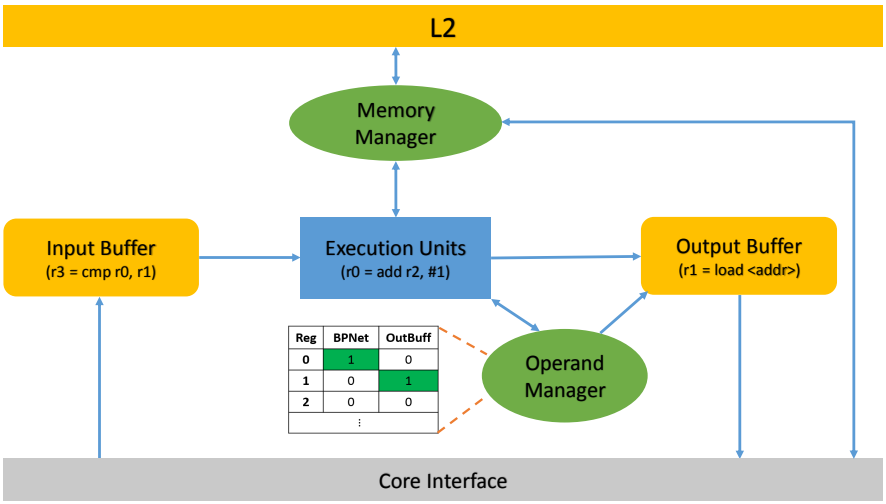


Fig. 3.2 Overview of the IRU.

instructions. Secondly, input values may have already been generated by an earlier instruction and reside within the IRU output buffer.

The operand manager handles dependences between a group of instructions executing on the IRU by scanning the source operands of the oldest instruction and providing the most recent value of each to the execution units. It first chooses to source values from the bypass network, then the output buffer, and finally the input buffer if the register has not yet been generated by the IRU.

Figure 3.2 shows an example of how the operand manager works. In this example, a load instruction has already executed and generated the value of register $r1$, whose result is in the output buffer and marked as such in the operand manager. In execution is an add instruction which will generate the value of $r0$, which is marked as being available for bypass in the operand manager. It reads $r2$ which has not been generated locally by the IRU (marked invalid within the operand manager), so it uses the value for $r2$ that was sent over by the core. Finally, there is a compare instruction in the input buffer, which will compare the values of $r0$ and $r1$ with each other. When the compare instruction moves to the execution units the operand manager will provide $r0$ from the bypass network and $r1$ from the output buffer, ignoring the values of $r0$ and $r1$ that were sent from the core.

3.3.4 Memory instructions

The memory manager of the IRU is responsible for handling memory operations that need re-execution. It performs address translation using the core interface as well as directly accessing the L2 cache either to read data or to write it.

The IRU does not contain a TLB, therefore virtual-to-physical address translation is handled by the original core's TLB through the core interface. Providing a TLB within the IRU would add significant complexity with little benefit. This complexity would arise from servicing TLB misses and ensuring that only the pages available to the core could be accessed by the IRU. Either the OS would have to be made aware of the IRU and the core it was currently executing instructions for, or the IRU would have to present itself to the OS as this core. In addition, since it is expected that the IRU is going to be used infrequently, there would be little temporal or spatial locality between the memory pages that would be accessed by the re-executed instructions.

After performing address translation, the IRU directly accesses the L2 cache. Again, due to the infrequent nature of instruction re-execution, significant locality between memory addresses accessed by the IRU would not be expected. Providing an L1 cache would also mean an extra cache for the coherence protocol to consider (if a shared L2) or would mean introducing coherence between the IRU L1 and the core's L1 (where there is a single IRU for each core and a private L2). Hence the IRU has a connection to the shared last-level cache. Our method makes a strong requirement about the cache-coherency protocol accepting the shared last-level cache as a valid level for storing modified data.

3.3.5 Summary

REPAIR works by identifying instructions that may have used faulty hardware within the main core and sending them to a re-execution unit to be executed again. This unit can be shared by multiple cores, but handles groups of instructions from a single core at a time. It contains buffers for holding instructions that are waiting to execute, as well as an operand manager for handling dependences between instructions. After execution of the whole group, results are written back to the core, the IRU is reset and normal execution can continue. The next section describes how REPAIR interfaces with the cores to actually identify faulty instructions and prepare them for re-execution.

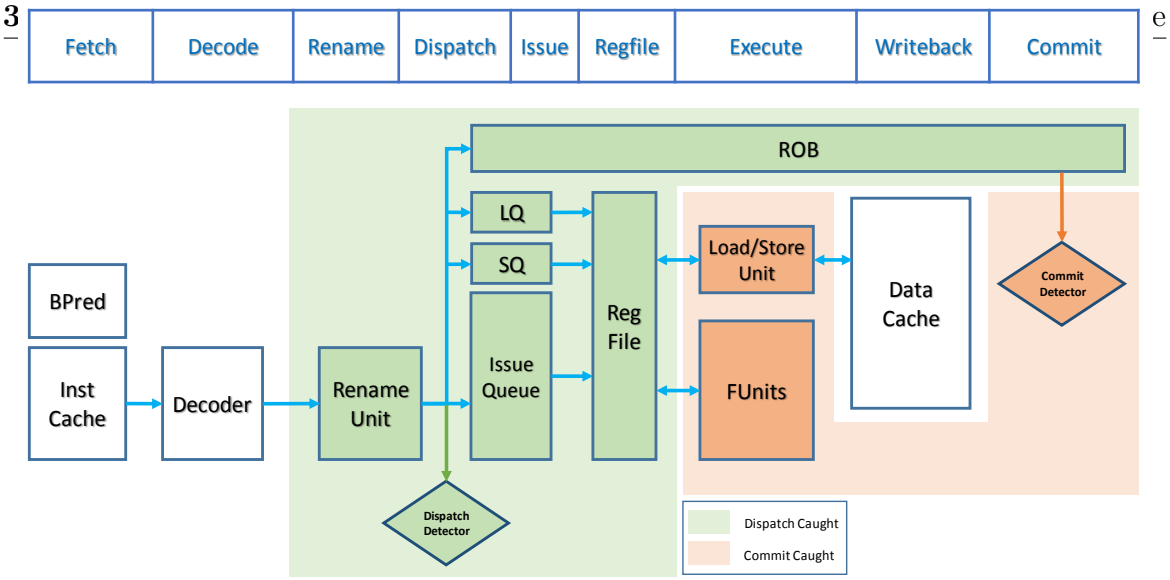


Fig. 3.3 Detection stages within a core.

3.4 Integration with standard cores

REPAIR is designed to require minimal integration into a standard out-of-order superscalar pipeline. It detects the use of faulty components at instruction dispatch and commit and takes advantage of existing branch mis-prediction logic to squash dependents of faulty instructions if necessary. This section first describes how instructions with potential errors are recognized, then discusses the core's operation in the presence of faults. Figure 3.3 gives an overview of how REPAIR is integrated into a superscalar pipeline.

3.4.1 Fault coverage

REPAIR protects the SRAM cells within the architectural arrays [2] used to provide out-of-order execution within a superscalar processor's pipeline, as shown in Figure 3.3. It deals with stuck-at faults in the rename table, register files, register scoreboard, re-order buffer, issue queue, load queue, store queue and functional units. Non-architectural arrays (e.g., branch prediction logic) where errors only affect performance and not correctness are not monitored as the aim is to preserve correctness rather than performance on hard errors. Other logic and structures within the core are assumed to be error-free. Therefore REPAIR expects valid, decoded instructions to be presented

[2] Arrays holding architectural state whose corruption could cause incorrect execution.

to the rename stage of the pipeline, from which it can take over, detecting faults at dispatch and commit.

Should there be errors in other parts of the core, a variety of alternate schemes can be used to continue correct execution, which are orthogonal to REPAIR. Caches nowadays are augmented with error detection or error correction codes [43]. Similarly, pipeline lanes can be turned off for errors in the fetch queue or decoders. In the extreme case, a core can simply be marked as faulty and powered down [44].

Faults can also occur in the hardware required for REPAIR. In this case, for correctness, it is simply assumed that the IRU does not work and cannot be used. Therefore a core can continue to run whilst it is error-free, but as soon as it develops a single error it must be marked as faulty. However, for the rest of this chapter, it is assumed that each core is fault-free in all the logic and circuits that are not covered by REPAIR and that there are no faults in any of REPAIR's hardware.

3.4.2 Identifying faulty instructions

REPAIR identifies instructions that have used faulty processor structures through the use of fault maps at the dispatch and commit pipeline stages. The fault maps indicate the processor structures that have been detected to have hard errors within them and REPAIR compares this information with resources actually used. The fault maps themselves are periodically populated using built-in self-test [26]. As the faults increase with time, the frequency of testing for faults has to increase accordingly. BIST incurs performance loss for the system, and if the frequency of testing is significantly large, then the performance of the system would suffer. It is recommended to decrease the clock-frequency if the total performance of REPAIR lags below the frequency scaled value, and refill the fault maps accordingly.

A fault map is a simple array of bits for each structure where each bit represents the presence of fault in one entry of that component. For example, in our processor described in Table 3.1 there are 32 entries in the issue queue, therefore the fault map for the issue queue contains 32 bits, each of which represents the presence or absence of an error in the corresponding entry.

The detector keeps a fault map for each structure that REPAIR covers within the core, and at every clock cycle checks instructions at the dispatch and commit stages to see if they have used, or will use, any faulty components. If so, they are marked

for re-execution by the IRU. Since checking for errors at the dispatch and at commit stages are subtly different, their operations are described in the next two subsections.

3.4.3 Dispatch checking

At dispatch REPAIR checks for errors in the ROB, IQ, LSQ, register scoreboard, rename map and physical registers. Instructions are checked for faults as they are dispatched into the reorder buffer and instruction queues. If there is an error in any of the entries that they are being dispatched into, then the corresponding instruction is marked as faulty and requiring re-execution. However, at this point, there is a problem with the validity of the instruction's bits that have been written. Since REPAIR makes the error checks at dispatch and does not find out until the end of the cycle that the instruction is using faulty hardware, the instruction bits in the previous pipeline latches would have already been overwritten. Yet the instruction bits that have been written into the queues cannot be used, since they may be erroneous. To solve this, a small buffer is provided that is the same width as dispatch (three instructions in our core) within the detector, called the dispatch-fault buffer. As the instructions dispatch, a copy of the instructions is also written into this buffer to allow access to valid instruction bits in the event that an error is found.

Once an instruction has been identified as faulting, its entries in the reorder buffer and other queues (and those of any younger instructions that have dispatched in the same cycle) are annulled to avoid them being executed erroneously. It is then held in the detector's buffer until the pipeline has drained as all older instructions commit, when it can be sent to the re-execution unit to be executed correctly. During this time an older instruction may flush the pipeline due to a branch misprediction or an error caught at commit. In this situation the instructions held in the dispatch-fault buffer are flushed too, meaning that they will not be sent needlessly to the IRU.

Once the pipeline has drained, the faulting instruction reads the values of its source registers directly out of the physical register file. These are bundled up with the opcode and destination architectural register ID and sent over to the IRU. After results have returned, results are written back into the register file, the faulting instruction is annulled in the dispatch-fault buffer, and execution can continue with the following instruction. This may have been held in the rename stage or could also be in the dispatch-fault buffer, having dispatched at the same time as the faulty instruction. If

an instruction is broken into a series of micro-ops during decoding, then, for the case of dispatch checking, only the faulty micro-op needs sending to the IRU.

3.4.4 Commit checking

The other stage where errors are checked is at commit. This is done by holding back instructions from completing until it is known that they did not use faulty hardware during their execution. At commit, errors in the ALUs and ports into the register files are checked. Since the ALUs perform many operations, REPAIR distinguishes between errors that affect the whole ALU and those that only affect one operation. For example, an ALU that cannot perform a shift operation may still be able to add two numbers together without faults. On the other hand, errors affecting the multiplexers into and out of the ALU prevent any operation from safely being executed.

Detection at the commit stage is similar to that at dispatch in that the fault maps are checked against the resources the instruction used. However, when a fault is found, the instruction can be sent immediately to the re-execution unit, since it is already the oldest instruction in the pipeline. (Where several instructions are checked together and multiple errors found, the oldest is the only one to be sent for re-execution.) At the same time, the pipeline is flushed of all younger instructions and fetch paused until the results return from REPAIR.

In contrast to catching errors at dispatch, when a micro-op is detected that is faulty at commit, it cannot be send by itself to the IRU. This is because later micro-ops from the same macro-instruction may have already used incorrect data from it. However, re-starting execution from a micro-op in the middle of a macro-instruction would be complex and require significant intrusion into the decode engine. To avoid this, all the younger micro-ops from the same macro-instruction are simply sent as the erroneous micro-op to the IRU for re-execution. After they are complete, fetch can then start again with the next macro-instruction.

If a faulty instruction is found, REPAIR relies on the speculative instruction squashing mechanism to restore a previous state. Squashing of speculative instructions requires value history for a given register. One way of doing this is by providing a retirement register set, as found in Intel Corei7 [45]. Another method is to use the ROB for this. On detection of a fault, each instruction that is in the pipeline, and was added to the pipeline after the erroneous instruction is squashed; the results from the

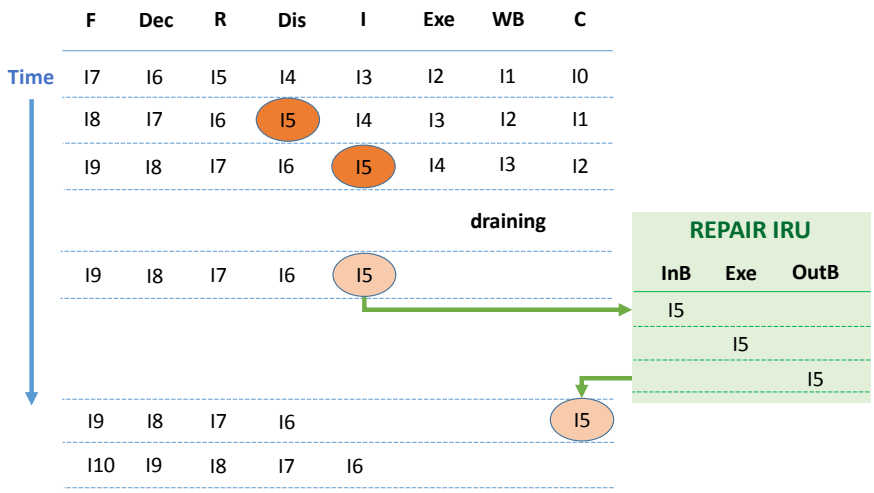


Fig. 3.4 Example rerun of an instruction using REPAIR.

re-execution of the erroneous instruction are then updated into the appropriate registers. The renamed registers for a given instruction are maintained. The micro-checkpointing in REPAIR is hence implicit as it reuses the instruction-squashing mechanisms.

3.4.5 Example instruction re-execution

An example of the complete flow for a dispatch-detected error is shown in Figure 3.4. Initially the core functions normally, however a fault is identified during dispatch on instruction $I5$. The core now waits for all older instructions in the pipeline to drain out, to be certain that none of them have an error and to ensure that all source operands for $I5$ are generated. During this point, $I5$ is held in the dispatch-fault buffer and later instructions ($I6$, $I7$, etc) are prevented from dispatching. After draining is complete, faulty instruction $I5$ and its source operands are sent to the REPAIR IRU for re-execution, which may take several cycles. In the IRU they are written into the input buffer, the instruction is re-executed and its result written to the output buffer. Since this is the only instruction requiring re-execution, the result is sent back to the core and written into the register file. The error on $I5$ is now cleared and it is removed from the pipeline since it has been executed and committed. Normal execution now resumes by dispatching $I6$.

3.4.6 Rename map errors

Errors in the rename map warrant special attention and care because they need to be both read from and written to by instructions that are being renamed. With all other structures (e.g., ROB, IQ, etc) a faulty entry can be completely unused, which is, in essence, what happens in REPAIR when an entry with an error is written into: the instruction is caught at dispatch and re-executed, and the next instruction is dispatched into the subsequent entry. Therefore no further reads or writes are made to this faulty entry until it is dispatched into again, and the process repeats.

Rename map entries, however, are different since a valid mapping of each architectural register to a physical register should exist, in order to correctly maintain state and to perform one of the most basic functions of the architecture—to be able to forward data between instructions through registers. Without being able to read register values and write into them, our core would be useless and have to be turned off.

This is avoided in two ways. First, it is required that a valid mapping can be found for each entry during power-on self-test so that the initial state of the rename map is stable. When there is a valid mapping, reads from the rename map will be successful and allow us to identify the correct physical register for a particular architectural register. Second, whenever an instruction writes into a faulty rename map entry (i.e., renames a faulty architectural register), it is caught at dispatch and re-executed. At the same time, the new mapping is undone and the old, valid mapping is restored. All younger instructions that have been renamed and use this architectural register as a source must also be patched up, since they will have the new mapping which now must be reverted back to the old. However, this operation is easily completed off the critical path while the instruction's results come back from the IRU.

3.4.7 Summary

The REPAIR architecture integrates IRU access into an out-of-order superscalar pipeline at the dispatch and commit stages. A fault at dispatch is held until older instructions are drained from the pipeline, then the instruction is sent for re-execution. At commit, all older instructions are flushed and fetch restarted with the next instruction once results have come back from the IRU. Using REPAIR, the architectural arrays used to provide out-of-order execution can be protected from multiple errors in a simple and effective manner.

