Designs for Increasing Reliability While Reducing Energy and Increasing Lifetime

Gulay Yalcin

Department of Computer Architecture

Universitat Politècnica de Catalunya

A dissertation submitted in fulfillment of
the requirements for the degree of

Doctor of Philosophy / Doctor per la UPC

December 2014

Advisor : Osman Unsal
Coadvisor : Adrian Cristal

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Curso académico:

Acta de calificación de tesis doctoral

Nombre y apellidos
____

Programa de doctorado
____

Unidad estructural responsable del programa
____

Resolución del Tribunal

Reunido el Tribunal designado a tal efecto, el doctorando / la doctoranda expone el tema de la su tesis doctoral titulada ______________________________
______________________________.

Acabada la lectura y después de dar respuesta a las cuestiones formuladas por los miembros titulares del tribunal, éste otorga la calificación:

☐ NO APTO ☐ APROBADO ☐ NOTABLE ☐ SOBRESALIENTE

(Nombre, apellidos y firma) Presidente/a	(Nombre, apellidos y firma) Secretario/a

(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal

______________________, _______ de __________________ de _______________

El resultado del escrutinio de los votos emitidos por los miembros titulares del tribunal, efectuado por la Escuela de Doctorado, a instancia de la Comisión de Doctorado de la UPC, otorga la MENCIÓN CUM LAUDE:

☐ SÍ ☐ NO

(Nombre, apellidos y firma) Presidente de la Comisión Permanente de la Escuela de Doctorado	(Nombre, apellidos y firma) Secretaria de la Comisión Permanente de la Escuela de Doctorado

Barcelona a _______ de ____________________ de __________

Acknowledgements

I would like to use this opportunity to express my gratitude to everyone who supported me throughout my PhD studies.

First and foremost, I have to thank my research advisors, Osman Unsal and Adrian Cristal. Without their assistance and dedicated involvement in every step throughout the process, this dissertation would have never been accomplished. I was warned in my first year of PhD that I would hate my advisors before the graduation. I am sincerely grateful to Osman and Adrian that they have proved that this claim has an exception. Both Osman Unsal and Adrian Cristal have always been supportive, showed their confidence, provided the best research environment and been thoughtful to supply opportunities even more than I asked for.

I would also like to show my gratitude to Oguz Ergin. As being my Ms advisor, he was the one who encouraged me to start PhD, to be a researcher in the Computer Architecture area and to study on reliability. He also kindly invited me to a 3-month internship at TOBB University of Economics and Technology. His technical advises and teachings during both Ms studies and the internship have played a major role in taking decisions in critical moments of the research.

In September 2011, I went to Carnegie Mellon University for three months to study with Onur Mutlu and his research group Safari. My time at Safari has been highly productive and working with Onur Mutlu was an extraordinary experience with his always positive attitude and enthusiasm as well as his tough questions improving the immune system of a PhD student. During my internship, I have had the pleasure to collaborate extensively with Yu Cai and I have benefited greatly from his friendship and broad technical knowledge. I would like to thank Onur Mutlu and Yu Cai for being excellent research partners.

I would like to show gratitude to my defense committee, Prof Israel Koren, Vilas Sridharan, Jaume Abella, Ramon Canal and Ferad Zyulkyarov. Despite their incredibly busy schedule, they have been very kind to accept serving in the committee of my PhD defense and/or pre-defense.

Getting through my dissertation required more than academic support, and I have many, many people to thank for listening to and having to tolerate me over those years. I cannot begin to express my gratitude and appreciation for their friendship.

My special thanks go to Ruken Zilan who convinced me that I am qualified to register PhD and that is the logical move to do. Also without her help, I would never survive in a foreign country. I express my warmest thanks to Gokcen Kestor for listening my research problems and giving feedbacks while we have our morning coffees together. Those coffees have been the highest motivation to get up early in the morning. Thanks Roberto Gioiosa, Ilker Demirkol, Justine Demirkol, Nesrin Karavar for the fun moments and interesting conversations. I would also like to thank Sheila Abibula who opened her warm home, helped me pass through hard moments of life and became one of the best friends in Barcelona. Coming to my colleagues from the office, I am thankful for their aspiring guidance, invaluably constructive criticism and friendly advice during my studies. I am sincerely grateful to them for sharing their truthful and illuminating views on a number of issues related to my publications. Starting from Adria Armejach who provided the correct format for writing the thesis, many thanks go to Nehir Sonmez, Srdjan Stipic, Vesna Smiljkovic, Sasa Tomic, Vladimir Gajinov, Oriol Arcas, Behzad Salami, Nikola Markovic, Daniel Nemirovsky, Azam Seyedi, Vasilis Karakostas, Omer Subasi, Ivan Ratkovic, Javier Arias, Milan Stanic, Milovan Djuric, Oscar Palomar, Santhosh Rethinagiri, Tassadaq Hussain, Timothy Hayes and many others.

Most importantly, none of this could have happened without my family; my father Metin Yalcin, my mother Fatma Yalcin, my brother Yunus Yalcin and my sister Gokcen Gonen who happened to be born in a different family but always been more than a sister to me. A successful person should have a force to be pushed and God created you guys since I can not push myself enough. Every time I was ready to quit, you did not let me and I am forever grateful. This dissertation stands as a testament to your unconditional love and encouragement.

Abstract

In the last decades, the computing technology experienced tremendous developments. For instance, transistors' feature size shrank to half at every two years as consistently from the first time Moore stated his law. Consequently, number of transistors and core count per chip doubles at each generation. Similarly, petascale systems that have the capability of processing more than one billion calculation per second have been developed. As a matter of fact, exascale systems are predicted to be available at year 2020.

However, these developments in computer systems face a reliability wall. For instance, transistor feature sizes are getting so small that it becomes easier for high-energy particles to temporarily flip the state of a memory cell from 1-to-0 or 0-to-1. Also, even if we assume that fault-rate per transistor stays constant with scaling, the increase in total transistor and core count per chip will significantly increase the number of faults for future desktop and exascale systems. Moreover, circuit ageing is exacerbated due to increased manufacturing variability and thermal stresses, therefore, lifetime of processor structures are becoming shorter.

On the other side, due to the limited power budget of the computer systems such that mobile devices, it is attractive to scale down the voltage. However, when the voltage level scales to beyond the safe margin especially to the ultra-low level, the error rate increases drastically.

Nevertheless, new memory technologies such as NAND flashes present only limited amount of nominal lifetime, and when they exceed this lifetime, they can not guarantee storing of the data correctly leading to data retention problems.

Due to these issues, reliability became a first-class design constraint for contemporary computing in addition to power and performance. Moreover, reliability even plays increasingly important role when computer systems process sensitive and life-critical information such as health records, financial information, power regulation, transportation, etc.

In this thesis, we present several different reliability designs for detecting and correcting errors occurring in processor pipelines, L1 caches and non-volatile NAND flash memories due to various reasons. We design reliability solutions in order to serve three main purposes. Our first goal is to improve the reliability of computer systems by detecting and correcting random and non-predictable errors such

as bit flips or ageing errors. Second, we aim to reduce the energy consumption of the computer systems by allowing them to operate reliably at ultra-low voltage level. Third, we target to increase the lifetime of new memory technologies by implementing efficient and low-cost reliability schemes.