Parameter	**C**onfiguration
Processor	1GHz, 3-wide, out-of-order,
ROB	40 entries
L/S Queues	16 / 16 entries
Issue Queue	32 entries
Registers	128 integer, 128 FP
ALUs	3 Int, 2 FP, 1 Mult/Div
Branch Pred.	Tournament with 2048 entry local, 8192 entry global, 2048 entry chooser, 2048 entry BTB and 16 entry RAD
L1 Caches	32kB, 2-way, 64B lines, 2-cycle hit
L2 Cache	1MB (single core) or 2MB (multicore), 8-way, 64B lines, 12-cycle hit
Main Memory	DDR-1600 11-11-11-28 @ 800MHz, typically 67-cycle access

Table 3.1 Experimental setup for cores and memory.

REPAIR is a general-purpose solution to cope with hard faults in a core's out-of-order execution resources. Although this can be feasibly dealt with, on a per-structure manner (e.g., by skipping ROB entries with known faults), this would become quite complex, especially in the presence of many faults. In addition it would require subtly different implementations for each structure that needed protection whereas with REPAIR a general purpose solution is presented, that easily copes with many faults in numerous different resources within a single design.

3.5 Experimental setup

REPAIR[3] is evaluated using the gem5 simulator [19] running the ARMv7-A ISA. Our cores have out-of-order superscalar pipelines, and resemble the Cortex-A15. Each core has a private 32KB L1 data cache and 32KB L1 instruction cache. There is a 1MB L2 cache for the single-core experiments or 2MB shared L2 for the multicore simulations. Table 3.1 details the processor core and memory characteristics, which is also the setup used in all the experiments in rest of the dissertation.

Our benchmarks are taken from the SPEC CPU2006 suite and compiled with gcc 4.6.3. All the benchmarks are used from this suite, apart from *dealII*, *lbm*,

[3]code available at https://github.com/jyosoman/docker-gem5-repair-pirafix

Name	Benchmarks
G1	perlbench, libquantum, gromacs, gcc
G2	omnetpp, leslie3d, gobmk, gamess
G3	milc, h264ref, gcc, cactusADM
G4	mcf, gromacs, gamess, bzip2
G5	libquantum, gobmk, calculix, bwaves
G6	hmmer, gcc, bzip2, zeusmp
G7	h264ref, gamess, bwaves, xalancbmk
G8	gromacs, calculix, astar, tonto
G9	gobmk, cactusADM, zeusmp, soplex
G10	gcc, bwaves, tonto, povray
G11	gamess, astar, soplex, perlbench
G12	calculix, zeusmp sjeng omnetpp
G13	cactusADM, xalancbmk, povray, namd
G14	bwaves, soplex, omnetpp, mcf
G15	astar, sjeng, namd, libquantum
G16	bzip2, tonto, perlbench, milc
G17	GemsFDTD, bzip2, xalancbmk, sjeng
G18	leslie3d, GemsFDTD, cactusADM, astar
G19	namd, hmmer, GemsFDTD, calculix
G20	povray, mcf, h264ref, GemsFDTD

Table 3.2 Benchmark groupings for 4-core workloads.

sphinx3 and *wrf* which did not compile correctly for our environment. For the 4-core experiments, 20 workloads are created, each consisting of four benchmarks to be run concurrently. These groups are shown in Table 3.2 and were created by first ordering the benchmarks alphabetically and then assigning them in turn to groups, ensuring each pair of benchmarks appears in at most one group.

Single-core benchmarks are run for a total of 250 million instructions, prior to this, the benchmark is checkpointed after 1 billion instructions and the cache and branch predictor are warmed for 100 million instructions. Multicore experiments run for 62.5 million instructions per core (250 million in total) after fast-forwarding and warming. The spread of instructions for each of the benchmarks in this window of instructions is shown in Figure 3.5. Each benchmark continues running until all cores have reached their target instruction count; stopping cores on reaching this would incorrectly reduce contention for shared resources on the remaining cores. However, the performance

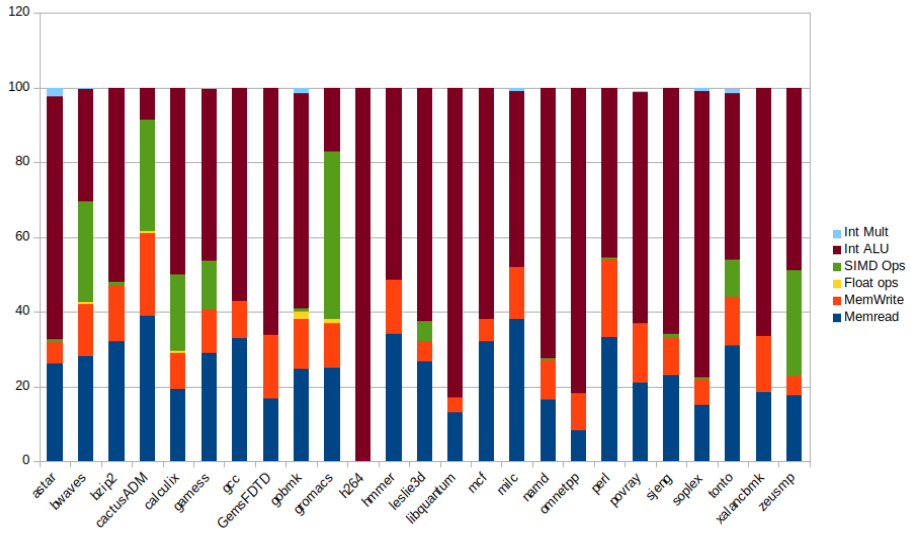


Fig. 3.5 The spread of instructions in the different benchmarks.

statistics for the target number of instructions are reported. Weighted speedup is used as the performance metric.

Our experiments require us to execute on cores that have errors. In our setup, there are approximately 500 elements which can have faults in them. The size of the design space (approximately 250,000 single-core configurations with 2 errors, 125m with 3 errors, etc.) meant that every point cannot be exhaustively simulated. Therefore, 50 single-core systems each with a single unique fault are randomly created, which represents 10% of the 500 element single-error space that we cover. Then single-core systems with 2 errors are created by randomly adding faults to the single-fault systems, and likewise for 3, 4 and 5 errors. The faulty multicore systems are also created in the same manner. Unless otherwise stated, graphs show the median performance across all systems.

3.6 Results

In this section, the impact of applying REPAIR to a single core is shown first, and then the results from sharing a REPAIR IRU between a cluster of four cores.

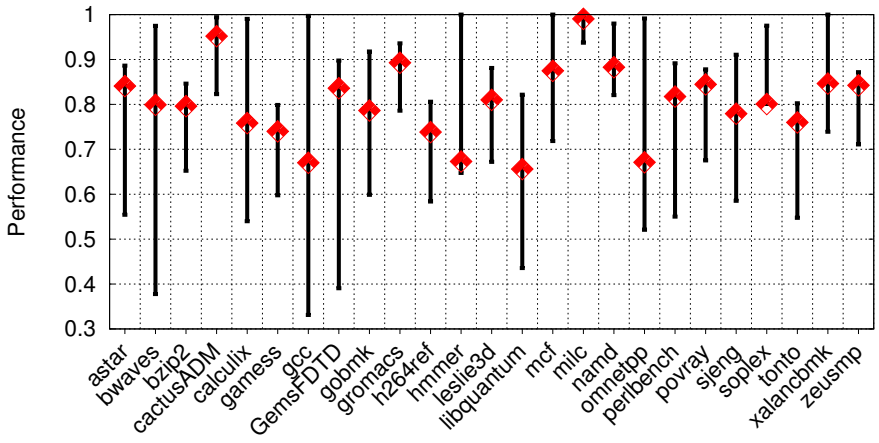


Fig. 3.6 Performance of REPAIR across single-core systems, each with a single error.

Factor	**C**orrelation
Memory writes	0.17
Memory reads	0.30
ALU operations	-0.53
Branching instructions	-0.25
Branch misprediction	-0.22
CPU running cycles	-0.21
CPU Idle cycles	0.24
No. of re-executions	-0.75

Table 3.3 Correlation between different application statistics and the performance of REPAIR

3.6.1 Single-core REPAIR

In the experiments discussed here, REPAIR efficiently tolerates single-bit errors in the core with an average performance of 0.81× the fault-free case. Figure 3.6 shows the results for our 50 single-core systems, each with a randomly-placed single-bit error. For each benchmark the maximum, median and minimum performance are shown across all sampled configurations. The results showed a performance ratio of 0.33× to 1.00× with an average performance of 0.81×, corresponding to a slowdown of around 23%. The maximum performance of 1.00× is achieved when there is an error in a component that is unused by a particular benchmark (e.g., a floating point register). The minimum occurs within *gcc* when there is an error in the rename map entry for architectural register 3, which is used as destination register for 13.5% of all instructions executed

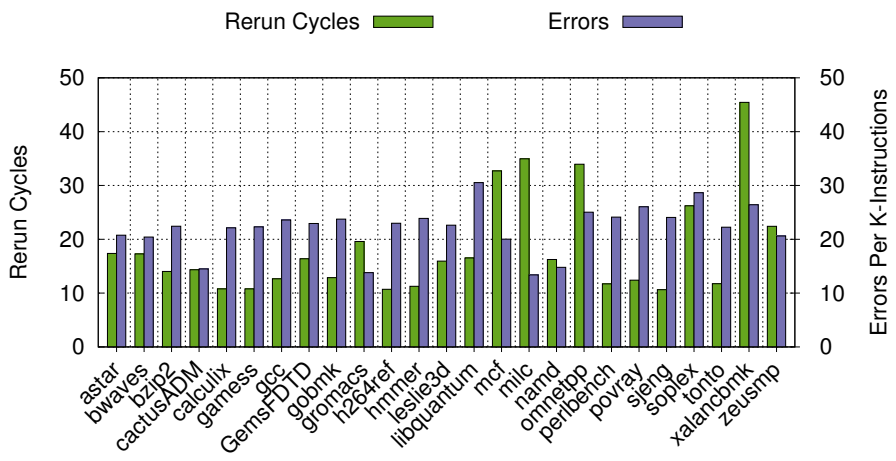


Fig. 3.7 Cycles taken for re-execution and number of errors per kilo-instruction.

in this application (within the 250 million instruction window that were monitored). This is analysed further later in this section.

The application *milc* is of interest in that it shows the least performance degradation in the presence of errors, with even the worst performance being only 0.94×. This is explained by *milc* experiencing significant pipeline stalls due to (L2) cache misses. In fact its baseline CPI is 2.60 and although in the worst case it suffers over 14 errors per kilo-instruction, its overall CPI increases to just 2.69. This equates to a fall from 0.384 instructions per clock cycle to 0.37 instructions per clock cycle. The time taken for re-execution is hidden by the time taken for the memory instructions to complete executing.

Figure 3.5 shows the spread of the various instruction in the sample that was benchmarked. It is notable that benchmarks which have a large memory requirement performs better with REPAIR, as the overhead of re-execution is hidden by the time taken by the memory instructions. Table 3.3 shows the correlation between the application statistics and the performance with single error on a single core. The number of re-executions has the highest effect on the performance. Memory operations improve the performance, as they hide the latency of the re-execution. This is also evident from the positive correlation between the CPU running cycles and performance and the negative correlation between CPU idle cycles and performance. From the correlation table, it is evident that any mechanism that slows down a error-free application has a positive correlation, and any factor that can lead to increased re-execution leads to a negative correlation with performance.

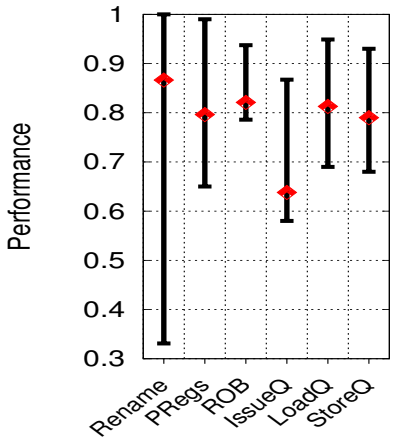
Analysis

Figure 3.7 further analyzes the behavior of our benchmarks in the presence of errors, considering the number of errors seen per kilo-instructions and the average number of cycles taken to perform a re-execution on the IRU (including pipeline draining / squashing and communication latency). All applications take, on average, at least 10 cycles to re-execute instructions, with the maximum being 45 cycles in *xalancbmk*. This benchmark also has a high number of errors per k-instructions (26.4) which would be expected to considerably impact its performance. However, as Figure 3.6 shows, its median performance is 0.85×. This is due to its high baseline CPI of 2.1, which rises to 2.8 in the worst case. As a comparison point, *tonto* has a low baseline CPI of 0.7, so it is more sensitive to the performance penalties that come from faulty hardware, even though it has a smaller average re-execution time and fewer errors that require the IRU.

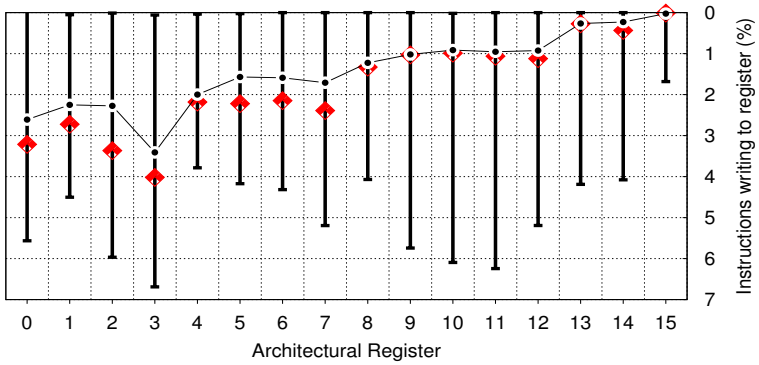
Differences between arrays

Errors in different architectural arrays yield a range of slowdowns, as shown in Figure 3.8a, where all the configurations of single architectural registers with a single error are included. Within the queues, ROB and physical registers, variability in performance comes from the differences between applications, and not from the position of the error, since each entry within these arrays is equally likely to be written to. Therefore errors are not simulated in every position within each of these structures. However, within the rename map the performance variability comes from the position of the error and application behavior, since benchmarks do not write to each architectural register equally. This means that the rename map has the highest variability, although the median performance is 0.87×. The issue queue has the worst median performance of 0.64× because it is small (only 32 entries) and is used by every instruction. The ROB, in contrast, is larger, so errors do not manifest themselves as often, whereas the load and store queues are only used by a fraction of instructions.

The larger variability in performance for faults in the rename map entries is explored further in Figure 3.8b. Faults in the rename map mean that a mapping to a new physical register cannot be made when the architectural register corresponding to the faulty entry is a destination operand. Figure 3.8b shows the performance corresponding to an error in each rename map entry / architectural register and the fraction of instructions



(a) Arrays



(b) Rename map

Fig. 3.8 Performance across different architectural arrays and within the rename map; also the fraction of instructions committed using each architectural register as a destination.

using each architectural register as a destination. It can be seen that errors in the first eight registers impact performance more significantly than the remainder. This is an artifact of the ARM architecture, in particular the Thumb-2 16-bit instructions which can only directly access these first eight registers (and are hence compiler-favored). Again the highest performance loss comes from a fault in register 3, which is due to a facet of our compiler—gcc's register allocation attempts to use register 3 before all others, meaning it is regularly assigned as a destination for temporary variables.

It is also clear that the median performance of each rename map entry closely follows the trend in the fraction of instructions writing to each architectural register. This is no surprise—if a certain register is used as a destination often then a fault in

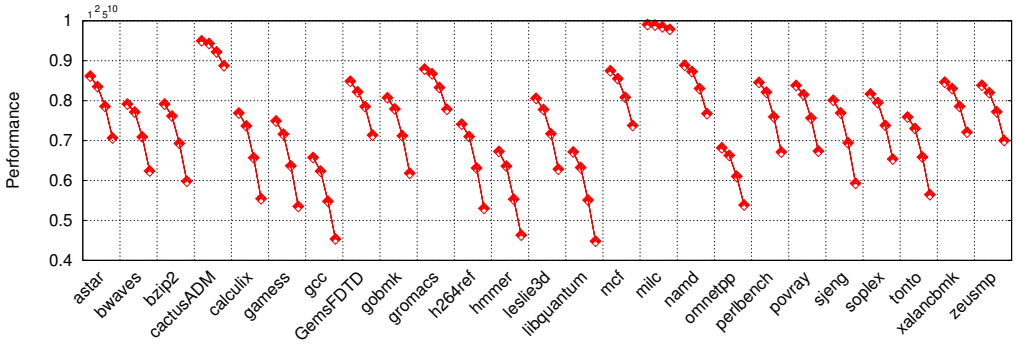


Fig. 3.9 Performance on a single core with the addition of extra communication cycles between the core and IRU.

that rename map entry will be encountered much more often than it would be were the register little used.

An interesting artifact isThe performance of REPAIR can be quantified by the following: the low overhead caused by the presence of errors in the program counter (r15), link register (r14) and stack pointer (r13). Procedure calls are relatively infrequent compared with the number of instructions executed within a function, meaning that the link register is little used. The stack pointer is generally only altered at the start and end of a procedure, again accounting for its low overhead. In contrast the program counter is updated on every cycle. However, it is important to note that it is not actually used as a destination register at all often (in only 0.2% of all instructions), meaning its overhead from faults in its rename map entry is low. (As an aside, an actual ARM implementation would not use this destination register as the actual PC, but another register closer to the fetch hardware.)