Contents

List of Figures

List of Tables

1 Introduction

In the last decades, the society benefited immensely from Moore's Law, which states that the number of transistors on chip doubles every two years. However, the computing community faces a reliability wall – similar to memory and power walls of the past – in the sense that the future of semiconductor device scaling is threatened.

The problem has three dimensions. First, transistor feature sizes are getting so small that it becomes easier for high-energy particles to temporarily flip the state of a memory cell from 1-to-0 or 0-to-1. These faults are increasing not only in the probability but also in the impact. For instance, a single such corrupt bit completely collapsed Amazon's S3 cloud computing service [5]. Second, the increase in total transistor and core count per chip will significantly increase faults for desktop and exascale systems even if we assume that fault-rate per transistor stays constant with scaling. If no drastic action to improve reliability is taken, exascale fault rates will be in the orders of minutes, in practice rendering these systems inoperable. We already see warning signs: HP´s ASC Q supercomputer was crashing 15 times a week due to the inability of software and hardware to collaborate in error recovery [5]. Third, increased manufacturing variability and thermal stresses exacerbate circuit ageing; resulting in timing violations in processor structures earlier in their lifetime.

Due to these issues, reliability became one of the first-class design constraint for modern systems in addition to power and performance. Moreover, reliability plays even increasingly important role when computer systems process sensitive and life-critical information such as health records, financial information, power regulation, transportation, etc.

Error Correcting Codes (ECCs) are the most well-known reliability techniques which rely on encoding some information from the stored data and checking this information upon reading the value [6]. They are widely used to detect and correct faults in the memory structures such as caches, main memories, hard disks or non-volatile memories. However, ECC is not adequate to provide reliability for combinational logic (e.g. execution units). Also, ECC may increase the execution time of applications when it is utilized for the time-critical memory components in the microarchitecture (e.g. register file). Thus, other reliability schemes should be provided for processor pipeline structures.

Similar to the ageing problem of CMOS technology, newer non-volatile memory technologies (i.e. Flash Memories, Phase Change Memories, Memristors and Nano-Electromechanical Switches) also suffer from the finite endurance problem. These new memory technologies are proposed as a replacement of CMOS technologies since scaling of the CMOS technology is getting closer to the limits. The current mainstream non-volatile memory is Flash memory. NAND and NOR flash memories are used for quite different applications – data storage for NAND and code storage for NOR flashes. However, as Flash cells scale down to smaller technology nodes, they become increasingly vulnerable to circuit level noise. Thus, fault rate increases exponentially when they are programmed/erased many times (i.e. P/E cycle) and get older. This fault mechanism leads to concerns regarding the reliability and endurance of flash memories. One way to improve flash lifetime is to use stronger error correction codes (ECC) [7]. However, the bit fault rate increases exponentially with P/E cycles while ECC error correction capability increases less than linearly. Thus, techniques that tolerate bit faults in flash cells without relying on stronger ECC are desirable.

On the other side, as the power envelope becomes one of the key design concerns, researchers proposed reducing the voltage to the ultra-low level for microarchitectures and caches. Although voltage downscaling offers substantial energy savings, it also introduces additional reliability challenges due to drastically increasing fault rate. Utilizing ECCs in the low-power operating mode is an appealing and prevalent solution for reducing the safe operating margin for V_{dd} of memory structures. However, ECC schemes may require complex encoders/decoders which may increase the energy consumption and latency of accessing the memory structure which cannot be tolerated in level-1 caches. Thus, to take advantage of potential energy savings, it is essential to design suitable reliability mechanisms for both the microarchitecture and the level-1 cache.

In this thesis, we present reliability designs for detecting and recovering the faults occurring in the microprocessor, in the L1 cache and in the NAND flash memory. While these designs increase the reliability of the computer components, they also provide execution at the ultra-low voltage level to reduce energy consumption as well as increase the lifetime of CMOS technologies and NAND flash memories.

1.1 Problem Statement and Contributions of Thesis

This thesis addresses several issues of reliability that we briefly list in this section. We also present the contributions of this thesis that address these issues.

1.1.1 Error Detection for High Reliability

Executing instruction streams redundantly in chip multiprocessors (CMP) provides the strict reliability requirement of mission-critical systems. However, even if there are enough resources for replication in the system, redundancy-based error detection presents a performance overhead every time it is triggered for the comparison of execution results in order to detect a divergent execution. Moreover, if error-detection is frequently triggered, the possibility that a benign fault cause an error recovery increases.

In this thesis, we propose utilizing transactional semantics of Hardware Transactional Memory (HTM) in order to defer error detection until a transaction commits so that the cost of error detection can be reduced while its efficiency can be increased. Transactions record their tentative reads and writes in a read-set and write-set respectively. HTM systems already have well-defined comparison mechanisms of read-/write-sets in order to detect if there is any conflict between transactions. While comparison of addresses is sufficient for conflict detection, some systems also send data along with addresses. We adapt these already existing conflict detection mechanisms for error detection. We build error detection on top of an HTM featuring lazy conflict detection and lazy data versioning. The main advantage of using lazy-lazy HTM is that comparison of write-sets for error detection reduces the comparison overhead of redundancy-based fault detection due to multiple writes to the same address.

1.1.2 Error Recovery

A reliable system requires a simple recovery mechanism to roll back to an error-free state after detecting an error. Checkpointing is a well-known error recovery scheme which would present scalability issues as we move towards many-core systems. This drawback of checkpointing is due to the required synchronization mechanisms to guarantee that all structures (e.g. cores) rollback to the same state in case of an error. Also, when an error is detected in one core, all the other cores communicating with it (faulty or not) have to roll back since errors could have propagated to the error-free cores through shared variables. Moreover, in addition to the performance degradations, checkpointing schemes require supplementary hardware structures (e.g., buffers to save checkpoints) which are non-functional for performance, but they are only utilized for reliability. These structures increase system verification and test complexity. Thus, most of the academic reliability proposals have not been implemented in real hardware. Although software-based reliability schemes have been proposed in order to avoid new hardware design, these schemes present high overhead in the execution time, and they require the recompilation of the system software or the application code.

In this thesis, we propose leveraging Hardware Transactional Memory (HTM) for low-cost error recovery which provides mechanisms to roll back to an earlier state in the execution for resolving conflicts occurring during the parallel execution. In HTM, transaction-start can be viewed as a checkpointed stable state and provides a simple recovery mechanism. We minimize the implementation complexity of error recovery by utilizing existing hardware.

1.1.3 Executing at Ultra-Low Voltage Level

As the power envelope becomes one of the key design concerns, the dramatic improvement in the energy efficiency in computer systems is required in order to keep the power under control. A very effective approach in reducing the energy consumption is to reduce the supply voltage (V_{dd}) below the safe margin. Voltage downscaling can offer substantial energy savings by trading off performance. However, the energy reduction in the low-power mode comes with a drastic increase in the number of faults [8] both in combinational logic and memory cells. In order to fully exploit the dynamic energy savings of voltage downscaling, a potentially attractive idea is to implement reliability solutions that allow the system to operate below the safe margin of V_{dd}.

In this thesis, we handled the microarchitecture and L1 cache separately for the operations below the safe margin of V_{dd}. For microarchitectures, we investigate the usefulness of TM-based error detection schemes such as replication, encoded processing, symptom-based and invariants for reducing energy consumption. For L1 caches, we propose two schemes: 1) We Adopt a Single Error Correcting - Multiple Adjacent Error Correcting Error Correcting Code [9] in the faulty lines of L1 caches for below safe voltage operations. 2) We present a simple, circuit-driven solution that duplicates or triplicates all the available lines in the cache and achieves each read or write access to multiple lines without increasing access latency.