Communication latency

REPAIR is sensitive to the latency of communication between the cores and the IRU, but can tolerate high latencies without a severe impact on performance. The source of latencies are either IRUs placed at a significant distance from the originating processors. Another source of such latencies would be found in situations where the IRU and the processor are running on different frequencies, and hence a variation in the frequencies would cause delays in the response. In our experimental setup, we have modelled frequency difference between the IRU and the processor as latencies, as the Gem5 version in which REPAIR was implemented did not support frequency

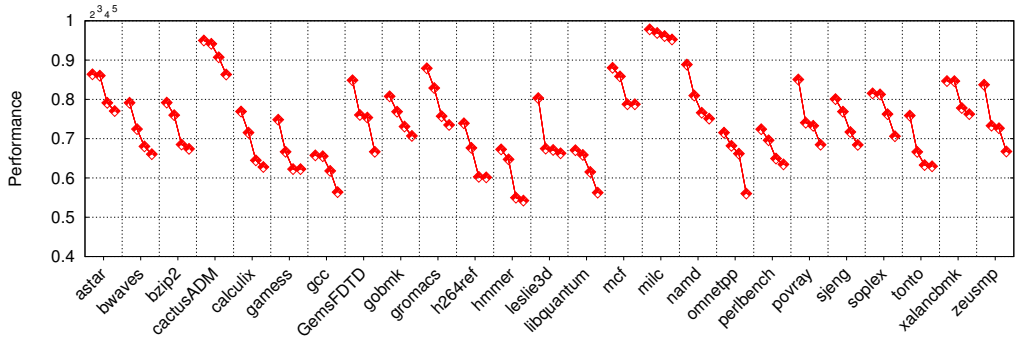


Fig. 3.10 Performance on a single core as the number of errors increases.

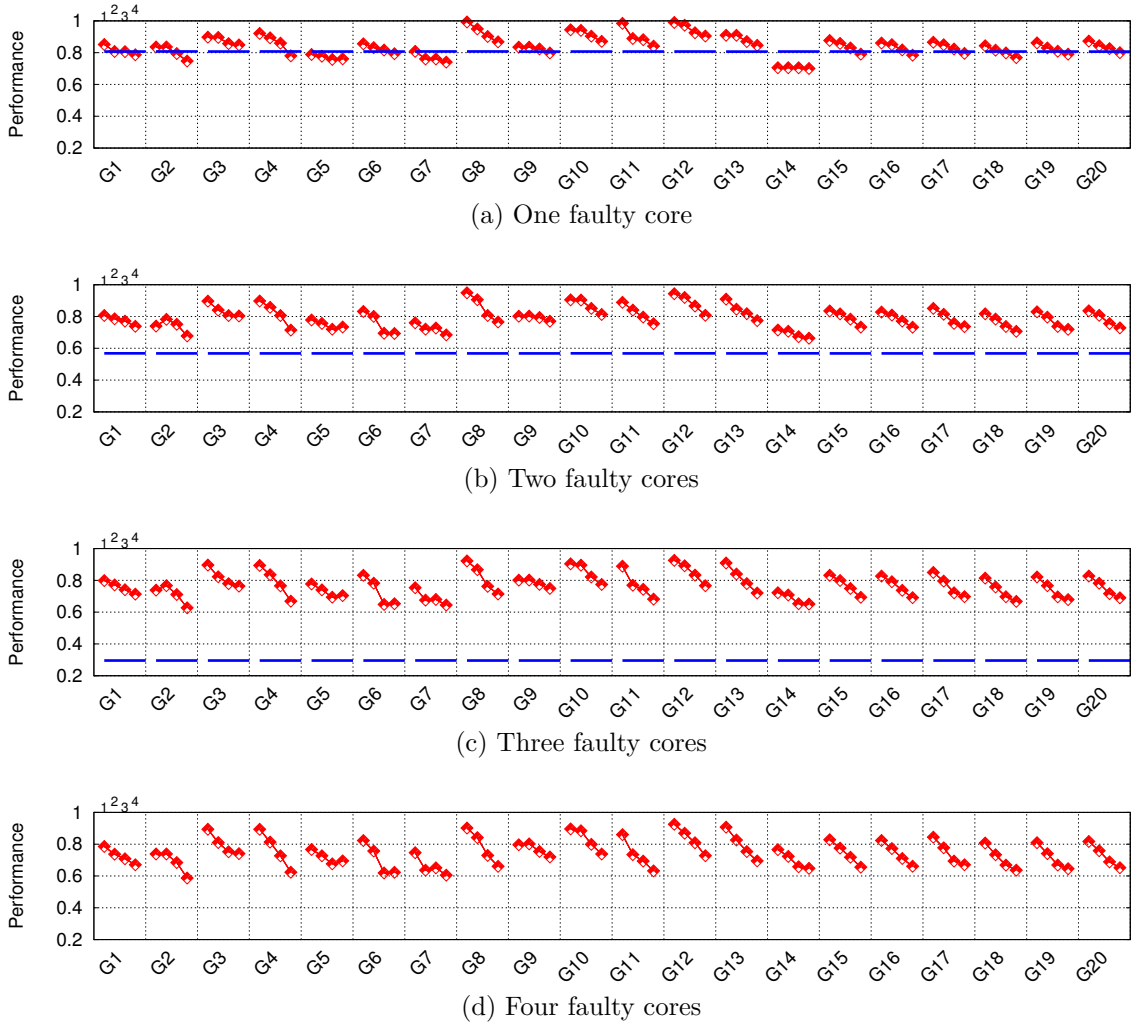
scaling trivially. Figure 3.9 shows the results when increasing latency between a standard core and the IRU through the addition of 1 to 10 extra cycles in each direction (baseline communication is 1 cycle in each direction). The increase in delay affects all benchmarks to some extent; as before *milc* is barely affected. An additional 10 cycle latency causes up to 56% slowdown but this is at the extreme end of the spectrum. An extra cycle latency, or at most 2, is achievable [46], and the latter still attains an average performance of 82% of the peak.

Number of errors

Figure 3.10 shows the median performance as the number of errors within the core grows up to 5, where average performance over all benchmarks is 0.70×, or a 43% slowdown. Some applications, such as *milc*, are barely affected by the increased work that REPAIR must perform to keep the core functioning correctly. Others incur a more substantial performance impact (e.g., *GemsFDTD* with a drop from 0.85× to 0.67×). Unfortunately, *GemsFDTD* has a very low baseline CPI and therefore experiences a drop in performance with each new error that is introduced, whereas other applications (e.g., *mcf* which has a high number of L2 misses) have a lower initial CPI and can better absorb the performance impact of the faults.

3.6.2 Multicore REPAIR

The impact of adding REPAIR to a cluster of four cores is now discussed. In this scenario all cores share a single IRU, meaning that when multiple cores are faulty and need to re-execute instructions, they must contend for the re-execution resource. For



(a) One faulty core

(b) Two faulty cores

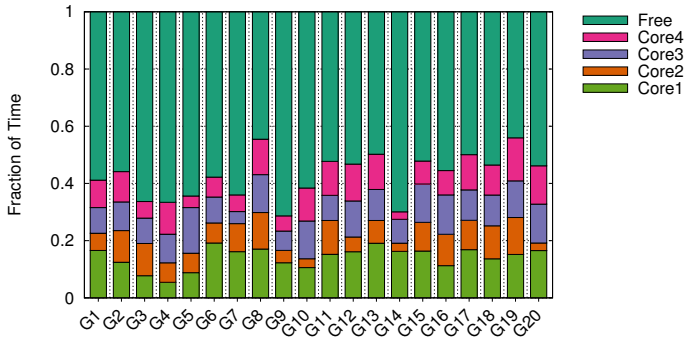
(c) Three faulty cores

(d) Four faulty cores

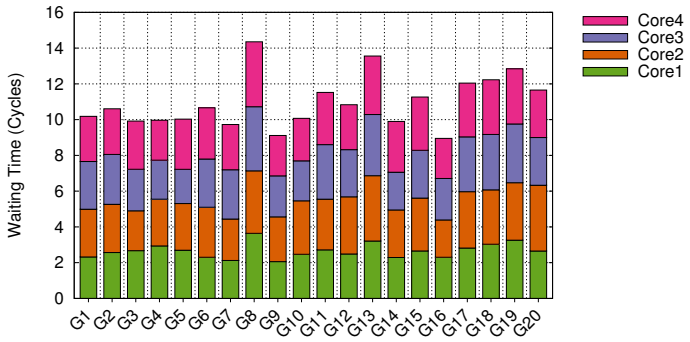
Fig. 3.11 Performance on a 4-core system as the number of errors per core increases. Also shown is a comparison scheduler.

comparison a time sliced scheduler which distributes benchmarks across $4-k$ cores is implemented, where k is the number of faulty cores. This simulates faulty cores being switched off when REPAIR is not present to continue correct execution. Our basic scheduler uses a scheduling quanta of 1 ms and does not actually move applications around but pauses each core when it is "off". This means that the L1 cache and branch predictor remain warm and there is no overhead to scheduling, favoring this comparison scheme over an actual implementation.

Figure 3.11 shows the results of using REPAIR on this system with 1–4 erroneous cores as the number of errors per core increases. Our comparison scheduler is also



(a) IRU usage



(b) Cycles waiting for IRU

Fig. 3.12 IRU usage and core waiting time on 4 cores, each with 4 errors.

shown, although it is missing in Figure 3.11d because with 4 faulty cores and no REPAIR, the whole system would have to be turned off!

As in the previous section, more errors leads to worse performance, but still REPAIR is able to keep the system functioning without significant slowdowns. With a single error in two cores, performance is, on average, just 0.86×, dropping to just 0.83× when these two cores have four errors. Were REPAIR not present, the two faulty cores would have to be turned off and our scheduling scheme used, which achieves an average of 0.59× performance (higher than the expected value of 0.5×). When all four cores contain a single error, average performance is 0.84×, dropping to 0.68× (or a slowdown of 47%) when each of the four cores contains four errors. This compares favorably to having no usable system since, without REPAIR, all four cores would have to be disabled.

There are some anomalies in the graphs, for example, the performance would be expected to decrease as the number of errors increases. This is not the case for a number of groups. For example *G2* when running on two faulty cores achieves a

performance of 0.74×, 0.78×, 0.75× and 0.68× when these cores have 1, 2, 3 and 4 errors respectively. The reason for the increase in performance of the two-error scenario is that the second core, in the baseline, is cache-unfriendly—it uses a large fraction of the L2 meaning that the other applications incur a greater number of misses than they would normally. When core 2 is faulty, the frequency that it makes L2 accesses is reduced and so the other applications can make better use of the shared L2, raising overall performance.

To explore the usage of the IRU further, Figure 3.12 analyzes the median point for each workload group in our worst-case scenario which is four cores each with four errors. Figure 3.12a shows that even with this number of faults in each core, the IRU is still unused for approximately 50% of the time. Variations between workload groups fall within the range of 29% usage (*G9*) to 56% usage (*G19*). Within each group the variations between each core's usage are due to the types of errors experienced by those cores. For example, the IRU usage for cores in *G14* are 16%, 3%, 8% and 3%. This is because the benchmarks on cores 1 and 3 experience approximately 3.5× the number of errors compared with cores 2 and 4. In addition, the baseline performance of cores 2 and 4 is lower, meaning that they both have fewer requests for IRU access to make, and that they have the opportunity to do it less frequently too.

Although the IRU is under-utilized, its services are required often enough that cores must wait to gain access to it, as Figure 3.12b shows. It plots the average number of cycles each core must wait before using the IRU (total cycles waiting divided by the number of IRU uses). In general, each core waits on average the same number of cycles as all others—no one core is favored over another—which is due to the round-robin scheduling policy adopted to arbitrate access to the IRU. Applications in workload *G8* wait the longest (3.6 cycles on average), whereas those in *G16* wait the least amount of time (2.3 cycles). This is because the applications in *G8* require many more re-executions than those in *G16*, thus increasing contention for the IRU and therefore the waiting time.

3.6.3 Hardware overhead

McPAT [47] was used to obtain the estimates of the area and power overheads of REPAIR. Each computational block of REPAIR was added into McPAT. Usage

statistics are taken from Gem5 and passed into McPat through the statistics file generated using Gem5.

In a 2-core configuration, it was seen that the area overhead is at 11% of the processor. For a 4-core processor, this reduces to 6% of the processor. The power overhead depends on the number of re-executions and the additional power is between 3 to 15% as compared to a non-faulty case[4].

3.7 Conclusions

REPAIR is a fault tolerant system capable of handling multiple faults in a single core and in a multicore system as well. The performance of the system is well within practical range and shows the capabilities of a fault tolerant system.

REPAIR shows that a faulty processor can provide practical performance by re-executing faulty instructions remotely. The performance is substantially reduced when there are faults in memory structures. The possibility of low cost (in terms of area) spares for memory structures allows REPAIR to focus on logic components where the cost of sparing is substantially higher. REPAIR shows that logical blocks can be a reasonable target for re-execution based fault-tolerance. Attempts to improve REPAIR is hence focussed on logic blocks. This requires understanding the nature of errors caused by faulty logical components. For this, we developed FaultSim, which is described in further details in the next chapter.

[4]Code available at: https://bitbucket.org/jyosoman/mcpat_pfix_repair

Chapter 4

FaultSim: An online fault-simulator

This chapter presents FaultSim, a fast online fault simulator capable of simulating most transistor and gate level fault models. FaultSim is designed to be integrated into architectural simulators such as Gem5 without adding considerable overhead in terms of performance. Additionally, FaultSim is a stand-alone fault-simulator which can be used to understand the effect of faults and how they manifest in the output of a given circuit.

Within the context of this thesis, FaultSim has been designed with the intention of understanding the frequency of errors caused by faults and hence being able to design fault tolerance solutions which are sensitive to the conversion rate of faults to errors, rather than just the location of the errors. FaultSim helps understand faults and its effects on the logic structure. As would be presented in the rest of this thesis, FaultSim quantifies fault-to-error ratio, as well as can be used in an online manner within an architectural simulator. Thus is essential in the context of this dissertation.

Within the wider context of fault simulations, there are two categories in which fault-simulators fall into, namely an offline analysis tool to generate test patterns that provide the maximum possible fault coverage; secondly, fault simulators that work as an online analysis tool that is used to understand the runtime behaviour of faults. FaultSim can be used in either methods, but is better suited for the latter use case. Within that use case, FaultSim is faster than the other available online fault simulators.

Given the significant development time required to implement full scale designs, a limited set of circuits is considered. The performance of the fault simulator on the Knowles family of adders [48] which encompasses many of the popular integer adder circuits such as the Kogge-Stone Adder, and Han-Carlson adders, is presented.

4.1 Related Work

Simulations vary based on the level of detail that is available in a simulation methodology. Given a fixed layer of abstraction (FLA), the simulations can be high level or low level. High-level simulation would use an abstraction above the FLA, providing a set of guarantees in terms of the simulation resolution and the results. Transaction level simulation (TLS) is a high level simulation method. Lower level simulation would work at the FLA or at a lower abstraction, and would model all the details of that level.

As discussed earlier, Fault Simulators can be categorised into two categories, namely offline and online fault-simulators. Offline fault-simulators are primarily built with the purpose of generating fault-testing, and online versions are for characterisation of faults. Offline testers do not have the advantage of being able to understand the implication of faults which online testers are capable of. It is also possible to analyse the evolution of faults with online fault simulators, but that is not covered in this thesis. In this discussion we focus on fault-characterising simulators, which may or may not operate in an online manner.

The purpose of online fault simulators is to understand the effect of faults relevant at a specific point of time. Currently, there is an ever-increasing focus on transistor ageing [49] and the resulting faults including delay faults and stuck at faults. Different methods of studying faults have been presented in literature. The most common method for fault simulation is a transistor level fault simulation. Other methods include probabilistic methods [50, 51] and critical path tracing [52].

Fault simulators either work at transistor level, gate level or are probabilistic simulators that estimate the effect of faults. Understanding faults at the same level they they occur in, is important to maintain the precision of the simulation. Swat-Sim [53] is one such simulator, that has shown results pertaining to the effect of errors on the application. Swat-Sim shows broad results from errors injected in the ALU, Decoder and address generator respectively. SWAT-Sim uses a gate level Verilog based simulator interfaced to an architectural simulator. Specific components into which faults are injected are simulated at the gate level, while the rest of the components are simulated at the architectural level to allow for faster simulations. The fault models used in the work are at the gate level. Our methodology on the contrary uses a transistor level simulation as well as more detailed fault models. Notably, SWAT-Sim produces upto three times the performance degradation for an architectural simulation.

On the contrary, our system increases the overhead by less than 42 percent. Their work focussed on simulating faults in the ALU, Decoder and Address Generator and understand the effect of such errors on the application. LIFTING [54] presents another fault simulation framework, that uses a C++ based simulator with a verilog bridge. The simulator uses a single global queue for event based processing. Gulati et al. [55] present a GPU based accelerator which is able to perform fault simulation using GPUs giving large speedups over commercial tools. An accelerator based fault simulator is shown in [55].

Quantifying effects of faults has seen renewed interest lately [56, 6, 57, 58] due to the increasing effect of transient faults on computation. Similarly, work has been done in understanding the effect of faults on memory systems, with its formalisation in terms of Architectural Vulnerability Factor of a circuit or architectural component [59, 60]. Existing research on designing a test methodology to support designs with unacceptable faults and the formalisation of effect of unacceptable faults in the form of error rate has been discussed in [61]. The authors argue that the error rate is relevant to understanding the usability of a circuit and potentially increasing the yield of processors at fabrication. Sridharan et al. [51] present results based on probabilistic modelling of arithmetic circuits. They present a stand-alone methodology to study the effect of random faults on the output of the adders. The individual transistors are given a probability of error and accordingly the probability of error in the output is computed using a Hadamard product.

This work presents an architectural simulator integrated fault simulator named FaultSim, which is capable of fast simulation of the implemented circuits. The following section presents FaultSim as well as its integration with the Gem5 [19] simulator.

4.2 Method

FaultSim has three parts: circuit modelling, circuit simulation and fault injection. Circuit modelling offers a simple way to represent circuit designs in the simulation stage and for integration into architectural simulators. Circuit Simulation takes care of efficiently simulating the circuit model. In this work, iterative optimisations are targeted to reduce the time taken for simulations.