1.1.4 Limited Endurance of NAND Flash Memories

A flash memory cell has limited endurance; i.e. data cannot be reprogrammed into the cell more than a limited number of times. A single-level flash cell (SLC) can tolerate ∼10k program/erase (P/E) cycles while a 2-bit, multi-level cell (MLC) can only survive for ∼3k P/E cycles for 30-40nm (i.e., 3x-nm) technology generations. The available P/E cycles are expected to decrease even more as flash cells continue to scale down in size and more than 2 bits are programmed per cell. Generally, storage systems have strict requirements on reliability. For example, the uncorrectable bit error rate during usage should be less than 10^{-15} and stored data should be available for 5-10 years [10]. Thus, it is a challenge for flash memories to satisfy the lifetime requirements for enterprise Solid State Disks (SSDs).

In this thesis, we exploit the dominance and characteristics of retention errors and programming errors. In order to mitigate retention errors, we introduced Flash Correct-and-Refresh scheme that periodically reads, corrects, and reprograms or remaps the stored data. In order to mitigate programming errors, we introduced Neighbor-cell Assisted Correction scheme in which data is re-read by utilizing the information extracted from the data stored in neighbor cells.

1.2 Thesis Organization

Chapter 2 presents the nomenclature of faults in a computer system and the state-of-the art for error detection and recovery schemes.

Chapter 3 introduces FIMSIM, a fault injection infrastructure for microarchitectural simulators.
Chapter 4 introduces SymptomTM, an architectural reliability scheme which leverages Hardware Transactional Memory (HTM) mechanisms for error recovery and detects errors by monitoring error symptoms.
Chapter 5 presents FaulTM, a redundancy-based error detection and recovery proposal based on Hardware Transactional Memory (HTM) providing high reliability for mission-critical systems.
Chapter 6 seeks architectural solutions for energy minimization by executing at ultra-low voltage level.
Chapter 7 presents on-chip SRAM memory designs which can tolerate very high bit failure rates of ultra-low voltage execution in L1 caches.
Chapter 8 presents flash data correct techniques which increase the lifetime of flash memories significantly.
Chapter 9 concludes this dissertation.

2
Background in Faults and Reliability

In this chapter, we present the nomenclature of faults in computer system, and the state-of-the art for error detection and recovery schemes.

2.1 Faults in Computer Systems

In computer system, a hardware defect is termed as a *fault*. Errors are the manifestation of faults. This means that an error is caused by faults but not all faults lead to errors. Also, fault within a particular scope (i.e. circuit, architecture, operating system) may not appear as an error outside the scope if the fault is either masked or tolerated within the scope.

Faults experienced by semiconductor devices fall into three main categories: transient, intermittent and permanent. Moreover, when these faults affect more than a bit at a time, multi-bit faults occur. In this section we explain these faults.

Besides this classification, faults are also classified according to their outcomes [11]. If a fault disappears or is masked without being noticed by the user, it is termed as *benign*. Obviously, if the fault is not benign, it leads to an error unless it is detected and corrected. If an error is not caught by the system, it leads to a Silent Data Corruption (SDC). In another case, an error can be caught by the system but it can not be corrected. In this case, the error is termed as Detected Unrecoverable Error (DUE). Mission-critical systems such as airplanes must have extremely low SDC and DUE because people's lives may be at stake. Both SDC and DUE lead to an error showing up at a boundary where it becomes visible to the user such as a change in the bank account with or without warning. A failure, simply a special case of an error, is defined as a system malfunction that causes the system to not meet its correctness, performance, or other guarantees.

2.1.1 Transient Faults

A *transient fault* (also known as Soft Error: a transient fault cause a soft error) is a bit flip due to some radiation event or power supply noise.

Three key radiation mechanisms inducing transient fault in computer systems have been discovered: (1) Alpha particles from packaging material, (2) high-energy neutrons from cosmic radiation and (3) low-energy cosmic neutron interactions with the isotope brone-10. Obviously, these radiation events are unpredictable and it is not easy to avoid from them. Energetic particles (e.g. alpha or neutron particles) generate electron–hole pairs (directly or indirectly) as they pass through a semiconductor device. Transistor source and diffusion nodes can collect these charges. The state of a logic device (i.e. latch, static random access memory (SRAM) cell, or gate) is inverted if a sufficient amount of charge accumulates, thereby a logical fault is introduced into the circuit's operation. These faults are temporal (transient) since the data bit stored in a device is corrupted until new data is written to that device [12]. As transistor dimensions and operating voltages shrink, sensitivity to radiation increases dramatically. Thus, it is foreseen that future systems will be more prone to transient faults.

Despite the fact that transient faults are nondestructive functional errors and they can be fixed by re-setting or re-writing of the device, they may cause dramatic impact on computer systems unless they are mitigated [13]. Here we present examples from the reported past events that are due to transient faults.

- In 1978-1979, Intel Corporation delayed in delivering its chips to AT& T due to soft error problem. The problem was because the packaging modules were made by a new ceramic factory and in the factory these modules got contaminated with uranium from the water of Colorado's Green River that passes through an old uranium mine [14].

- In 1986-1987, IBM Corporation faced a similar radioactive contamination problem. The source of the problem was that bottles used to store the acid required in the chip manufacturing process was cleaned by using a radioactive contaminant [15].
- In 1996, Normand studied the error logs of several large computer systems and reported a number of incidents induced by cosmic ray strikes [16].
- In 2000, Sun Microsystems observed soft error phenomenon in their UltraSPARC-II-based servers, where the error protection scheme implemented was insufficient to handle soft errors occurring in the SRAM chips in the systems.
- In 2004, Cypress semiconductor reported a number of incidents arising due to soft errors [15]. In one incident, a single soft error crashed an interleaved system farm. In another incident, a single soft error brought a billion-dollar automotive factory to halt every month.
- In 2005,Hewlett-Packard acknowledged that a large installed base of a 2048-CPU server system in Los Alamos National Laboratory which is located at about 7000 feet above sea level, crashed frequently because of cosmic ray strikes to its parity-protected cache tag array [17]. It is reported that HP's ASC Q supercomputer was crashing 15 times a week due to the inability of software and hardware to collaborate in fault recovery.
- In 2008, a single corrupt bit completely collapsed Amazon's S3 cloud computing service which took 6 hours to recover the entire system [5].

2.1.2 Permanent Faults

Irreversible physical changes in the semiconductor devices are called *permanent faults*. Permanent faults tend to occur early in the processor lifetime due to manufacturing faults (called "infant mortality"), or late in the lifetime due to thermal and process related stress. Thus they are typically characterized by the classic bathtub curve, shown in Figure 2.1. Initially, the error rate is typically high because of either bugs in the system or latent hardware defects. Beyond

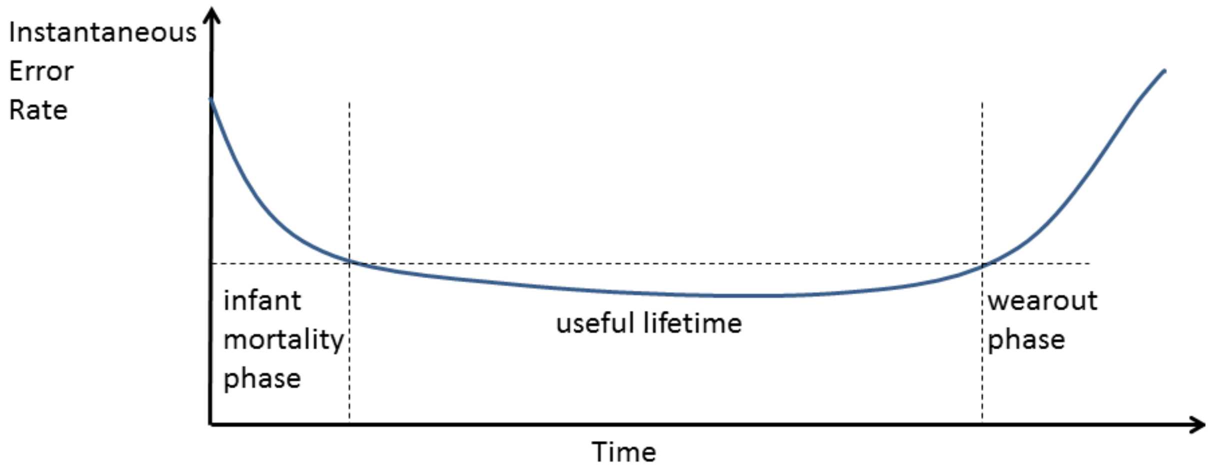


Figure 2.1: Bathtub Curve: The fault rate in the phases of infant mortality, useful lifetime and wearout phases.

the infant mortality phase, a system typically works properly until the end of its useful lifetime is reached. Then, the wearout accelerates causing significantly higher error rates.

Reliability mechanisms usually disconnect the faulty structures hit by permanent faults, and replace them with fault-free spare structures. Systems having these mechanisms tolerate permanent faults. In fact, the lifetime reliability of a system is defined by its ability to tolerate these faults.

The silicon industry typically uses a technique called burn-in to pass the infant mortality phase and move the starting use point of a chip to the beginning of the useful lifetime period Burn-in removes any chips that fail initially, thereby leaving parts that can last through the useful lifetime period. Further, the silicon industry designs technology parameters, such as oxide thickness, to guarantee that most chips last a minimal lifetime period.

Faults in the wearout phase are caused by electro-migration, stress-migration, gate oxide breakdown or thermal cycling. *Electromigration* is the current-induced atomic transport generated by collision of electrons with metal atoms. The depletion and accumulation of material creates voids and hillocks which can lead to open and short faults respectively. *Stress migration* is caused by thermo-mechanical stress which are caused by differing thermal expansion rate of different materials in the device. So that metal atoms in the interconnects migrate due to mechanical stress. *Gate oxide breaks down* with time and fails when a conductive path forms in the dielectric. In *thermal cycling*, permanent damage accumulates every time there is a cycle in temperature in the processor, eventually leading to failure.

A permanent fault can be detected by performing built-in self test (BIST) [18]. For instance, to check that in a memory structure if there is a permanently faulty bit producing always ‘0’or always ‘1’(i.e. stuck-at-zero or stuck-at-1), first ‘0’s are written to the memory structure and read back to see if they are read correctly. Then the process is repeated with writing all ‘1’s to the memory structure.

Typically faults in the wearout epoch are manifested first as intermittent faults (we explain in the next section) then progress to permanent faults.

2.1.3 Intermittent Faults

Process variation or in-progress wear-out, combined with voltage and temperature fluctuations cause burst of frequent faults, called *intermittent faults*, that last from several cycles to several seconds. An intermittent fault occurs repeatedly at the same location; It tends to occur in bursts for a period of time when the fault is activated and the replacement of the offending circuit mitigates the intermittent fault [19, 20]. It has been suggested that intermittent faults have the potential to impact program execution to a greater extent when compared with transient faults [21]. In this sense, intermittent faults can be considered as similar to permanent faults. However, similarly to transient faults, it is hard to diagnose an intermittent fault by post facto using hardware/software tests because intermittent faults do not persist and the conditions that caused the fault are hard to regenerate.

Continued device scaling results in increased Process, Voltage and Temperature (PVT) variations, increased cross-talk and decreased noise margins all of which lead to increased susceptibility to intermittent faults. Moreover, wear-out failures are expected to become much more frequent but devices typically do not fail suddenly, they display intermittent behaviour for a period of time beforehand. Therefore, it is prevised that the rate of occurrence of the intermittent faults will increase [22, 23].

2.1.4 Multi-bit Faults

Multi-bit faults occur when hardware faults affect multiple bits at a time due to several reasons. In this section we explain a couple of them. Multi-Bit Upsets can be transient or permanent.

Spatial multi-bit upsets of transient faults occur when a single particle strike causes more than one bit-flip in the neighbourhood. Results of irradiation tests on 90nm commercial processes reveal that multi-bit upsets as large as 13 bits can occur in sub-100nm technologies [24]. These faults are expected to increase in the future processors due to shrinking size of the transistors. A two-bit spatial multi-bit upset in a memory structure can manifest in two ways: horizontal or vertical [25]. Horizontal means two adjacent bits on the same word are upset. Vertical, on the other hand, means that two bits in two adjacent words but in the same bit position are upset.

Temporal multi-bit upsets of transient faults happen when multiple independent particles strike distinct locations of the structure causing upsets on multiple bits. Since the likelihood of particle strike is high in the high altitudes, the probability of temporal multi-bit upsets increases in higher altitudes [26].

Bridging permanent faults are caused by shorts between normally unconnected signal lines. There are two types of bridging faults: (1) dominant-1 which is modelled by assuming that there is an AND gate between bits and (2) dominant-0 which is modelled by using an OR-gate. Note that, if there is a short between two bits, only the value of one bit changes. However, we classify these bridging faults under multiple faults since it is related to multiple bits. Also, this short can be between more than two bits which leads to changes in the values of multiple bits. In bridging faults, when two shorted lines have the same value, the result comes the same. On the other side, when two shorted lines have opposite driven values, one value (the strong one, either 0 or 1) overrides the other.

2.2 Reliability Schemes

In order to characterize the behaviour of a system in the presence of a fault, two attributes are defined: *Reliability* and *Availability*. The reliability of a system is the probability that the system does not experience a user-visible error. Availability, on the other hand, is the probability that a system is functioning correctly at a particular time. *Fault-Tolerance* is the ability of a computer system to survive in the presence of faults. In the literature, Fault Tolerance and Reliability are tend to be used interchangeably. A reliable system should have two main aspects in order to avoid errors: Error Detection and Error Recovery. *Error Detection* is the process of discovering that an error has occurred. *Error Recovery* is the process of restoring the system's integrity after the occurrence of an error.

Some reliability mechanisms just detect errors and alert the Operating System (OS) or the user (the so-called fail-stop systems) while other mechanisms both detect and recover from errors (the so-called fail-safe systems).

One of the simplest ways of error detection in memory structures is using parity bits. A parity bit is added to a string of a binary code and it indicates whether the number of bits with the value of '1' is even or odd. Single bit parity can be used to detect odd number of errors. However, it is not adequate to correct errors or to detect even number of errors.

Error Correcting Codes (ECCs) are the most well-known reliability techniques rely on encoding some information from the stored data and checking this information upon reading the value [6]. They are widely used to detect and correct faults in the memory structures such as caches, main memories, hard disks or non-volatile memories. However, ECC is not adequate to provide reliability for microarchitectures and execution units.