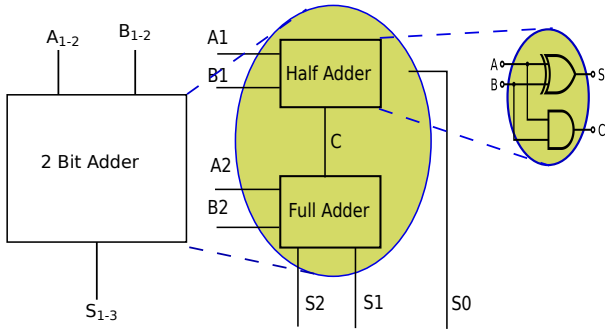


Fig. 4.1 A heirarchical model of a 2-bit adder.

In FaultSim, transistor-level circuit modelling is used to study the effects of transistor ageing and transistor faults. Gates that are built using the transistors are the building block of the digital circuits. In FaultSim, any digital circuit is referred to as a Circuit Node (CN). CNs can be as low-level as a gate, or as high-level as a block, such as a microprocessor. Each CN can be built using smaller CNs as well as wires to connect the smaller CNs. The network of wires connecting the CNs is referred to as the Circuit Connectivity (CC). Each circuit in FaultSim is hierarchically represented; each CN encapsulates a CC and a lower level CN. At the lowest level, a CN is a transistor and does not have an associated CC. Such a representation fits well with the Micro-architecture to device hierarchy in processors.

In Figure 4.1, a $2-bit\ adder$ is shown. According to the earlier definition, the $half\ adder$, $full\ adder$, the AND gate and the XOR gate are all CNs. The wiring between the half adder and full adder is the circuit connectivity of the 2 bit adder.

4.2.1 Circuit modelling

Each CN represents a logic block of the circuit, modelled as a multi-port block with internal logic. A CN has multiple ports that other CNs connect to, its internal logic transforms the input signals into the output signals. Large circuits are formed by connecting multiple CNs using a logic network. Hierarchical modelling is used to scale the Fault Simulator and for ease of usage. For example, in a buffer cell designed as two inverters placed one after the other, the parent CN will consist of two inverters and the CC will consist of one input and one output wire, with an an internal connection between the two inverters. The internal CNs connect to their specific port of the

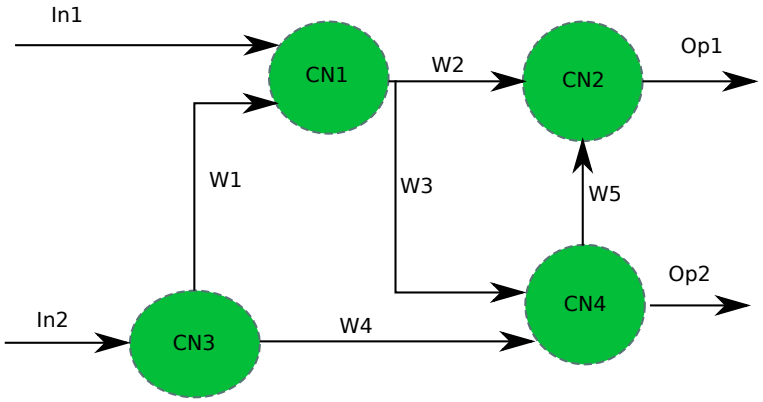


Fig. 4.2 Path delay modelling example

parent CN. Its notable that the critical path delay will be of 2 units in a buffer. This information is used while simulating and will be discuss further in this section.

Hence, for each CN, the internal logic blocks and their connectivity has to be specified; the critical path delays for each output is inferred upon initialization of the circuit. To scale the simulator, the network of the simulator is specified. Thus, a CC-CN co-modelling is used for hierarchical scaling. Each element has to be defined as an I/O node and circuits are designed as a network of such nodes. Each such network is then modelled as a node with specific inputs and output wires.

Hierarchical modelling allows for a CN to represent both high-level components, such as a decoder, or low-level components, such as a gate. The smallest CNs are gates for practicality. Hence, each circuit is modelled as a hierarchy of encapsulated CCs. The smallest unit of simulation is a transistor. For example, a 2-bit adder can be a gate-level specification or a higher-level modular specification using a half adder and a full adder. Once the circuit connectivity and behaviour are specified, then the entire circuit is preprocessed. Preprocessing and its significance are discussed in the next section.

4.2.2 Circuit simulation

An overview of the circuit simulation methodology is shown in Algorithm 1. The first step is initialisation the circuit.

Algorithm 1 FaultSim Overview

```
Initialise Circuit under Test (CUT)
for Each Input IV do
    Load IV to input wires
    Simulate(CUT)
end for
function SIMULATE(CN)
    for Each changed input wire W of CN do
        ScheduleWire(W)
    end for
    while Scheduler of CN is not empty do
        Find next simulation ready CN
        Simulate(CN)
    end while
    for Each Output Wire (OW) of CN do
        ScheduleWire(OW)
    end for
end function
function SCHEDULEWIRE(W)
    for Each CN in Connected(W) do
        Place CN in Scheduler of Parent(CN)
    end for
end function
```

Initialisation

When a circuit under test (CUT) is simulated, it is first initialized, as are all of the internal CNs. Initialisation involves finding critical paths for each output over the connectivity network. The weight of each wire is equal to the delay of the corresponding port of the CN it is connected to. As shown in Figure 4.2, if all of the CNs have unit delays, then the outputs Op1 will have a maximum delay of 3 units and Op2 will have a delay of 2 units. In case of delay faults, the delay is changed. For example, if CN1 now has a delay of 2 units, then Op1 will have a delay of 4 units (in2->w1->w3->op2) and Op2 will have a delay of 3 (In1->W2->Op1) units. For each node, we can individually identify the maximum delay from its output to the output of the circuit. This delay is called the delay distance for the CN. For example, the delay distance of CN1 will be 1 unit, and CN3 will be 3 and 2 units, depending on whether CN1 is faulty or not. Hierarchically finding the critical paths enables us to initialise the capacity of a scheduling queue for each CN. In this case, a queue capable of handling 4 time units is sufficient to simulate the circuit. The next step in the initialization process is to simulate the full circuit with zero input.

Circuit simulation

Once initialised, the input vectors are then provided sequentially to the CUT. Each input simulates the CUT and for each input wire where the value has changed (ScheduleWire in Algorithm 1), the wire marks all internal Child CNs (CCNs) connected to it for future simulation. The CCNs are simulated in the reverse order of their delay distance. To facilitate this, the CCNs marked for simulation are book-kept in a calendar queue (CQ) [62] of the enclosing or parent CN. Each CQ consists of multiple lists, each representing a predefined delta from the beginning of the simulation.

When the output of a CCN is different from the last iteration, the corresponding wire is marked as changed and all the CCNs connected to the wire are marked for simulation by placing them in the corresponding list (depending on its delay distance) in the CQ. The simulation happens in the order of the delay distance of the CCNs. For example, as shown in Figure 4.2, a change in W1 will cause CN1 to be scheduled to be simulated. The simulation of only CCNs connected to wires with changed output allow for iterative speedup of the system, especially if the input vectors do not vary

substantially. Such a method also prevents repeated simulations on a large part of the circuit.

Another optimisation technique is to have separate CQs for each CN. Having a CQ per CN instead of a global CQ reduces the number of potential time states to be book-kept and effectively reduces the CQ size and complexity. Additionally, FaultSim uses the event-driven motif to the gate level. Each gate is modelled as a CN which, if simulated, will simulate all of its transistors.

Higher level models

Given that FaultSim is primarily a fault simulator, the operation of the simulator is accelerated by only fully simulating the faulty components and using higher level logical models (HLM) otherwise whenever possible. Such a strategy is consistent with other fault simulators, such as CHAMPS [63]. For example, Figure 4.1 shows that if the circuit has no fault in the half adder, there is no need to fully simulate it and only a logical model of the half adder specifying the I/O relationship as well as the delay is required. The logical description for the higher level model is user provided. Critical path information for a higher level model is found through an offline simulation of the circuit. For each output signal, the corresponding critical path delay is also provided.

4.2.3 Fault models and fault injection

FaultSim supports multiple fault models, including transistor-level short and open faults, gate level stuck at faults and delay faults. FaultSim is able to do so by allowing faults to be injected at the user specified component. Faults are injected into the wires, specific transistors or into gates. It is notable that the different fault injection methods are equivalent to each other [64].

Among the fault models, transistor open faults have become a topic of interest. This is due to NBTI and HCI related transistor degradation [65]. In this context, transistor open fault deals with the increase in threshold voltage and effectively, the transistor preventing the current from passing through. In such a case, other transistors have to drive the output. This causes a state in which none of the transistors are driving the output, it is thought that the capacitive effects at the output cause the earlier output to be maintained. It is important to note that the the previous output of the transistor supports transistor open faults.

Delay faults are simulated by changing the delay value of the appropriate transistor or gate. As previously discussed, delay changes require reassessment of the entire circuit to establish the delay distance of each CN in the entire circuit. In the case of delay faults, the CN is then reassessed to resize its CQ if needed. The entire upward hierarchy of the CN is similarly updated.

Time specific injection of faults for supporting transient faults is also supported. In this case, a stuck-at fault is simulated for a predefined interval and then the fault is reset. For the purpose of this paper, not all the functionalities of the simulator were used; only stuck at faults were simulated to keep the perspective narrow.

When faults are injected into a transistor or gate, the state of the output wire connected to it is marked as changed, and the connected CNs are scheduled for simulation next time the whole circuit is simulated, irrespective of both the previously kept value or whether its inputs change in the current simulation. This assures that the output of further simulations resembles the appropriate value. Such a pre-emptive scheduling reduces the requirement for explicitly simulating the entire circuit every time a fault is injected or removed.

4.2.4 Integration with Gem5

FaultSim is directly integrated into gem5 CPU models. FaultSim components that are to be simulated are initialized and invoked when needed. Invocation is done by using gem5 event queues, whereby faults are injected at the specified time. Only components which have an error in them are simulated. During each clock cycle, if the specific architectural component is used in the gem5 simulation, the corresponding FaultSim implemented component (FSIC) is called. The necessary inputs for FaultSim are also collected at the same time from gem5. The values from the simulation are then collected and used as input to the FSIC. If the errors are time-limited or transient, the corresponding errors are removed queuing corresponding events in the event queue. Once all the errors are removed, the simulation of the FSIC is discontinued. Otherwise, FSIC included gem5 simulation continues until the end of the simulation. Such a method is consistent with similar methods, such as Swat-Sim [6]. Swat-Sim is integrated into Gems [66] which is an earlier version of gem5.

Modules as and when needed can be integrated into gem5. This is similar to the way Swat-Sim is integrated into Gems. FaultSim is initialised along with the CPU in

the architectural simulator. A recursive instantiation then occurs, where the modelled circuit is created by generating all the blocks in the internal heirarchy of the circuit; at each level of the heirarchy, the necessary internal connections are made. When the transistors are initialised, the delay of each transistor is set. Once the The CN enclosing the gate-level CN then uses the computed delays while initialising its CQ. The process is continued until the high-level block is complete.

4.3 Knowles Adders

Knowles [48] presented a simple parametric method to represent multiple adders, which, in our context, we will refer to as the Knowles family of adders (KFAs). KFAs uses the property that overlapping logic produces the same effect as a non overlapping one in the context of adders. For example, $F(1,2,3) \cup F(2,3,4) = F(1,2,3,4)$, where $F(R)$ is carry and propagate [67] for range R. Given that there are possibly $O(n^2)$ ways of partitioning a range into two overlapping sets, multiple methods exist to find the union over a given range. In any adder, each structure (black cells and white cells) [67] in the intermediate stage represents a merge of two such ranges. Hence, each cell in the adder structure is seen as a merge operation between the two cells that are linked into it. It is notable that multiple overlapping ranges can generate the same output based on the previous definitions. Specifically, adjacent cells on the same level in the adder structure can share one of their inputs. This allows for multiple different designs with largely similar logical structures, but different connectivity between those structures. It allows the design to have control over the number of wires in the system. Additionally, some members of the KFA have lesser logic compared to others. Such a property makes the Knowles class of adders an interesting case in fault tolerance. The variation in fault tolerance given minute changes in the structure is particularly interesting.

Each adder of regular KFAs can be specified by the number of outgoing wires at each level. For 32-bit adders, this then leads to six layers, with five sets of wires connecting these layers. Additionally, the maximum number of outgoing wires in a level is limited by 2^{ln}, where ln is the layer number. For regular KFAs, the number of outgoing wires is either one or 2^{ln}. Hence, each adder can hence be represented by a non-decreasing sequence (Adder Sequence or AS) $[s_5, s_4, s_3, s_2, s_1]$, where $s_5 \geq s_4 \geq s_3 \geq s_2 \geq s_1$ and $s_i = 2^j, j \in (0 \text{ to } i-1), i \in (1-5)$. AS shows the number of outgoing wires at each level.

Another interesting property is that some standard adder designs are elements of the KFA. For instance, 16-bit Sklansky adders can be derived from the KFA AS [8,4,2,1] ; also, KFA AS [1,1,1,1,1] is the same as a Kogge Stone 32-bit adder. Given the generic nature of the KFA and the ability to represent named designs as elements of the KFA, the KFA makes for a relevant generalisation class for our study. Hybrid KFAs follow a similar design principle as a regular KFA, with the only difference being that the number of outgoing wires can be any number between 1 and 2^{ln}. Hence, hybrid KFAs are hence a mix of multiple adder designs within the regular KFAs.

In the next section, we discuss work done in quantifying errors in different circuits (including adders) and the various fault simulators currently available.

4.4 Experimental Setup

The experiments were run in two sets: gem5 [19]-integrated online experiments and offline FaultSim-only runs. For both experiments, we generated the inputs using gem5 and the ARMv7-ISA, running SPEC2006 [68] benchmarks. Table 3.1 presents the exact architectural details used within the gem5 simulation. Each benchmark was fast-forwarded for 1 billion instructions and then caches and branch predictors were warmed for another 250 million instructions, after which 25 million instructions were run. Fast-forwarding is performed to neglect the initial phase of the application, which generally deals with initialising and reading the input files. Warming-up is intended for the experiments benchmarking the fault simulator. Due to continuity requirements, the same set-up is used for both set of experiments. Numerical results from the FaultSim experiments are not given back into the gem5 simulation. This is done to keep all of the experiments consistent, processing the same set of operations. Hence, in the experimental set-up, the effect of faults on the software level are not considered.

For off-line FaultSim experiments, the relevant inputs for each instruction in both in-order and out-of-order mode are noted. Additionally, for the out-of-order core, each instruction is marked with committed or not committed, in order to tabulate the irrelevant computations in the out-of-order setup. After logging, the logged operations are run on our fault simulator in an offline setup. The number of operations ranged from 2 to 15 million.

For on-line experiments, an ALU is built using FaultSim (AFSim). The faults that need to be simulated are also inserted into AFSim. The AFSIM is then connected to the appropriate locations in Gem5. In our implementation, we added AFSim to the functional unit pool of the Out-Of-Order core implementation in Gem5. Whenever an ALU instruction is executed, it is passed to AFSim. Since FaultSim does not provide any fault-detection, the Gem5 execution of the instruction is not changed. The difference in output value or Error is calculated based on that. As discussed earlier, the granularity of execution depends on whether the faulty component is going to be used or the inputs have changed across iterations. Given that there is explicit state being kept for each block, the state kept for a circuit is a multiple of the number of transistors in the circuit. FaultSim makes a tradeoff between space and performance in favour of performance.

FaultSim is implemented using C++11. The benchmarking experiments were run on an Intel(R) Core(TM) i7-3770 CPU with 32 GB of memory, and were compiled with gcc 5.4 with the O3 flag. Time is noted using the C time.h library.

Adders are a commonly used circuit and an important part of fault tolerance literature. For ease of usage with gem5 as well as for comparison with similar work in literature, adders were selected. Knowles family of adders (KFA) [48] is used for simulation due to its ability to represent numerous adders parametrically. Each member of KFA can be represented by a pattern of size $log(n)$; where n is the number of bits of individual input to the adder. For example, a 16 bit adder would be represented by a 4 element pattern. Larger adders that share a prefix with smaller adders would have the smaller adder structure within them. As a result, scalability experiments can be performed with a consistent design motif. To this extent, KFA is selected to study effect of faults on many circuits and to maintain consistency in scalability experiments, KFA is chosen.

4.5 Results

Two set of experiments were run, the first benchmarking the simulator and the second testing the error resilience of the adders. The benchmarking experiments form the primary focus of this paper. The sequence of benchmarks are the same across the results, with Table 4.2 showing the mapping of the benchmarks with their IDs.

4.5.1 FaultSim performance

Figure 4.3 shows the scaling of FaultSim as the size of the circuit undergoing testing increases. The relative time and size as compared to a 2-bit adder is shown. Do note that the adders are modelled using the Knowles Adders notation. Hence, each larger adder shares part of its design with the smaller adders. As can be seen from the figure, the results show a nearly constant relationship between the size of the circuit and the time taken to simulate it.

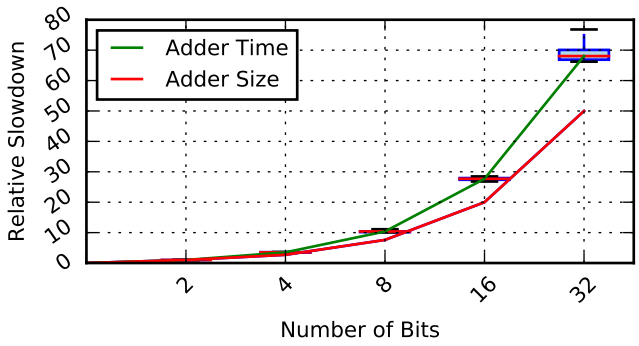


Fig. 4.3 Scaling of FaultSim with increasing circuit complexity. Relative increase in time and size compared to a 2 bit Knowles adder is shown.

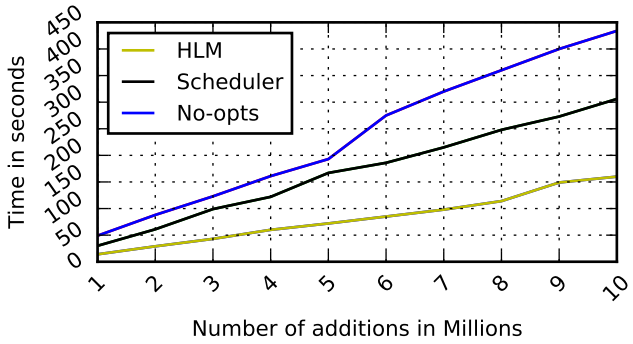


Fig. 4.4 Performance comparison of different optimisations

Figure 4.4 shows the effect of different optimisations on the performance of FaultSim. The mapping from each benchmark id to the benchmark from SPECCPU2006 [69] is shown in Table 4.1. The No-opts version shows the faultSim version with only a level-based trigger using the calendar queue. For a given circuit, components are triggered in the reverse order of their delay distance from the output. All the nodes

Id	Benchmark	Id	Benchmark
1	astar	2	bwaves
3	bzip2	4	cactusADM
5	calculix	6	gamess
7	gcc	8	Gems
9	gobmk	10	gromacs
11	h264ref	12	hmmer
13	lbm	14	leslie3d
15	libquantum	16	mcf
17	milc	18	namd
19	omnetpp	20	perlbench
21	sjeng	22	soplex
23	tonto	24	zeusmp

Table 4.1 Benchmarks and their ids

are simulated, which effectively stopped the iterative optimisation. The scheduler version adds in the scheduler and masks out nodes which do not have their inputs changed. As seen in the figure, approximately 40 percent reduction in time can be seen between the two versions. Additionally, on average, approximately 10 percent of the transistors were simulated for each operation. This reduction in simulated transistors reduced the overheads substantially. The limited number of transistors simulated per instruction can be attributed to the event-based scheduler used. When HLMs are used, a reduction of approximately 65 percent as compared to the No-opts, and a 50 percent improvement as compared to the scheduler versions can be seen. Profiling showed that the time needed to simulate the circuit nodes was reduced by a factor of 6. The number of transistors scheduled is also significantly reduced, and the scheduler became the dominant factor in the simulation.

The performance of the simulator is further dependent on the sequence of operations. Sequences with a low difference between them cause fewer CNs to switch their outputs, causing fewer simulated CNs, hence greatly affecting performance. Such a variation can be clearly observed in Figure 4.5, where results from different benchmarks are seen. The time per million additions vary across the benchmarks, and is due to the number of CNs within the adders that are simulated per addition operation. Its clearly visible that hmmer had the largest variations in the input for the sampled duration and sjeng had the least.

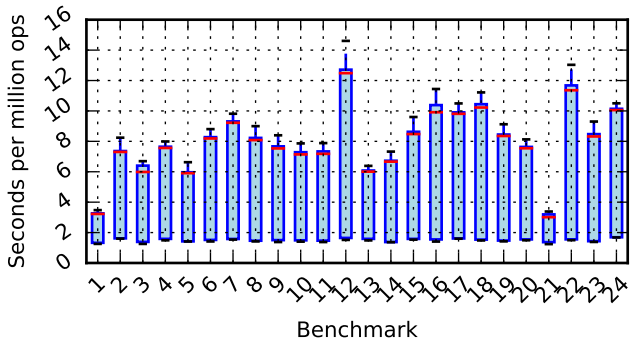


Fig. 4.5 Time in seconds per million 32 bit addition operations using FaultSim using SPEC2006 benchmarks.

Performance improvement over iterations is also observable. With 1000 addition operations using random numbers taking 28 micro-seconds per addition operation. This improved to 18 micro-seconds for 10k additions, and after 160k additions performance increases to 14 micro-seconds, and does not improve any further. This again is indicative of the scheduling mechanism.

4.5.2 FaultSim-Gem5 performance

The first set of experiments benchmarks FaultSim integrated in Gem5 and running SPEC2006 [69] benchmarks. The CPU utilised in the set-up has two adders, and in one experiment both of the adders are simulated, and in another only the most used adder is simulated. Over the 24 benchmarks, with an out-of-order super-scalar processor, on average the overhead was an additional 16.1 percent on average, with the highest overhead being 43 percent. If only one adder was simulated, the average overhead reduced to 10 percent, and the peak overhead reduced to 26%. Do note that the second adder is given lesser preference in gem5 simulator. In our experiments, in single adder mode, the second adder is not simulated, leading to a non-linear overhead.Table 4.2 shows the overheads for each of the benchmarks. Total simulation time of gem5 and the number of adder operations affect performance of the simulator. Additionally, similar values at the inputs cause reduction in time taken.

It is notable that even though performance is poorest for the leslie3d benchmark, it is not the slowest benchmark in our simulation, as seen in Figure 4.5. The variation is caused by gem5 being able to process leslie3d faster in comparison to hmmer for

Id	Benchmark	Slowdown	Id	Benchmark	Slowdown
1	astar	1.05	2	bwaves	1.01
3	bzip2	1.04	4	cactusADM	1.07
5	calculix	1.06	6	gamess	1.17
7	gcc	1.11	8	Gems	1.14
9	gobmk	1.26	10	gromacs	1.09
11	h264ref	1.17	12	hmmer	1.30
13	lbm	1.09	14	leslie3d	1.43
15	libquantum	1.33	16	mcf	1.14
17	milc	1.16	18	namd	1.14
19	omnetpp	1.18	20	perlbench	1.07
21	sjeng	1.07	22	soplex	1.23
23	tonto	1.16	24	zeusmp	1.41

Table 4.2 Relative runtime of gem5-FaultSim as compared to a gem5 only run

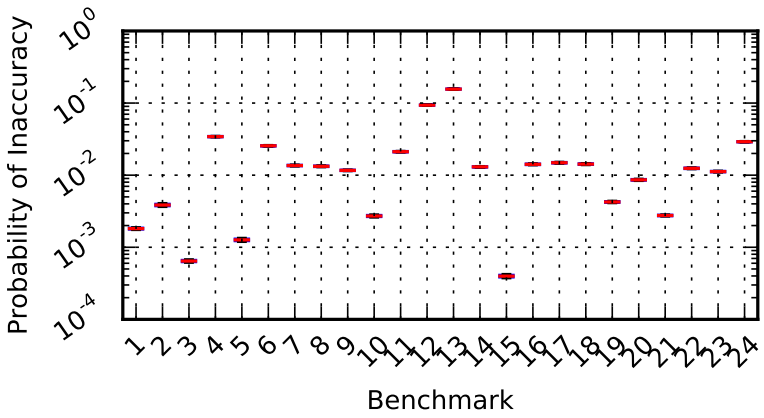


Fig. 4.6 Probability of error for single faults across different benchmarks.

the sampled duration. Additionally, Figure 4.5 shows results using benchmarks, with the position of the fault varied, clearly showing that the position of the fault is an important factor in the performance of FaultSim.

Figure 4.6 shows the results for a single fault (fault placed at different places in 42 different 32-bit Knowles Adders) and the effect on accuracy of the output across benchmarks. For most benchmarks, a single fault gave a less than 1 percent probability of error. Benchmarks hmmer and lbm have a greater than 10 percent probability of error in the output. The faults do not cause errors everytime as Knowles adders are inherently redundant, with each input passing through at least two blocks. Such a logical redundancy allows Knowles adders to be fault-tolerant.

4.5.3 Comparison with other fault simulators

The overheads are substantially lower than those reported by Swat-Sim [6]. Swat-Sim integrates their simulator into an earlier version of the Gem5 infrastructure (Gems simulator). They report slowdowns of the order of 56 percent on average and a maximum of 120 percent using gate stuck-at fault models. Do note that ours works at a transistor level, and is able to support larger number of fault models. Furthermore, in their case only one adder was simulated out of the two available. As shown above, the overheads of FaultSim are approximately three times lesser than Swat-Sim.

Similar fault simulation work using GPUs is shown by Gulati et al. [55]. Among the circuits they have simulated, they have shown 4-bit adder simulations and that 32K simulations on their GPUs (GTX 8800) for 4-bit adders from ISCAS89 take 0.089 seconds to simulate. On the contrary, our single threaded method takes 0.040 seconds to simulate on a Corei7-3770.

4.6 Conclusions

FaultSim is a fast transistor-level fault simulator which is capable of numerous fault models. FaultSim uses a delay distance measure along with calendar queues to speedup up performance. Performance results presented are comparable or better than other fault simulators currently available. The overhead for a Gem5 integrated version is on average 16 percent. Additionally, the average time for a 32-bit addition is 15 micro-seconds. Scalability is clearly seen, with the simulator scaling linearly with respect to the number of elements in the circuit.

FaultSim presents the probability of error for faults in different parts of an adder. The results show that for single errors, the probability of error is less than 10 percent for most fault-benchmark pairs on the Knowles family of adders. Hence, it can be seen that large faulty circuits can do useful work most of the time, given sufficient overlapping computations and natural redundancy. If error-detection and error-correction methods were present, then the errors caused by the faulty circuit would not affect the correctness of the system. PreFix, as presented in the next chapter, is one such method that uses prediction and detection of error along with instruction re-execution to tolerate faults while keeping the faulty components active.

Chapter 5

PreFix: Fault Tolerance using predictive remote re-execution

REPAIR showed that re-execution of instruction remotely can provide practical fault-tolerance, especially when supporting logical blocks within the processor. Further investigation of logical blocks using FaultSim presented that the probability of fault is less than 10 percent for single faults. This points towards a system combining targeted re-execution for select instructions. PreFix brings together targeted re-execution developed using REPAIR with fault models developed with FaultSim.

Continuing fault oblivious operation of a processor, in the presence of hard errors, is a challenging task. This is because micro-architectural components will behave differently from their designed behaviour in the presence of errors. The modular design of processors allows errors to be localised and not have any direct hardware effects outside the module to which they belong. Hence, the zone of inaccuracy in terms of hardware correctness is limited and can be isolated. This can lead to schemes that allow such errors to be handled. Modularity also allows for component usage logging for a given instruction accurately, the estimation of components given the current system state, can also be done with limited knowledge of current processor state.

In PreFix [1], each instruction is evaluated to check if any errors were generated while using faulty components and if so, the instruction is re-executed in a remote core and the results are compared. This is similar to REPAIR, but instead of pro-actively re-executing, we reduce the re-execution substantially by predicting faults, and verifying

[1] This work is due to be published at DFTS 2017

them. We claim that hard faults in both data flow and control flow based sections of the processor can be easily dealt with, using PreFix.

There are four major contributions of PreFix. Firstly, it allows instructions to continue executing on faulty hardware. Secondly, PreFix only handles instructions which might have caused an error. Thirdly, the possibility that an instruction will cause an error is known before the fetch stage receives the instruction from the memory. Lastly, the hardware faults are modelled and mapped to instructions using a fault tree.

5.1 Motivation

Traditional reliability methods do not use the faulty components, and rely on completing the computation elsewhere. These techniques have been called *Fault Intolerant* [70] as they block off the component which has developed errors. In contrast, we propose to deal with hard faults using a *Fault Tolerant* approach, allowing components to be active despite the presence of errors. This is backed by prior work which has shown that at least 30% [53], to up to 65% [71], of faults do not affect the correct execution of a component over millions of cycles.

In the literature, fault tolerant methods have been used exclusively in handling soft faults. Transient faults are sporadic and detectable, yet non-predictable and infrequent over spans of hundreds of cycles. These properties lead to detection and correction schemes that maintain circuit usage despite the possibility of errors. In the same vein, prior work has shown that hard faults can also have similar properties [53, 71]. Hence, an unexplored area of research is in the area of fault tolerant methods specialized for hard errors.

Chapter 2 presented that process variation and the type of workload affects the ageing of a processor core. Two cores running different applications will age differently. This allows to build a collaborative system, which can use the ageing profile of the cores, which redirects the execution of an instruction to an appropriate core, and the multi-core would continue running applications oblivious of faults in the cores. As seen from the results from Chapter 4, the probability of a fault being converted into an error is significantly low. Hence, using the collaborating cores to verify execution provides an opportunity for an out-of-order core to continue executing instructions

which do not have any dependency on the remotely executing instruction. If an error is detected, the instruction pipeline is squashed.

Our solution, named PreFix, is one such collaborative re-execution system which remotely re-executes instructions. PreFix aims to provide a low-overhead error detection and correction technique for tolerating hard errors. This is achieved by predicting and verifying the possibility of error on each instruction passing through the core. Instead of turning off faulty structures, faulty structures continue operation and the results of individual instructions that use these circuits are corrected. Errors in the results of each instruction is conservatively predicted and their execution is duplicated on a different core, the results are corrected if needed along with any side effects of the instruction. PreFix allows fault-tolerance by sacrificing performance for correctness. PreFix brings together fault detection techniques with redundant execution on a different core to both detect and correct permanent errors. Overall, it enables continued use of faulty components, allowing them to perform useful computation when faults do not propagate.

5.1.1 Related Work

Prior research in fault tolerance is split into schemes for error detection, methods for error tolerance, and techniques to extract useful performance from a core with permanent faults. Multiple methods to handle hard faults focusing on fault intolerance have been discussed in literature [72]. Stagenet [28] is a method by which multiple processor pipelines are interleaved to allow for switching off parts of the pipeline which are faulty. In contrast, Necromancer [31] uses faulty (so-called "dead") high-performance cores to accelerate operations on a smaller core. A compiler-based method is presented by Meixner and Sorin [35], where code is recompiled so that the faulty hardware is not used. Khan et al. [36] present a method where a hypervisor keeps track of faulty cores and the profile of the threads running in the system. It uses this information to match a core to a thread at runtime. Finally, Rescue [25] presents a method to isolate logic modules, providing for better fault localization.

Fault tolerant methods, on the other hand, generally execute a second version of the application. Same core multi-threading based redundancy for soft errors is used by [73] (some permanent faults can be handled) and [74] (sampled subset re-executed using prediction) Further, idle cycles have been used to provide soft error protection

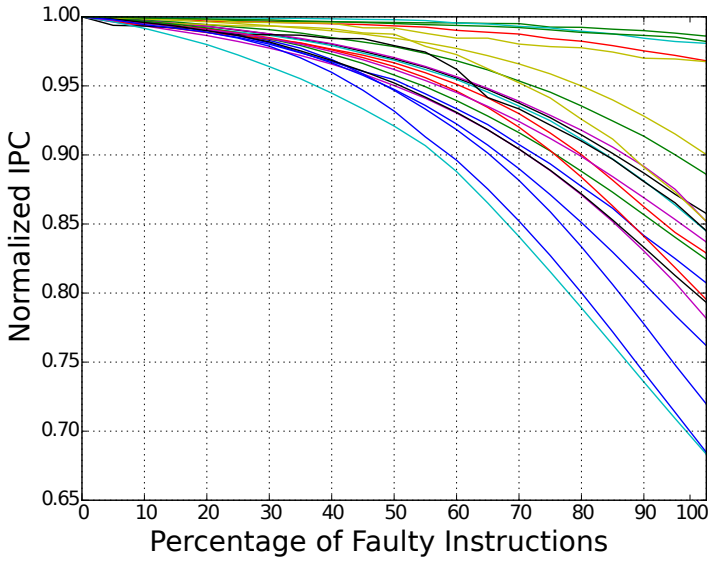


Fig. 5.1 Performance degradation caused by preventing instructions that would incur errors from passing through a partially faulty ALU.

by Gomaa and Vijaykumar [75] to guard against transient faults. However, each of these schemes mainly targets transient faults whereas our work focuses on hard errors. Blackjack [24] targets hard faults, through execution in a separate SMT context with instruction shuffling in the front-end to avoid the original and duplicate instructions from using the same components.

In summary, there has been significant prior work that farms out execution of duplicate instructions to an additional thread. However, little work has addressed the issue of reducing the number of instructions that require duplication and re-execution to keep a faulty core running. In Chapter 5.2 we develop PreFix, which uses prediction of whether a fault will manifest itself on a particular instruction to reduce the overheads of instruction re-execution on a separate core.

5.1.2 Justification for PreFix

To show the utility of keeping components active despite faults, an experiment with two ALUs was designed. One of the ALUs has faults in it and will only execute a certain set of instructions correctly. This ALU rejects any other requests. Such a system was implemented within Gem5 [19], with one of the ALUs marking itself not ready to accept certain instructions. The ratio of such instructions was increased from

zero to hundred percentage. These instructions can be executed instead on the other, fault-free ALU within the same core. Figure 5.1 shows the degradation in performance when one of two integer ALUs is faulty, yet allowed to continue operating. In this oracle study, to show the upper bound on performance available, we simply prevent the ALU from accepting an instruction if it will produce an erroneous result. The x-axis shows the fraction of instructions for which the ALU would have computed erroneous results; the y-axis gives the IPC, normalized to the fault-free case. We plot values for each of the SPEC CPU2006 benchmarks, described in Section 5.3 along with full details of our experimental setup.

When the ALU is faulty for every instruction (i.e., 100%), there is considerable difference in performance between the benchmarks. The variation ranges from a negligible 2% to a much more substantial 30%. For these latter applications, it is vital to keep the ALU functioning, even if it only produces error-free results for a fraction of the time. When 50% and of the instructions pass through, performance reduction is a mere 7%. Many existing works would turn off this ALU, meaning a 30% loss of performance instead. PreFix aims to address this challenge by predicting whether each instruction will actually be faulty as it passes through the core's pipeline, providing duplicate execution for instructions that may have erroneous results. In this manner it allows faulty components to continue performing useful and correct work.