In this section, we cover error detection and recovery schemes providing reliability for microarchitectures.

2.2.1 Error Detection

Redundant execution is the most common error detection solution utilized for detecting faults occurred in microprocessors including execution unit and other data-path components. The value is generated multiple times with a single or multiple resources and later checked with simple voting. To protect processor logic from transient faults, some studies utilize Simultaneous Multi-threading (SMT) by executing two identical threads in the same core and comparing their results [27, 28, 29]. However, they are not suitable to detect permanent faults. In recent work, researchers leveraged chip multiprocessing (CMP) for error detection by pairing cores for redundant execution and checking their results [30, 31, 32, 33].

Lockstepping is a classical, redundancy-based error detection scheme for microprocessors that is widely used by systems integrators [32, 33]. In lockstepping, two synchronized and lockstepped processors run two identical instruction streams by receiving the same input. The output signals from processors are compared in each cycle to check for faults. When a mismatch is detected in the outputs, it triggers a relevant action, such as alerting the OS in a fail-stop system or triggering hardware or software recovery actions in a fail-safe system.

Lockstepping can be deployed at different granularities. For instance, in the fine-grained approach, output of each instruction or even the output of each pipeline stage is compared.

Although the error detection occurs immediately when the execution of the faulty instruction completes, the overhead of this approach across threads is significantly high. On the other side, in the coarse-grained approach, only the off-core activities such as the result of the memory instructions are compared by considering that the fault is benign unless it goes out of the scope. In a coarser approach such that in HP non-stop servers [33], full boards are replicated and only off-board activities are compared. Trivially, the error detection overhead of coarse-grained approaches are much lower than the fine-grained ones. On the other side, error detection latency is much higher in coarse-grained approach compared to the fine-grained one. In order to provide early error detection in a coarse-grained lockstepping system, Hernandez and Abella proposed LiVe in which register values are exposed in the network for comparison in the round-robin fashion [34]. Lockstep systems can be implemented for mixed-critical systems where core may run reliability-critical instruction streams in lockstep mode while non-critical ones are executed in non-lockstep mode. For instance, Infineon AURIX implements 3 core processor in which two of them are lockstepped by comparing off-core activities while one of them is non-lockstep [35].

Lockstepping can reduce the undetected Silent Data Corruption rate to almost zero for components it is covering, i.e. microprocessors. However, it can also increase the rate of the false positive case especially when it is implemented with fine granularity in which error recovery is triggered although the detected fault is benign. For instance, a fault in the branch predictor do not usually cause incorrect execution, it only slightly impacts the performance due to the misprediction, thus it is generally benign. However, two lockstepped processors do not follow the same execution path although the final output of the execution is the same, hence, they consider the fault as an error. On the other side, Lockstepping requires tightly coupled processor pairs driven by the same clock signal. Driving the clock signal across cores becomes an increasing burden as device scaling continues. Later studies avoid this burden by focusing on two main issues: (1) input replication and (2) output comparison

Input replication ensures that both processors observe identical load values, cache invalidations and external interrupts. Chip-level Redundant Threading (CRT) [31] and Chip-level Redundant Threading with Recovery (CRTR) [30] utilize load-value queue (LVQ) to provide an identical view of memory to the redundant executions. However, this strict input replication requires significant changes to the critical components of the processor core and cache hierarchy. Also, it forbids using the cache hierarchy for multi threaded executions. ReUnion [36] relaxes the input replication by allowing redundant cores to issue memory operations independently. It solves incoherant inputs of redundant threads by rolling back the threads if their outputs are different. However, ReUnion is not convenient for permanent fault detection, since it can not differentiate if two successive mismatches between redundant executions are caused by input incoherence or a permanent fault. Note that, ReUnion utilizes the rollback mechanism of reorder buffer (ROB) to recover from detected faults which requires architectural register file to be protected by ECC with encoding/decoding overhead at every access.

Output comparison checks the result of redundant executions to determine if they are identical. CRT [31] validates only the memory values assuming that a fault is benign unless it propagates to the memory. However, full-state comparison is essential to guarantee the error-free execution since the last validation. For this issue, CRTR [30] compares the results of register instructions together with memory values which makes the comparison overhead very high. It reduces the bandwidth requirement by employing dependence-based checking elision which lets only the last instruction in the queue to use the register value queue. However, it still has a high comparison overhead. Fingerprinting [37] strives to compare the result of all instructions with a very low comparison overhead by producing the signature of execution history. However, it has two main drawbacks. Firstly, it is highly likely that some faults are not detected in the summary value. Thus, the fault coverage of Fingerprinting is not 100%. The consequence of undetected faults could be catastrophic in mission-critical domain. Secondly, it treats benign faults and errors equally. which presents unnecessary recovery time overhead of benign faults (In our experiments in Chapter 5, we find that 19% of transient faults for spec cpu2006 applications are benign and 20% of them are treated as error by Fingerprinting). Moreover, it necessitates hash value circuits in the architecture which increase the power consumption and necessitate additional pipeline stages [36]. Note that, ReUnion [36] utilizes efficient compression through Fingerprinting for output comparison.

In order to avoid the redundancy overhead, researchers propose other error detection schemes such as online testing or symptom-based error detection. ACE [38] is one of the latest on-line testing schemes which stops the execution of the system periodically in order to execute predefined test vectors to check if the system produces the expected results. However, on-line testing schemes can not detect transient faults. Symptom-based error detection schemes [39, 40], which monitor error symptoms (e.g. fatal traps, miss-predictions) for error detection in order to avoid redundancy. However, their error coverage is limited which causes higher Silent Data Corruption (SDC) rate [41]. For instance, a fault may cause an erroneous amount of money to be transferred to a bank account. Due to this limitation, Shoestring [41] partially replicates the instruction stream to reduce the SDC rate.

2.2.2 Error Recovery

There are mainly two categories of recovery mechanisms: 1) Forward Error Recovery (FER) and Backward Error Recovery (BER). FER is based on replicating the execution in order to use the correct results if the actual execution fails. This approach assumes that the replicated execution is error-free. Hence, we do not consider further this mechanism in this study. BER (also called checkpoint/rollback mechanism) stores an error-free state of the system (checkpoint) and reverts the system state upon error detection (rollback). BER is classified in three groups according to the checkpointing strategy used.

2.2.3 Global Checkpointing

Global checkpointing, a well-known recovery scheme, recovers detected errors by rolling back all processors to an earlier validated state [42, 43]. The scalability of this approach is limited when we move towards many-core systems, as it introduces the significant overheads due to two reasons: (1) during barrier synchronization performed at checkpointing, some processors might stay idle if load is not properly balanced between them (e.g., some processors perform I/O operations before the checkpoint), (2) the recovery requires all communicating processors to rollback to an earlier validated state, which causes unnecessary rollbacks of the error-free processors.

2.2.4 Coordinated-local Checkpointing

The overheads of global checkpointing are mitigated by synchronizing only the set of processors that have communicated with each other between two checkpoints to decide on a common checkpoint, whereas all other processors can perform local checkpoints [44, 45]. This approach has been shown to outperform global checkpointing [46]. ReVive [43] and SafetyNet [47] are well-known coordinated-local checkpointing schemes that create system-wide checkpoints periodically. ReVive is only feasible in coarse granularity due to its large checkpoint size. However, I/O operations can only be supported in small checkpointing intervals [37]. Also, large checkpoints suffer from long recovery times. These schemes present several difficulties. First, they typically implement relatively complex synchronization mechanisms to guarantee that all structures (e.g. cores) rollback to the same state in case of an error. For instance, in SafetyNet, late synchronization causes several unvalidated checkpoints to be saved in the system which leads to an area overhead. Second, these schemes rollback all processors during recovery which causes a loss of error-free operations. In order to address this limitation, Rebound [46], the state-of-the-art architectural checkpointing scheme, rollbacks only the communicating processors after error detection instead of all processors. However, it still may require more than one processor to rollback and it suffers from the problem of cascading rollbacks.

2.2.5 Uncoordinated-local Checkpointing

In contrast to the two previous approaches, uncoordinated-local checkpointing performs checkpointing locally at each processor without any synchronization and also stores the interactions between processors in order to rollback to a consistent checkpoint [48, 49]. This approach is interesting for executions where processors communicate rarely.

3 FIMSIM: Fault Injection Infrastructure

Technology trends are leading to more hardware errors (transient or permanent errors) due to various phenomena such as high energy particle strikes, ageing or wear-out, infant mortality, and so on. Thus, Reliability is becoming a first class design constraint for computer designers. However, reliability techniques introduce penalties in performance, in power, in die size or in design time. Therefore, it is essential to carefully evaluate the level of reliability provided by these schemes.

Fault injection is a widely used experiment-based reliability evaluation approach in which faults are injected either (1) to the real hardware, (2) to the simulator or (3) to the software (operating system or application). While hardware fault injection requires at least the physical prototype of the system, software fault injection is limited in the sense that they cannot inject faults into locations that are inaccessible to the software. On the other hand, simulation based fault injection is applicable early in the design time and it can inject faults to the processor structures that cannot be excited by injecting faults at the software level. Nevertheless, it provides the correlation between the criticality of circuit level faults and their impact on the application level. Consequently, simulation based fault injection is more appealing for researchers than software and hardware based fault injections.

Several simulation based fault injectors model the Register Transfer Level (RTL) of microprocessors [50, 51]. These models are incapable of modelling a wide range of systems (e.g. multi core multi threaded architectures) due to their design complexity and their high simulation time. Moreover, they are impractical for the researchers who design reliability in the microarchitecture level since they use microarchitectural simulators for their design. In this chapter, we present FIMSIM, a fault injection infrastructure for microarchitectural simulators.

Reliability schemes generally targets particular fault models. FIMSIM is capable of injecting transient, permanent and intermittent faults either as a single-bit fault or multi-bit faults as combination of several fault models. Hence, FIMSIM is convenient for the evaluation of many dependable systems. On the other hand, faults in small-sized structures (bypass logic, PC) may lead to drastic errors despite the fact that the likelihood of the fault occurrence in a bigger-sized structure (e.g. register file) is higher. FIMSIM, apart from the prior fault injectors, injects faults also to critical small sized structures besides other large buffered structures. Inevitably, the sizes of structures are taken into account by FIMSIM for the calculation of the reliability of the entire microarchitecture. Consequently, the compact characteristic of FIMSIM provides opportunity to make a comprehensive evaluation of the vulnerability of different microarchitectural structures against different fault models.

Simulation time is a prominent limitation for the fault injection technique. FIMSIM, reduces the simulation time due to two reasons. First, FIMSIM is on microarchitecture level simulation which keeps the simulation time shorter compared to RTL level simulations. Second, we utilize M5 [52], a microarchitecture simulator that provides a checkpointing mechanism for the implementation of FIMSIM. Therefore, numerous fault injections, especially in the late phase of the applications, are achieved in shorter amount of time by restoring the checkpoints in the later phase without the need to run the application from the beginning.

Repeatability of fault injection experiments (injecting the exact fault again) is also an essential issue for validation of dependability since, compelling test cases can be determined and prepared. FIMSIM is able to repeat a fault injection experiment easily. In FIMSIM, the user can define the fault injection point explicitly instead of injecting faults to randomly generated points.

In Section 3.1, we explain the design principles of FIMSIM. In Section 3.2, we present our fault injection result evaluating the vulnerability of the in-order Alpha [53] microarchitecture by utilizing spec cpu2006 [54] benchmark suit.

3.1 Implementation

In this section, we describe the infrastructure of the FIMSIM tool and its capabilities. First, we define the simulator that we enhanced with fault injection capability. After that, we describe the aspects of a golden run. Then, we explain the fault injection implementation. Finally, we present the calculation of the processor reliability according to the fault injection results.

3.1.1 Simulator

We enhance the M5 full system simulator [52] with fault injection capability. M5 includes processor cores, peripheral devices, memories, interconnection buses, and network connections. The timing model of M5 ensures that all operations (i.e. branch prediction, cache misses, fetches, pipeline stalls, thread context switching, and many other subtle aspects of microprocessors) are executed in the proper virtual time. M5 is an execution-driven simulator which takes the binary file of applications to be simulated. We select M5 for our base simulator due to its several properties that are convenient for a fault injection tool. First, M5 is a full system simulator that we can observe the effects of hardware faults on the operating system and user applications. Second, M5 has a checkpointing mechanism that dump the whole inner-state of the architecture to a checkpoint file whenever it is desired. In FIMSIM, we can trace the effect of a fault by comparing checkpoints easily. Moreover, we can accelerate repetitive fault injections to the late phase in applications by restoring the checkpoints without requiring the execution until that point. Note that M5 is a deterministic simulator that the results do not vary at all between two identical runs at different times. Third, in M5, we can add command line options easily by modifying Python scripts. This is beneficial because we can define the fault injection and the golden run options without modifying the simulation fundamentals. Fourth, the M5 is implemented in an object oriented programming manner so that we can enhance the simulator with fault injection capabilities without modifying the rest of the simulation dramatically. Fifth, M5 is a popular open-source simulator with a large user-base, therefore it is a good substrate for FIMSIM; especially with a view towards releasing FIMSIM to the reliability research community.

Figure 3.1 presents the fault injection classes that we added to the M5 simulator and how these fault injection classes interact with the existing classes in M5. The white area was already implemented by M5, we add the shaded classes: *FaultList* and *Fault*.