5.2 PreFix

PreFix is a technique that allows the continued use of faulty micro-architectural components while maintaining high performance and correct execution. PreFix consists of at least two cores: the first contains one or more errors, and we denote it the Faulty Core (FC); the second is healthy and we call it the Remote Core (RC). The remote core is responsible for re-executing instructions that have been marked as possibly faulty on the faulty core. In the scenario that both the cores are faulty, they can continue operating till their faults are no longer orthogonal to each other.

We augment each core with a fault prediction unit that gives an indication of whether each instruction is likely to produce erroneous output or not. PreFix duplicates those that are, and sends some of them over to a central queue for duplicate execution on the RC and their results placed back in a central queue. As instructions pass through the

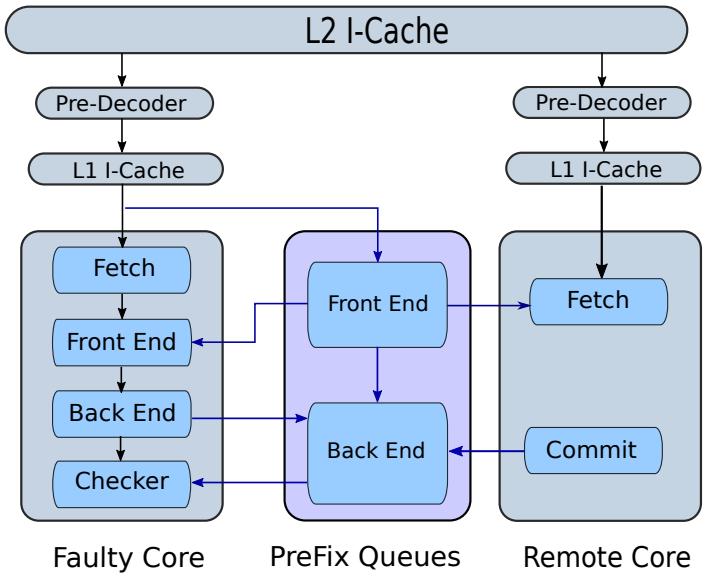


Fig. 5.2 PreFix overview showing CPU pipeline integration.

pipeline on the FC, additional logic verifies the substantiation of errors. The results of the erroneous instructions from the FC and RC are compared at commit to check for consistency. Any difference in execution causes the FC to update its results, using the duplicate execution on the RC, and flush its pipeline to avoid propagating the error to any later, dependent instructions.

PreFix ensures that errors contextually do not propagate beyond the FC, despite the use of hardware with errors. It assumes that the FC maintains at least the ability to load and store both instructions and data. PreFix is intended for handling faults in the core logic, not those in buffers, since there are simpler and more efficient methods to accomplish this [76]. An overview of PreFix is shown in Figure 5.2. Additionally, Figure 5.3 shows PreFix in a multi-cpu configuration.

We first describe how instructions flow through the pipeline, then give details of each of the micro-architectural structures that PreFix requires.

5.2.1 Instruction Flow

The first interaction instructions have with PreFix is within the pre-decoder that sits between the L1 instruction and L2 caches. The pre-decoder identifies resources that each instruction requires and stores that within the L1 instruction cache, for use in a later stage. When instructions are fetched into the core, they simultaneously

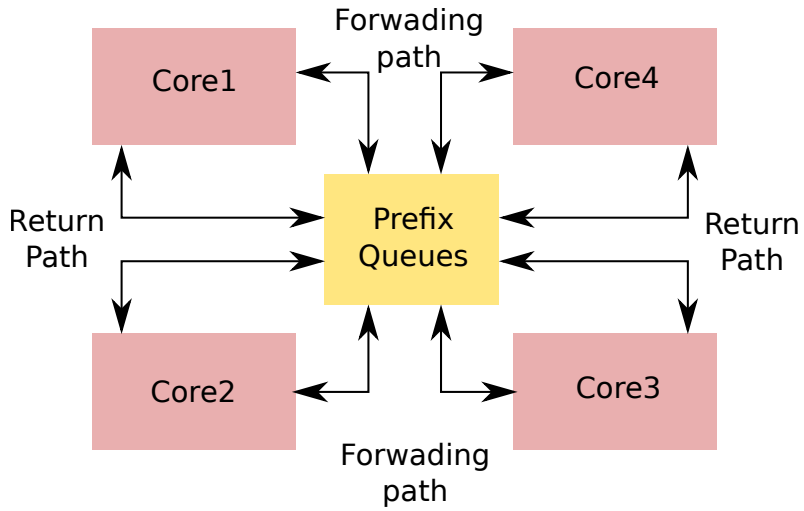


Fig. 5.3 PreFix overview showing a multi-CPU configuration.

pass through the first main PreFix structure, the fault prediction unit. This uses the information from the pre-decoder and the instruction itself to determine whether the instruction is likely to execute correctly within the core. If an error is possible, then a copy is placed into a holding queue for duplicate execution on the RC. If the instruction cannot be handled by the FC, then it is forwarded immediately; other instructions are held in the queue until the PreFix back-end informs it of possible faults.

Meanwhile, all instructions enter the faulty core's fetch queue and pass as normal through the core's pipeline. Fingerprinting logic monitors execution and use of resources, flagging up an error if an instruction does not produce the correct result. At the commit stage, a corrector module uses the outputs from the fingerprinting, and only allows instructions with the correct result to commit and leave the pipeline. Those with faults are replaced with the results from the duplicate instruction on the re-execution queue; the pipeline is flushed and fetch restarts with the next instruction.

5.2.2 Pre-Decoder

The pre-decoder is responsible for extracting early, high-level information from each instruction that enters the instruction cache. Many modern processors contain pre-decoders at this level [77, 78] to reduce the amount of repeated work that the pipeline's decode stage must perform, trading off additional L1 instruction cache storage space

against a reduction in logic, energy and time in the decode unit. Every instruction placed into the L1 instruction cache is pre-decoded and expanded to have information on the resources needed for each instruction. Instruction pre-decoding in current architectures extracts information about the instruction's class and resources required during its traversal through the pipeline, which is exactly the information required by PreFix. This enables methods to predict processor utilization [79, 80] which is also useful for our predictor, as explained in the next section.

5.2.3 PreFix Frontend

Having briefly described how an instruction interacts with PreFix, we now describe each of the structures from the PreFix frontend in more detail.

PreFix Predictor

Pre-decoded instructions enter the PreFix front-end in parallel with being sent to the main core's fetch pipeline. The structures that make up the front-end are shown in Figure 5.4. The primary task of the predictor, supported by the fault trees is to classify each instruction into one of the three categories: not faulty (NF), highly likely to fault (HLF), or low likelihood of fault (LLF). If the detectors in the processor are not capable of detecting certain faults, then the corresponding instructions are marked as HLF. Faults marked LLF are caught by detectors. The predictor is necessarily conservative; it generates false positives but never says an instruction is fault-free when it isn't. To actually make its prediction, this unit relies on hardware fault trees, described in more detail in Section 5.2.3. The predictor further contains a logic which calculates when an instruction will use a faulty pipeline lane, to deal with errors in specific fetch or decode units.

NF and LLF instructions pass through the core's pipeline. LLF and HLF instructions are duplicated, with the HLF duplicates immediately sent to the RC. LLF instructions are only re-executed if the detectors catch an error. The stream of HLF and LLF instructions are written into the re-execution queue, from which they leave in program order only when their original counterparts commit or are squashed in the main core. The re-execution queue is dual channeled, one sending instructions and operands to the RC (the outward queue) and the other receiving results from the RC, if and when that occurs (the results queue).

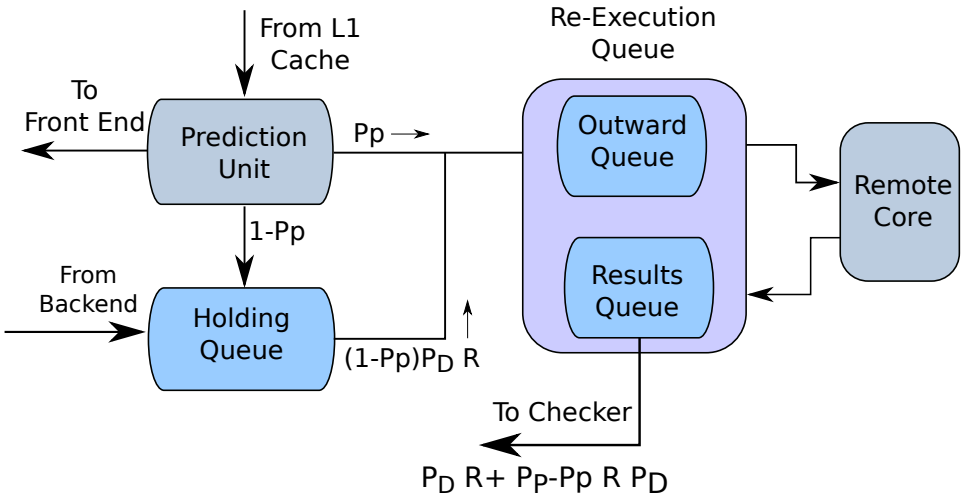


Fig. 5.4 PreFix front end showing predictor and the queues. Further the probabilities through each section are shown.

Fault Trees

Given the inherent hierarchy of processor components, a fault tree is a natural method for storing faults in micro-architectural structures. The fault tree in our method works over the ISA. It groups instructions by resource usage (i.e., core structures) and predicts whether instructions from each group might use faulty components as they pass through the processor pipeline.

Pre-decoders support ISA-based component analysis allowing the creation of fault trees where groups are based on ISA-level characteristics. At the head of the tree, all instructions are part of the super-group, that is the group of instructions using the processor. Further down the tree, the instructions are split into more specialized groups, for example, based on the specific type of functional unit they will use. As the tree becomes larger, the nodes start representing internal circuitry, such as an operation's bit width.

Hardware represents the tree in its flattened form as a bit array. Further, for each element of the array, the ability of PreFix to detect the fault is also stored. As mentioned in Section 5.2.2, the pre-decoder provides information regarding the instruction class and the resource requirements for each instruction. This is used to query the fault tree, which is populated using built-in self-test [81].

Duplicate Execution

The RC executes duplicate instructions alongside any workload it has to run. To achieve this, each core contains an otherwise-idle redundant thread. These obtain duplicate instructions from the re-execution queue and execute them when the RC allows. In each fetch cycle, the RC either fetches from its main thread or the redundant secondary thread (if it has work to do). The RC favors its main thread, giving it more fetch cycles than its redundant counterpart. In our experiments, fetch occurs from the redundant secondary thread only when the primary thread is inactive while waiting for either data or instruction from memory.

Instructions marked as ready in the re-execution queue contain not just their original instruction bits, but also their source operands. This means that the redundant secondary thread is free to execute each instruction in isolation, asynchronously to the original faulty core. Redundant secondary threads are non-speculative in the RC since they are independent of all other instructions and do not use the branch predictor. The results from this duplicate execution are available at commit and are written into PreFix's re-execution queue, for reading by the PreFix checker unit if the original instruction actually does experience an error.

5.2.4 PreFix Backend

The PreFix front-end is concerned with predicting whether a fault may occur with each instruction and providing efficient duplicate execution of it. The back-end, in contrast, is responsible for detecting whether an error has actually occurred and ensuring that no fault propagates out of the core to affect architectural state.

PreFix Fault Detector

The PreFix fault detector is responsible for detecting whether an error may have occurred. If so, it communicates with the front-end to ensure that the duplicate of the faulty instruction actually gets executed on the RC. Note that the detection need not be perfect, and over-prediction is acceptable with performance penalties.

The locations of the PreFix detectors are shown in Figure 5.5. The PreFix back-end contains both a usage monitor (not shown) and multiple detectors. The usage monitor

runs in parallel with instruction issue and checks instructions that were marked as possibly faulty by the front-end to see if they will actually use any faulty components. If not, then it reclassifies the instruction as NF. This component helps reduce the overheads of PreFix by removing false positives introduced by the front-end. It also serves as a backup to the detector. The usage monitor is responsible for filtering instructions that require checking and the detector is responsible for detecting faults in marked instructions (those classified LLF). Detector designs have been previously proposed, for example, using parity checking [82, 83], and PreFix can work with any of these methods. Instructions passed by the usage monitor are placed in a detector queue, pending the results of the detector. From here they are are either discarded (if no fault is detected) or, in the event of a detected error, sent to the holding queue in the PreFix front-end to ensure they are re-executed on the RC.

As Mohanram et al. note [83], to save on space overheads and provide targeted protection, detection need not provide complete coverage. To provide a fault safe design, an additional usage monitor can be recommended. In addition, Mitra and McCluskey [84] show that concurrent error detection methods themselves may be subject to errors. Inclusion of the usage monitor, therefore, protects against scenarios in which the detector as well as the actual circuit have complementary errors. Where the detector is itself faulty for certain errors, or does not provide complete coverage, the usage monitor's filtering alone is used to determine whether to re-execute the instruction. In these scenarios, false positives can occur from the PreFix back-end.

Corrector Unit

For each instruction that is still marked as potentially faulty (LLF), the corrector is responsible for checking if the results from both executions match. It sits at the end of the pipeline, alongside commit and is responsible for ensuring that instructions with erroneous results do not leave the pipeline and so do not contribute to the architectural state.

Instructions that have been marked as faulty by the PreFix back-end after their execution are intercepted by the corrector unit and prevented from committing until they have been validated. To accomplish this, the corrector unit queries the instructions at the head of the holding queue to find the duplicate of the erroneous instruction. If this has already been executed on the remote core through the re-execution queue and

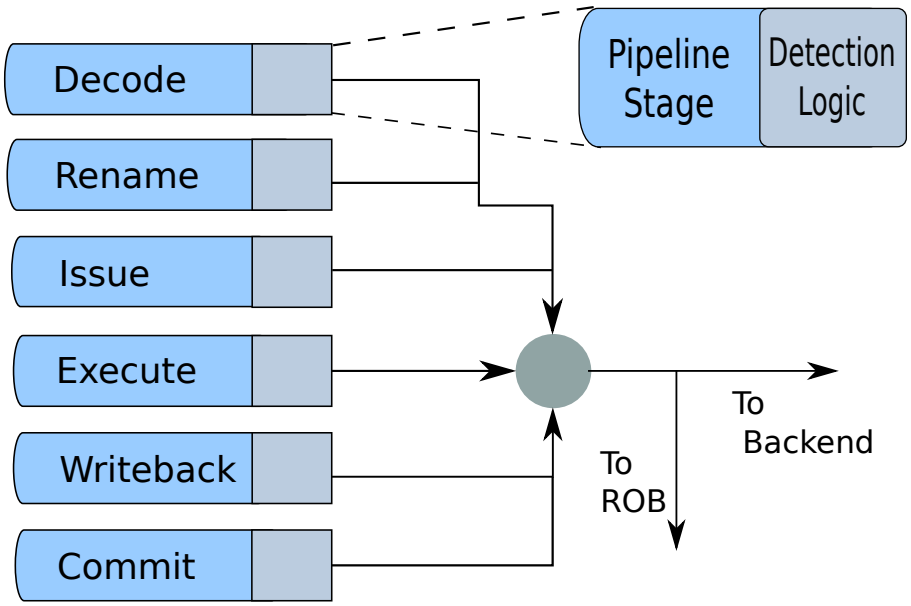


Fig. 5.5 PreFix detection logic. Each component has a related detector and any trigger causes a re-execution.

redundant secondary thread, then the values from the remote execution are retrieved. On the other hand, if duplicate execution has not yet finished for this instruction, the corrector unit stalls until the remote results come back.

As Section 5.2.4 described, the PreFix back-end can generate false positives. To avoid a loss of performance for these false positives, the corrector unit does not assume that a fault has actually occurred. Instead, it compares the results of remote execution with those from the faulty instruction to actually determine the instruction's fault status. If they are the same, then the faulty instruction commits as normal. If they differ, then the results from the remote execution are accepted as correct and copied into the faulty instruction's output registers. The corrector unit stalls the processor, and squashes all later in-flight instructions to avoid subsequent errors from dependent instructions reading the wrong value. At this point, as with a branch mis-prediction, the instructions in the holding and re-execution queues are also squashed. Any instructions currently being re-executed on the remote core are ignored when they finish. Execution on the remote core is independent and asynchronous to that on the faulty core, hence there is no interaction between the two.

5.2.5 Parameter Dependence

As shown in Figure 5.4, the instructions flow through two paths in the system. Either directly from the predictor (probability of a possibly-faulty instruction being an HLF is P_P), or LLF from the back-end with probability P_D (with the frequency of a possibly-faulty instruction being an LLF as $1-P_P$). R represents faulty component usage probability when the faulty component might have duplicates. Further, P_C is the probability of results from re-execution not matching. Let F be the actual rate of faults causing an error. For each predicted and detected instruction, let the delay for re-execution be d_1 and d_2 respectively. Also, let the delay for clearing the pipeline be d_3. The total delay caused by PreFix is T_d. Hence,

$$F = P_C(P_P + (1-P_P)P_D R) \tag{5.1}$$

$$T_d = d_1 P_P + d_2(1-P_P)P_D R + d_3 F \tag{5.2}$$

$$\Rightarrow T_d = P_P(d_1 - d_2) + \frac{d_1 F}{P_C} + d_3 F \tag{5.3}$$

Given that d_2 is always greater than d_1, prediction clearly improves the performance of the FC. Prediction cannot guarantee the exact component usage, and hence has to over-predict. Increased prediction causes the RC to slow down and also re-execute more instructions, increasing d_1 and d_2. Hence there exists a mid-point for the trade-off, which is explored in the experiments in Section 5.4.

5.3 Experimental setup

We evaluated PreFix [2] using the gem5 simulator [19] using the ARMv7-A ISA and randomly-selected pairings of applications drawn from the SPEC CPU2006 benchmark suite. The out-of-order cores have private L1 caches and a shared L2. Table 3.1 details the core and memory configuration.