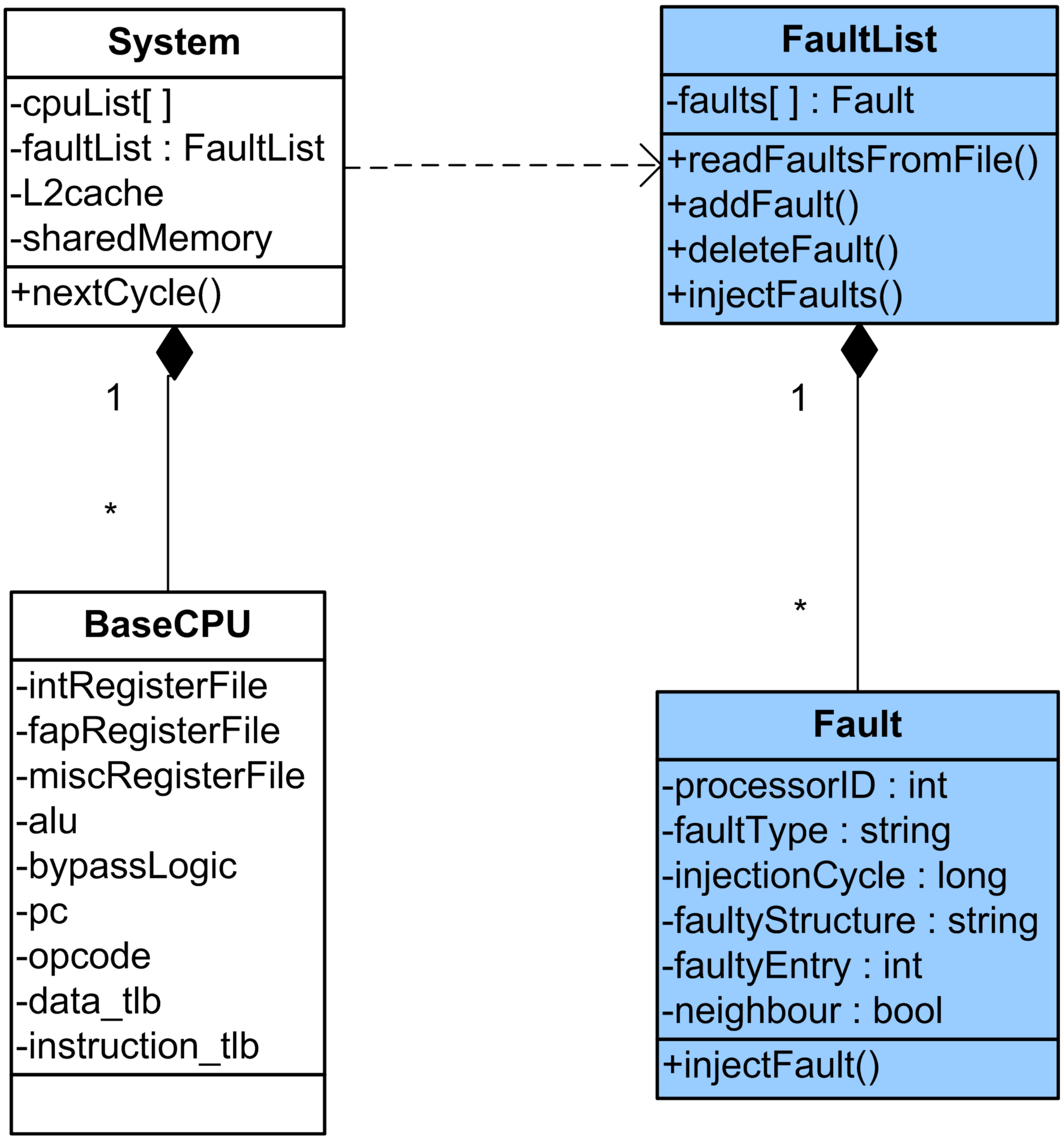


Figure 3.1: Class Diagram of FIMSIM. Blue classes are the extensions for the fault injection purpose.

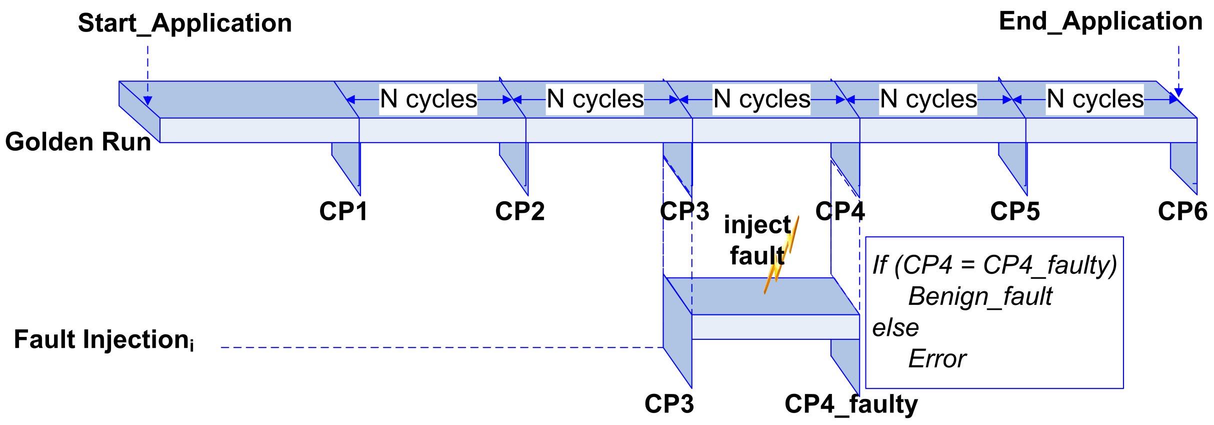


Figure 3.2: Checkpoint generation in golden run

3.1.2 Golden Run

In the golden run which is executed only once for each application, FIMSIM produces error free checkpoints periodically during the execution of the entire application. For example, in Figure 3.2, FIMSIM generates checkpoints at every N cycles (e.g. N = 10M), so that 6 checkpoints are generated for the whole application. For instance, to inject faults to the 4th chunk of the application, CP3 is loaded to the FIMSIM simulator without having to execute the application from the beginning. After the fault injection, CP4 of the golden run and the faulty run are compared without waiting until the end of the application in order to see whether the fault has completely disappeared (masked) or it still stays in the architecture. We argue that instead of comparing the final results of the application, comparing the architectural states is essential, because a fault may stay in the system or worse propagate to the operating system although it does not affect the final output of the application.

3.1.3 Fault Injection

In FIMSIM, we simulate transient faults by flipping (changing 0 to 1 or vice versa) the value of the randomly selected bit in a randomly selected cycle. We use stuck-at-0 and stuck-at-1 fault models to simulate the permanent faults in which a randomly selected value becomes stuck from the fault injection cycle until simulation ends. We simulate intermittent faults by utilizing stuck-at fault models for a predefined number of cycles.

In FIMSIM, the user defines the list of fault(s) in the input file (e.g. a single permanent fault and/or spatial multi-bit transient faults). In this file, the user defines each fault with the following properties:

- the processor id where the fault will be injected (0..(number of processor-1)),
- fault type (transient, stuckat0, stuckat1, dominant0, dominant1, intermittent0, intermittent1),
- fast-forward cycle before fault injection
- the cycle of the fault injection (0..(total Execution Cycle -1)),
- the faulty structure (intRF, specialRF, ALU, ITB, DTB, Bypass, PC),
- the faulty entry (e.g. register number),
- the faulty bit number (0..31 for 32 bit machine).
- persisting time of the intermittent faults,
- neighbour fault id for multi-bit fault injections (0..(number of faults-1)).
- direction of neighbourhood for spatial faults (vertical, horizontal)

These properties are also shown in Figure 3.1 under the class *Fault*. These parameters explicitly define a fault so that the user can repeat a fault injection experiment for debugging the effect of a particular fault definition. Note that, the user can prefer any/all of these parameters to be random so that multiple fault injection campaigns could be conducted and their results processed to calculate overall processor reliability.

At the beginning of the simulation, FIMSIM reads the list of faults from an input file and it sets the list defined by the `FaultList` data structure. Making a list of faults gives an opportunity to add multiple bit faults at a time. Some of these multi-bit faults affect the neighbour bits as well. In FIMSIM, these faults are described with the neighbour attribute to define the place of the second bit. For example, a spatial multi bit fault is defined as two faults that are neighbours to each other. In this case, the second fault takes the place of the fault from the first fault's options and it is located just next to it.