We created workloads for evaluation, as shown in Table 5.1, by randomly selecting applications from the SPEC CPU2006 [69] suite. We compiled each benchmark with gcc 5.2; missing applications would not compile or run correctly in our environment. For each experiment, we fast forwarded and warmed the caches and branch predictor for each benchmark for 500 million instructions and then executed for at least a further

[2] code available at https://github.com/jyosoman/docker-gem5-repair-pirafix

1	astar	sjeng	2	bwaves	pds50
3	bzip2	tonto	4	cactusADM	tonto
5	calculix	zeusmp	6	gamess	soplex
7	gcc	bwaves	8	gobmk	cactusADM
9	gromacs	calculix	10	h264ref	gamess
11	hmmer	gcc	12	libquantum	gobmk
13	milc	h264ref	14	namd	hmmer
15	perlbench	libquantum	16	sjeng	mcf

Table 5.1 Randomly-selected pairs of benchmarks studied.

250 million instructions. The weighted speedup [85] of the IPC of the main threads (i.e., those running applications on the FC and RC, but not the redundant thread on the RC) is taken as the performance indicator. To allow for a viable comparison, the base case is taken as the error free multicore case.

Our experiments required us to simulate cores containing errors, to test the effectiveness of PreFix. To achieve this we created 50 versions of the first core, each one containing exactly 5 errors in different components.

For benchmarking experiments, faults had a 20% chance of affecting the result of each instruction that used the faulty component. To observe the effect of variation in the parameters, we further perform experiments changing parameters while keeping the number of faulty instructions constant. Further, we studied the area and space overheads using McPAT [53].

5.4 Results

We first show the performance of the full PreFix technique, then the contributions of parameter variation on performance.

5.4.1 PreFix performance

Figure 5.6 shows the results of PreFix when the complete system is functional for each workload across all 50 erroneous systems. The x-axis gives the workload number from Table 5.1, the y-axis shows normalized performance and we plot the minimum, maximum, median, 25th and 75th percentiles of the distribution across erroneous systems.

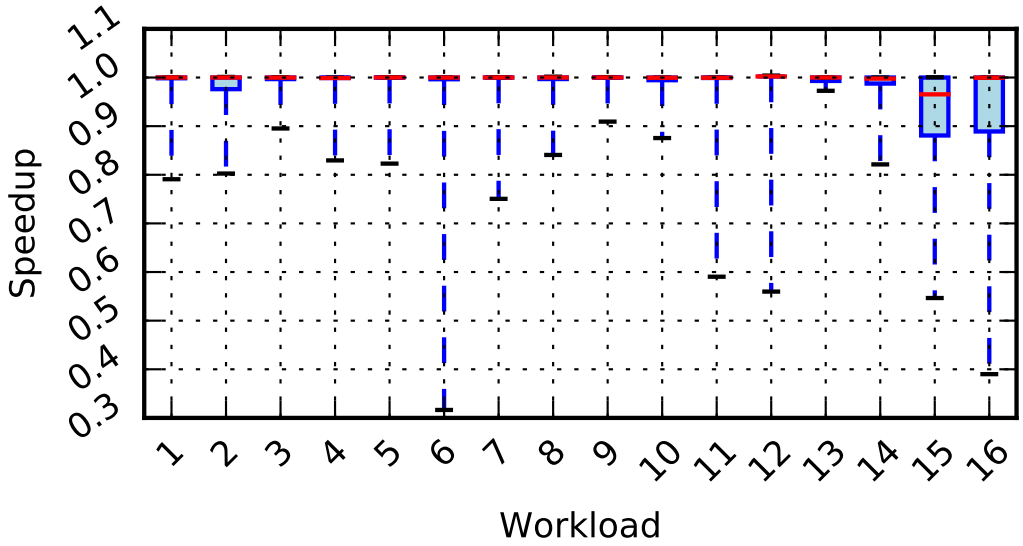


Fig. 5.6 Results with full PreFix. Frequent errors cause significant slowdowns, but median performance shows little impact.

For most workloads, such as 3, 4, & 5, there is little impact from running on a faulty core with the median performance at 1×, and few outliers as shown in Figure 5.6. Some workloads present noticeable degradation in certain error classes for certain benchmark pairs. For example, error class 13 causes significant performance degradation in workload 6, but shows relatively less performance degradation in other benchmark pairs. Also, workload 6 shows substantial resilience to other error classes. This clearly shows that performance degradation is related to the error and benchmark pair.

However, most workloads experience a range of slowdowns depending on the types of faults in the simulated systems. The worst performance is 0.3× on workload 6 which is due to a core with faults exclusively in the integer ALU. In contrast to the workloads that are barely affected, in this case both benchmarks have high baseline IPC. Frequent erroneous instructions reduce the IPC of the first workload (on the faulty core) because it must stall at commit to wait for instruction re-execution. Further, these additional instructions reduce the IPC of the second workload (on the remote core) because it does not get the full fetch capacity and cannot tolerate this reduction in bandwidth.

Figure 5.7 shows the average delay per re-executed instruction in the faulty core, measured using difference in total clock cycles against the error free base case. As is visible in the figure, for some workloads the performance of one of the cores improved due to the slowing down of the other core, especially workloads 1, 3, 5, 7, 8, & 9 which

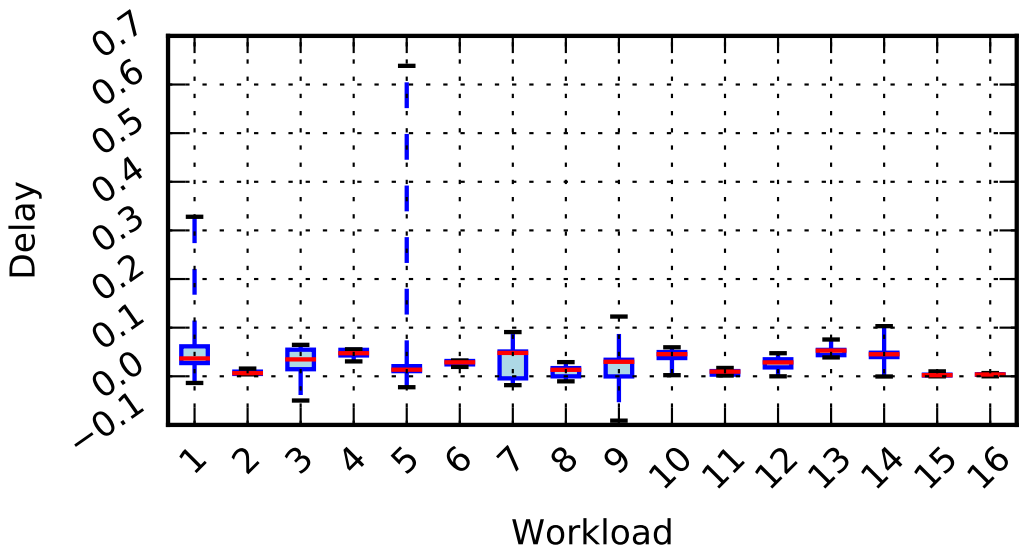


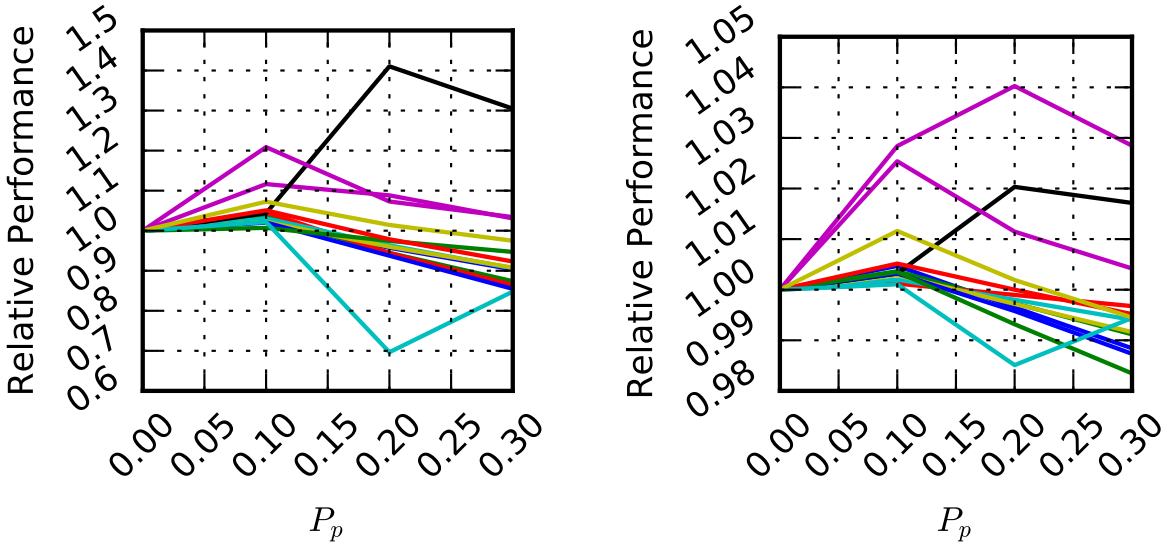
Fig. 5.7 Delay per re-executed instruction

show negative cost per fault in some cases. In these situations, the performance gains from an increased L2 hit rate swamp the overheads from the duplicate instructions.

5.4.2 Impact of prediction

Figure 5.8 shows the impact of prediction on the system, where higher performance is better. Initially there is an improvement in the performance with increasing prediction rates, as expected, but the performance starts to degrade as over-prediction starts increasing the number of instructions classified as HLF. Most workloads have an optimal prediction rate of 0.1 and three benchmark pairings have the optimal prediction rate at 0.2. This further shows that the optimal value of over-prediction is dependent on benchmark pairings, as described in Section 5.2.5. In the experiments shown previously, a prediction rate of 0.2 was used.

Given that the rate of prediction depends on the benchmark-fault pair, we recommend an analysis of the pairing for each core to find the optimal match of core to benchmark. Given the nature of the analysis, it is beyond the scope of the current discussion.



(a) Faulty Core Performance (b) Dual Core Performance

Fig. 5.8 Effect of varying the prediction rate on a 2-core system

5.4.3 Area and power overhead

Using McPAT [53], we obtained area and power estimates for the PreFix framework. It was seen that PreFix has an area overhead of 3.5% on a 2-core machine, and for a 4-core machine, the overhead decreases to 3.1% of the total processor area. The area and power overhead only considers the CPU, within limitations of McPAT. For power, the dynamic power overhead varies from 1-4% based on the application-fault profile. The static power increases by a larger factor of 3% due to the area increase. The total energy per computation increases by an average of 4%, and has a worst case peak of 340%.

Further details on the implementation can be found in Appendix A[3].

5.5 Conclusions

We have presented PreFix, a technique for low-overhead hard-fault tolerance. PreFix uses non-uniform ageing based degradaton and manufacturing process variation to allow processors to continue working by collaborating with each other. The supporting structures are only activated once the first faults are detected, hence do not incurr

[3]Code available at: https://bitbucket.org/jyosoman/mcpat_pfix_repair

any ageing related effects, but fabrication related issues would be present. PreFix would then be able to handle faults as they form. It achieves this by using prediction to identify possibly-faulty instructions, they are then duplicated, and executed on a remote core. PreFix enables faulty components to continue to contribute useful work and prevents errors from propagating outside the core. PreFix has a performance overhead of which is at worst 0.30× that of a healthy system. In most other cases, the overhead is limited to under 85% of a fault-free system. Additionally, it is shown that prediction plays a positive part in the performance, the exact prediction rate depends on the application-fault pairing, hence we recommend placing an analyser to set the prediction rate, this can either be done by static binary analysis [86] or using a just in time compiler [87].

PreFix adds an area overhead of 3.5 percent to the multicore processor it is integrated into. The electric power used by PreFix is 1–4 percent in addition to the power the processor uses. The performance overhead is on average below 5 percent.

Chapter 6

Conclusions

Towards justifying the hypothesis of this dissertation, three major methods have been presented and detailed. Two of the methods handle faults and the third quantifies the effect of the faults. In this chapter, a summary of the work is provided and arguments are presented in justifying how the hypothesis has been satisfied.

Section 6.1 presents a discussion on how the different parts of the thesis work together. Section 6.2 compares the differences between PreFix and REPAIR, and showing the improvements made in performance and additional overheads. Section 6.4 shows the limitations that our methods have and possible methods to fix them. Finally, Section 6.5 presents possible directions to fix the limitations that our methods have and suggests other strategies to keep faulty hardware active.

6.1 Contributions

The primary argument of this dissertation, is that a fault tolerant system can be built over an existing system, which despite using faulty components, can form a fault-safe system. The secondary targets of such a method are performance, area and power. In the rest of this chapter, we refer to these three as the secondary costs. Hence, to satisfy the requirements of this thesis, the fault-tolerant system should be fault-safe while using faulty components and have minimal secondary costs.

6.1.1 REPAIR

This dissertation firstly shows with REPAIR that there is a possibility of such a system. REPAIR is a fault safe design and the secondary costs are reasonable. REPAIR is able to use the external accelerator (IRU) to complete tasks and allow the processor to continue its immediate operations as well. The IRU is also the bottleneck point in the performance, especially in the presence of numerous errors. While functioning, REPAIR holds up the pipeline waiting for re-execution to finish. The re-executed instruction is compared with the original instruction and execution continues. Results showed that the overheads due to re-execution are within practical limits. The performance slowdown is worse than having explicit spares, but the area overhead is significantly lower. REPAIR presented two further directions of work, firstly, the possibility of continuing work on the faulty components within the processors and secondly, having a stronger fault detection mechanism. A more fine-grained understanding of faults and its effects on the output was needed, which FaultSim was useful in.

6.1.2 FaultSim

FaultSim uses higher-level models of transistors and gates, sacrificing analog accuracy (irrelevant for the fault models considered) for speed. A purely digital logic based simulation along with an efficient scheduler makes FaultSim faster than an analog circuit simulator. FaultSim results showed that the frequency of error occurrence, given faults is dependent on the location of the fault and below 10% for most faults in complex circuits. FaultSim used a simple interface to design circuits, and was used to design and simulate over 40 32-bit adders. The performance of FaultSim was better than other fault simulators, and added manageable overheads to an architectural simulator integrated run.

6.1.3 PreFix

Finally, using the directions presented by REPAIR as well as the fault probability models created using FaultSim, PreFix was developed. PreFix presented a simpler method to handle faults by reusing systems already present in the processor. PreFix uses pre-decoders, error checking circuitry, the memory hierarchy and deep pipelines to develop a fault tolerant system. PreFix uses nearby cores to complete the tasks

instead of any specialised hardware. PreFix shows performance degradation which is comparable to elaborate fault-tolerant systems while using substantially less area and power. In comparison to REPAIR, PreFix is able to improve on all the cost parameters, namely performance, area and power. A more detailed discussion follows in the next chapter.

As shown in Chapter 5, PreFix has a small performance overhead without substantially increasing the secondary costs. The hypothesis of this dissertation can be justified through the results presented.

6.2 Comparison between PreFix and REPAIR

PreFix and REPAIR have multiple similarities as well as differences. The two systems are divisible into 4 stages, namely pre-processing (testing if an instruction might use faulty components), scrutiny (observing for usage of faulty components), re-execution (moving data and instruction to a remote execution unit) and correction (comparing local and remote results and updating internal state). Table 6.1 presents a comparison of the two methods.

The performance of PreFix is better than REPAIR and is very close to a fault free core. This is due to the usage of error detection, which reduces the number of re-executions in the processor. The area and power overheads are also smaller. The area overhead is smaller as the different execution units needed for the IRU, take up large space as compared to the control and buffer blocks. Hence, the provision of duplicates is costlier than reusing components already present in the processor. The power overhead also is smaller due to the reduced re-execution rate. Hence, despite the added architectural complexity in PreFix the area, power and performance overheads are lesser compared to REPAIR.

Another difference is in the way the pipeline stalls are handled. In REPAIR, if the front-end of the processor is faulty, instructions using the faulty components are stopped along with all the instructions behind it, and the instructions already in the back-end pipeline are allowed to complete execution. PreFix on the other hand, does not block the pipeline and lets the instructions to pass through. The instruction is checked for correctness at the commit stage of the pipeline.

	REPAIR	PreFix
Pre-processing	Fault map	Fault tree, Holding queue
Scrutiny	Examining usage	Error detection
Re-execution	IRU (Accelerator)	Outward queue, Fault-free core
Correction	Value replacement	Value checking
Pipeline halting	Blocking	Non-blocking
New faults	No protection	Detects, then protects
Performance slowdown (1 faulty core)	0.8×	0.95×
Area Overhead (2 cores)	11%	3.5%
Power	3–15%	1–4%

Table 6.1 Comparison of REPAIR and PreFix.

REPAIR also is not able to handle new faults as its fault map is populated by a periodic BIST, hence new errors can pass through. PreFix on the other hand can tolerate instructions not being marked by its fault-trees as the error detection present in the later stages would be able to detect the occurrence of an error.

6.3 Usage scenario of REPAIR and PreFix

As mentioned in Chapter 2, if different parts of a processor have similar variation profiles, the chances of redundant elements forming permanent faults within a very small time-window of each other is high. In such scenarios, having inactive spares instead of always-on redundant components is useful in elongating the lifetime of the processor. Solutions such as DIVA are not applicable in such scenarios as DIVA has an always-on checker core. The probability of the checker core developing permanent faults within a limited time of the larger core, due to ageing is high. Hence, DIVA would effectively be a faulty core checking a faulty core in real-time. REPAIR is useful in such a scenario, as it has provisions for unused spares whose MTTF would be significantly higher than the MTBF of the core it is protecting. Also, due to variation in the applications run on multiple processors, the ageing profile of any two processors would be different. Such processors can be used to assist each other. PreFix is such an assistive method where partially functioning cores can be used to support each other's

functioning. DIVA would not be able to be of use here as the processor and the checker would run similar workloads and both would likely develop faults in components which perform a similar function.

6.4 Limitations and drawbacks

The methods presented in this dissertation have clear advantages, but the overheads associated, especially area, are still substantial. In REPAIR, the need for having explicit execution units has a large effect on the relative area it uses. If REPAIR supports small cores, which have a large portion of its area dominated by the execution units, the relative area that REPAIR uses would be large. Similarly, in small cores with no pre-decoders, PreFix would have a higher overhead as compared to the out-of-order processor used for comparison.

Power and performance are dynamic measures, depending on the actual usage of the system. As the fault probabilities in hardware are additive, the rate of instruction re-execution rises with the number of faults. Each re-execution increases the power, hence the fault probability directly affects the power used. A positive co-relation also exists between the performance degradation of a system and the fault probability. Performance also depends on the effect slowing down of one application has on the others. Hence, as the fault probability increases, both the power and performance increase. REPAIR has a higher power and performance overhead compared to PreFix, but the overall overheads are high when the number of faults starts to increase.

The work performed towards this dissertation showed that there is an area-power-performance-tolerance trade-off. These methods are suitable for scenarios with low number of faults, which keep the overheads of area, power and performance within 10 percent.

The fault coverage of REPAIR and PreFix depends on the prediction and detection circuitry. Faults that do not fall into the coverage set of either method would cause the fault-tolerance mechanism to fail.

FaultSim also has inherent issues. As no analog simulation is performed, the system in unable to model faults that are caused by frequency, clock, current and voltage variation. Also, FaultSim implements its own circuit description format. Hence, for

every new circuit to be tested, a circuit description suitable for FaultSim has to be created.

6.5 Future work

This dissertation focussed on the ability of a processor to continue operating despite having faults in it. The solution presented here focussed on the processor architecture. The body of work present in literature present fault-safe designs for each level of the computational system, such as operating system, compiler and micro-architectural levels. A cross-level method that does not require fault-secure behaviour from the lower levels, hence creating a system that works together to handle faults, rather than being separately complete is an interesting direction of work. For example, components can be designed so that they provide error detection but with false positives. This would have the advantage of smaller area overhead and place the onus of error-correction and handling false positives to a system such as PreFix.

Another possibility is developing a scheduler that can move applications to appropriate cores that have an error profile (type of faults, and their probability of occurrence) orthogonal to the application requirements, and hence would least affect the performance of the application. The scheduler would not be providing any fault-tolerance, but would improve the performance of the processor. A Just-In-Time (JIT) compilation based method would also be able to provide improvements, where the binary would be re-compiled to change its error profile. The application, so compiled, would cause lesser number of errors, and possibly improving the performance of the processor, while reducing the power used. A method combining a JIT and a scheduler would reduce the power and performance overhead that REPAIR and PreFix have.

FaultSim as presented in this dissertation, does not implement a standard Hardware Definition Language (commonly known as HDL) interface. Building a SystemC interface within FaultSim would allow FaultSim to have better usability and improve SystemC based fault simulations as well.

Appendix A

Guide to McPat usage

McPat version 1.3 is used in this thesis. Given that structural changes were needed, the McPat implementation was modified. In this chapter, we would discuss the basic McPat architecture and where the changes were made. This would additionally act as a resource for anyone making additional changes into McPat on a similar manner.

A.1 McPat Architecture

McPat has three basic blocks, the system components, CACTI for transistor-level simulations and Processor description files. McPat focusses on the on-chip components, but can model off-chip components such as the RAM. System components are the logical blocks of a processor. These include the cores, interconnects, memory controllers, caches, TLB cache, RAM and network interfaces. There is additional support for network-on-chip, PCIe and flash controllers.

An introduction to these blocks can be found in the XML_Parse class. The root_system class encapsulates the various components. Additions to the system can be made through addition of blocks in this class.

The Gem5 to Mcpat script converts the data in the Gem5 stats file into an XML file, reading the implicit dot-separated heirarchy in the stats file. The stats file hence needs to have all the necessary statistics of a given component together (The Gem5toMcPat script does not hold any state other than the current component whose statistics are being read). The XML generated from the conversion script is now in a format readable through Gem5.

The XML parser has structs describing each block of the processor architecture individually. The blocks are initialised used the XML received from Gem5. These configurations are then fed into the processor description files in McPAT, which are a wrapper on top of CACTI.

A.2 Using McPAT

To add REPAIR into McPAT, the appropriate blocks were created. These included structures in the XML processor description files, datastructures in the XML parser and in the system itself. Specifically, the re-execution unit in REPAIR is created, along with the instruction buffers in each processor.

For each logical block, the blocks are either inherited from the implementation of the core, or written from scratch. In each of the cores, the blocks are added. McPAT is not a logical equivalent of an RTL representation, hence there are multiple components that would be presented in an abstract manner, for example, connections and wirings are presented as buffers, the power and area consumption are calculated accordingly. This is due to usage of CACTI, which was not originally intended to be a processor power-area estimator. Our work has done the same as well.

For Prefix, the prefix queues are separately implemented, and the ALUs are expanded by 1.2X, to accommodate for the error-detection overhead. For REPAIR, the IRU is implemented separately. The error map used in REPAIR is implemeted as a buffer in the CPU. As mentioned earlier, the wires are not simulated in the McPAT simulation. Their effect on area and power is encapsulated within the buffers that they connect.

A.3 Code observations

The McPAT code is written using C++0x, and is in need for a proper rewrite caused by unnecessary code bloating. The issues are structural though.

References

[1] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1), 1998.

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5), Oct 1974.

[3] C. Prasad, L. Jiang, D. Singh, M. Agostinelli, C. Auth, P. Bai, T. Eiles, J. Hicks, C. H. Jan, K. Mistry, et al. Self-heat reliability considerations on intel's 22nm tri-gate technology. In *Proceedings of IEEE International Reliability Physics Symposium (IRPS)*. IEEE, 2013.

[4] S. R. Nassif, V. B. Kleeberger, and U. Schlichtmann. Goldilocks failures: Not too soft, not too hard. In *Proceedings of IEEE International Reliability Physics Symposium (IRPS)*. IEEE, 2012.

[5] PassMark Software. Single thread performance. *https://www.cpubenchmark.net/singleThread.html*, 2017.

[6] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of the International Conference on Dependable Systems and Networks,*, 2004.

[7] C. F. Chien, W. C. Wang, and J. C. Cheng. Data mining for yield enhancement in semiconductor manufacturing and an empirical study. *Expert Systems with Applications*, 2007.

[8] Smruti R Sarangi, Brian Greskamp, Radu Teodorescu, Jun Nakano, Abhishek Tiwari, and Josep Torrellas. Varius: A model of process variation and resulting timing

errors for microarchitects. *IEEE Transactions on Semiconductor Manufacturing*, 21(1):3–13, 2008.

[9] Yao Wang, Sorin Cotofana, and Liang Fang. A unified aging model of nbti and hci degradation towards lifetime reliability management for nanoscale mosfet circuits. In *Proceedings of the 2011 IEEE/ACM International Symposium on Nanoscale Architectures*, pages 175–180. IEEE Computer Society, 2011.

[10] Hao Cai, Hervé Petit, and J-F Naviner. Reliability aware design of low power continuous-time sigma–delta modulator. *Microelectronics Reliability*, 51(9-11):1449–1453, 2011.

[11] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109. IEEE, 2010.

[12] Daniel Siewiorek and Robert Swarz. *Reliable Computer Systems: Design and Evaluatuion*. Digital Press, 2017.

[13] L. W. Massengill, B. L. Bhuva, W. T. Holman, M. L. Alles, and T. D. Loveless. Technology scaling and soft error reliability. In *2012 IEEE International Reliability Physics Symposium (IRPS)*, pages 3C.1.1–3C.1.7, April 2012.

[14] Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[15] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *Proceedings of the 32nd International Symposium on Computer Architecture*. IEEE, 2005.

[16] S. Gupta, A. Ansari, S. Feng, and S. Mahlke. StageWeb: Interweaving pipeline stages into a wearout and variation tolerant CMP fabric. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010.

[17] K. Raghavan and V. Kamakoti. ROSY: Recovering processor and memory systems from hard errors. *SIGOPS Operating Systems Review*, 2012.

[18] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering*, (1):23–31, 1987.

[19] N. Binkert, B. Beckmann, G. Black, and Others. The gem5 simulator. *SIGARCH Computer Architecture News*, 2011.

[20] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[21] Liudong Xing, Haoli Li, and Howard E Michel. Fault-tolerance and reliability analysis for wireless sensor networks. *International Journal of Performability Engineering*, 5(5):419–431, 2009.

[22] Joe W McPherson. *Reliability physics and engineering*. Springer, 2010.

[23] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, 1999.

[24] E. Schuchman and T. N. Vijaykumar. Blackjack: Hard error detection with redundant threads on smt. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks,*. IEEE, 2007.

[25] E. Schuchman and T. N. Vijaykumar. Rescue: A microarchitecture for testability and defect tolerance. *ACM SIGARCH Computer Architecture News*, 2005.

[26] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-based online detection of hardware defects mechanisms, architectural support, and evaluation. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007.

[27] M. D Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. Architectural Core Salvaging in a Multi-core Processor for Hard-error Tolerance. In *ACM SIGARCH Computer Architecture News*, 2009.

[28] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke. The stagenet fabric for constructing resilient multicore systems. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2008.

[29] B. F. Romanescu and D. J. Sorin. Core cannibalization architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[30] A. Pellegrini and V. Bertacco. Cobra: A comprehensive bundle-based reliable architecture. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. IEEE, 2013.

[31] A. Ansari, S. Feng, S. Gupta, and S. Mahlke. Necromancer: enhancing system throughput by animating dead cores. In *ACM SIGARCH Computer Architecture News*. ACM, 2010.

[32] A. Tiwari, S. R. Sarangi, and J. Torrellas. ReCycle: Pipeline Adaptation to Tolerate Process Variation. In *ACM SIGARCH Computer Architecture News*, 2007.

[33] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proceedings of the 21st International Conference on Computer Design,*, 2003.

[34] A. Rajendiran, S. Ananthanarayanan, H. D. Patel, M. V. Tripunitara, and S. Garg. Reliable Computing with Ultra-reduced Instruction Set Co-processors. In *Proceedings of the 49th annual design automation conference*, 2012.

[35] A. Meixner and D. J. Sorin. Detouring: Translating software to circumvent hard faults in simple cores. In *Proceedings of IEEE International Conference on Dependable Systems and Networks*, 2008.

[36] O. Khan and S. Kundu. Thread relocation: A runtime architecture for tolerating hard errors in chip multiprocessors. *IEEE Transactions on Computers*, 59(5), 2010.

[37] D. Hardy, I. Sideris, N. Ladas, and Y. Sazeides. The performance vulnerability of architectural and non-architectural arrays to permanent faults. In *Proceedings of the 45th International Symposium on Microarchitecture (MICRO)*, 2012.

[38] M. D. Beaudry. Performance-related reliability measures for computing systems. *IEEE Transactions on Computers*, 1978.

[39] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *ACM SIGARCH Computer Architecture News*, 2011.

[40] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[41] H. Cho, L. Leem, and S. Mitra. ERSA: Error Resilient System Architecture for Probabilistic Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2012.

[42] J. Soman, Negar Miralaei, A. Mycroft, and T. M. Jones. Repair: Hard-error recovery via re-execution. In *Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, Oct 2015.

[43] Rodney M Goodman and Masahiro Sayano. The reliability of semiconductor ram memories with on-chip error-correction coding. *IEEE transactions on information theory*, 37(3):884–896, 1991.

[44] Ulya R Karpuzcu, Brian Greskamp, and Josep Torrellas. The bubblewrap many-core: popping cores for sequential acceleration. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 447–458. IEEE, 2009.

[45] J. L. Hennessy and D. A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[46] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G. Y. Wei, and D. Brooks. Helix-rc: An architecture-compiler co-design for automatic parallelization of irregular programs. *ACM SIGARCH Computer Architecture News*, 2014.

[47] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT 1.0: An Integrated Power, Area, and Timing Modeling Framework for Multicore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[48] S. Knowles. A family of adders. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, 1999.

[49] F. Oboril and M. B. Tahoori. Extratime: Modeling and analysis of wearout due to transistor aging at microarchitecture-level. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.

[50] S. K. Jain and V. D. Agrawal. Stafan: An alternative to fault simulation. *Papers on Twenty-five Years of Electronic Design Automation*, 1988.

[51] B. Srinivasu and K. Sridharan. A transistor-level probabilistic approach for reliability analysis of arithmetic circuits with applications to emerging technologies. *IEEE Transactions on Reliability*, 2017.

[52] M. Abramovici, P. R Menon, and D. T. Miller. Critical path tracing-an alternative to fault simulation. In *Proceedings of the 20th Design Automation Conference*, 1983.

[53] M. L. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari, and S. V. Adve. Accurate microarchitecture-level fault modeling for studying hardware faults. In *Proceedings of 15th IEEE International Symposium on High Performance Computer Architecture*, 2009.

[54] A. Bosio and G. D. Natale. LIFTING: A Flexible Open-Source Fault Simulator. In *Proceedings of the 17th Asian Test Symposium*, 2008.

[55] K. Gulati and S. P. Khatri. Towards acceleration of fault simulation using graphics processing units. In *Proceedings of 45th ACM/IEEE Design Automation Conference*, 2008.

[56] A. Savino, S. D. Carlo, G. Politano, A. Benso, A. Bosio, and G. D. Natale. Statistical reliability estimation of microprocessor-based systems. *IEEE Transactions on Computers*, 2012.

[57] N. Miskov-Zivanov and D. Marculescu. Multiple transient faults in combinational and sequential circuits: A systematic approach. *IEEE Transactions on Computer-Aided Design of ICs and Systems*, 2010.

[58] P. A. Lee and T. Anderson. *Fault tolerance: principles and practice*, volume 3. Springer Science & Business Media, 2012.

[59] S. S. Mukherjee, C. Weaver, et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th annual IEEE/ACM MICRO*, 2003.

[60] M. Wilkening, V. Sridharan, et al. Calculating architectural vulnerability factors for spatial multi-bit transient faults. In *Proceedings of the 47th IEEE/ACM MICRO*, 2014.

[61] K. J. Lee, T. Y. Hsieh, and M. A. Breuer. A novel test methodology based on error-rate to support error-tolerance. In *Proceedings of the IEEE International Conference on Test*, 2005.

[62] R. Brown. Calendar queues: a fast 0 (1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 1988.

[63] D. G. Saab, R. B. Mueller-Titans, D. B. J. A. Abraham, and J. T. Rahmeh. Champ: Concurrent hierarchical and multilevel program for simulation of vlsi circuits. *Urbana*, 51, 1988.

[64] S. Al-Arian and D. Agrawal. Physical failures and fault models of cmos circuits. *IEEE Transactions on Circuits and Systems*, 1987.

[65] S. More et al. Aging degradation and countermeasures in deep-submicrometer analog and mixed signal integrated circuits. *Technischen Universität München, Diss*, 2011.

[66] M. M. K. Martin, D. J. Sorin, et al. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *ACM SIGARCH Computer Architecture News*, 2005.

[67] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI design*, volume 188. Addison-Wesley New York, 1985.

[68] J. L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 2006.

[69] J. L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 2006.

[70] A. Avižienis. Fault-tolerance and Fault-intolerance: Complementary Approaches to Reliable Computing. *ACM SIGPLAN Notices*, 1975.

[71] K. Nepal, N. Alves, J. Dworak, and R. I. Bahar. Using Implications for Online Error Detection. In *Proceedings of the IEEE International Test Conference*, 2008.

[72] D. Gizopoulos, M. Psarakis, and Et al. Architectures for online error detection and recovery in multicore processors. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2011.

[73] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, 1999.

[74] V. K. Reddy, E. Rotenberg, and S. Parthasarathy. Understanding Prediction-based Partial Redundant Threading for Low-overhead, High- Coverage Fault Tolerance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[75] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. *ACM SIGARCH Computer Architecture News*, 2005.

[76] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating hard faults in microprocessor array structures. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 51–60, 2004.

[77] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE micro*, 16, 1996.

[78] D. A. Draper, M. P. Crowley, J. Holst, G. Favor, A. Schoy, A. Ben-Meir, J. Trull, R. Khanna, D. Wendell, R. Krishna, and Others. An X86 microprocessor with multimedia extensions. In *Proceedings of the 43rd IEEE International Solid-State Circuits Conference,*, 1997.

[79] Todd M Austin and Gurindar S Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proceedings of the 28th annual international symposium on Microarchitecture*, 1995.

[80] J. Ayala, M. López-Vallejo, A. Veidenbaum, and C. A. López. Energy aware register file implementation through instruction predecode. In *Proceedings of IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, 2003.

[81] V. Iyengar, K. Chakrabarty, and B. T. Murray. Built-in self testing of sequential circuits using precomputed test sets. In *Proceedings of the 16th IEEE VLSI Test Symposium*, 1998.

[82] M. Nicolaidis. Carry checking/parity prediction adders and ALUs. *IEEE Transactions on Very Large Scale Integrated (VLSI) Systems*, 11(1), 2003.

[83] K. Mohanram and *et al.* Sogomonyan. Synthesis of low-cost parity-based partially self-checking circuits. In *Proceedings of the 9th IEEE On-Line Testing Symposium*, 2003.

[84] S. Mitra and E. J. McCluskey. Which concurrent error detection scheme to choose? In *Test Conference, Proceedings of the International*. IEEE, 2000.

[85] S. Eyerman and L. Eeckhout. Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance. *IEEE Computer Architecture Letters*, PP(99), 2013.

[86] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2004.

[87] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 2003.