After starting the simulation, at every cycle (e.g. in `nextCycle()` method), M5 calls the `FaultList`'s `injectFaults()` method which also calls the `injectFault()` method of each fault. `injectFault()` method first compares the injection cycle of the fault and the executing cycle of the simulator. If the executing cycle is late enough, FIMSIM modifies the corresponding value in the pertinent structure according to the fault type of the fault. For instance, for a transient fault injection to the register file, FIMSIM flips the corresponding bit in the register file. After the injection, if the injected fault is transient, it is deleted from the fault list. Otherwise (permanent and intermittent), it keeps being injected in the following cycles either until the end of the simulation (permanent) or until the persisting time of the fault (intermittent).

In each fault injection, the checkpoint is created in the following checkpoint creation cycle unless the fault causes the application to crash. This faulty checkpoint is compared with the one produced in the golden run to see whether the fault is masked or it has changed the state of the microarchitecture. Thus, there are three possible outputs of the fault injection:

- **Crash:** The application crashes before the simulation reaches to the checkpoint creation cycle therefore it does not generate a checkpoint. This crash might be due to a segmentation fault or a fatal trap exception (e.g. incorrect program counter).

- **Benign:** If the checkpoints of the faulty run and the golden run are identical, that means that the injected fault was masked (e.g. a faulty register is written before it is read by any instruction)

- **Error:** If the faulty checkpoint is different from the golden one, that means that the fault led to a different architectural state. This error group also includes the silent data corruptions (SDC).

Here one can think that classifying faults in the checkpoints instead of waiting until the end of the application would not present an accurate result. In this sense, faults classified as errors at the checkpoint time may be masked or may lead to a crash after the checkpoint. However, Li et al. showed that for the faults causing a fatal trap exception after executing 10M instruction is less than the 5% of the injected faults. Similarly, we argue that some errors may stay in the architectural state although they do not change the output of the application. Thus, we believe that checkpoint comparison provides a fast and sufficiently accurate classification within the defined tracing window.

3.1.4 Reliability of Processor

When calculating the vulnerability of whole architecture, there are two essential points that need to be focused.

(1) the results of the faults (e.g. system crash or SDC)

(2) the likelihood of the fault occurrence in the structure (e.g. size of the structure)

We classify the faults into three groups according to the way they manifest themselves: benign faults, catastrophic failures and errors (including SDCs). In result critical applications (e.g. financial applications), SDCs are the most critical faults for reliability in which the user must completely trust the result computed by the application (e.g. transferring the correct amount of money). On the other hand, some other applications are required to operate continuously (e.g. web servers). In these applications, catastrophic failures are more critical than SDCs. Thus, when FIMSIM calculates the vulnerability of the entire architecture, it multiplies the error rate and the catastrophic failure rate with C_e , C_c coefficients respectively. The weight of these coefficients are application dependent and defined by the user.

There are several aspects that affect the likelihood of the fault occurrence (LF) in a structure such as size, temperature, age, environmental conditions etc. In this study we consider two aspects. First, we give the higher weight to bigger structures according to the number of wires in the structures. Second, for the combinational logic (e.g. ALU) we multiply the likelihood with the probability that a fault injected to the inner state of the structure propagates to the output of the combinational logic without being masked inside the structure. We adopt these probabilities from a previous study of RTL analysis of the processor structures [55]. Note that, for more accurate LF calculation, other aspects can be considered as well.

We calculate the processor vulnerability (V) with the following formula:

$$V = \sum_{i}^{structures} \left(\left(C_e \text{ x } ER_i + C_c \text{ x } CR_i \right) \text{x} LF_i \right)$$

$$C_e = Error\ Coefficient$$

$$C_c = Crash\ Coefficient$$

$$ER_i = \frac{Number\ of\ Errors}{Number\ of\ Injected\ Faults} \qquad : ErrorRate$$

$$CR_i = \frac{Number\ of\ Crashes}{Number\ of\ Injected\ Faults} \qquad : CrashRate$$

$$LF_i = SR_i \text{ x } OR_i \qquad : Likelihood\ of\ Fault\ Occurrence$$

$SR_i = \frac{Size\ of\ theStructure}{Size\ of\ the\ Entire\ Processor}$

$OR_i = The\ probability\ of\ fault\ propagation\ to\ the\ output$

Note that FIMSIM is flexible enough so that users can also generate their own processor dependability models.

3.2 Evaluation

In this section, we evaluate the vulnerability of in-order Alpha 21264 microarchitecture by utilizing our FIMSIM fault injection infrastructure and spec cpu2006 benchmark suite with test data set. We inject the faults to seven different structures in the core; bypass logic, data TLB (DTB), instruction TLB (ITB), arithmetic logic unit (ALU), integer register file (int-RF), special purposed register file (RF-special) and program counter (PC). Note that, in-order cores do not have some complex structures required for out-of-order execution such as the reorder buffer, issue queue and rename logic. We did not inject faults to caches such as L1 and L2 caches. It is because due to their large sizes, they require higher number of fault injection experiments and longer time to simulate those experiments. We inject 100 faults per structure in each application to a random location in the structure (e.g a random bit of a randomly chosen register in the int-RF). In each fault injection, we start the error-free application, execute until the randomly chosen fault injection cycle and continue executing until the next checkpoint creation time without any other fault injection. Each of experimental results presented in this work represent the result of 20 applications from spec cpu2006 benchmark with 2000 fault injections per structure. In total, we injected nine different fault models (i.e. transient, stuck-at-0, stuck-at-1, intermittent-0, intermittent-1, multi bit transient horizontal, multi-bit transient vertical, dominant-0, dominant-1) to 7 different structures (i.e. bypass, DTB, ITB, ALU, int-RF, RF-special, PC), thus, in total we run 126000 fault injection experiments which is similar or very higher than prior fault injection analysis [55, 56, 57, 58].

First, we generate checkpoints of the golden run at every 100M cycles after warming up 70M cycles. Then we inject faults at a random time (within 100M cycles) by loading the checkpoints by performing one injection per simulation. We injected faults to the first chunk since some applications in the spec cpu2006 benchmark suite does not have more than 200M instructions with test dataset. We calculate the vulnerability of the processor (V) by utilizing the formula that we explain in Section 3.1.4. The size values of each structure and other related

Structure	SR	OR
Program Counter	0,003	1
Special Register File	0,4	1
Integer Register File	0,4	1
Instruction TLB	0,05	1
Data TLB	0,05	1
Arithmetic Logic Unit	0,08	0,7
ByPass Logic	0,003	1
	C_c	C_e
	1,0	1,0

Table 3.1: Number of Bits in Microarchitecture Structures and their average temperature.

coefficients that we utilized in the formula, are presented in Table 3.1.

Figure 3.3 presents the single fault injection results. Unsurprisingly, processors are vulnerable to permanent faults at most, as it is seen in the figure. The interesting result is that stuck-at-1 faults are more harmful for the applications than stuck-at-0 faults. This is because bit values are mostly zero (e.g. more than 70% of bits in PC and more than 90% of bits in special register file are zero). Thus, stuck-at-0 faults (permanent or intermittent) are generally benign.

When we compare the effects of the faults on ALU and Bypass logic, the bypass logic is slightly more vulnerable to the faults. This is because the fault affects the next instruction in the bypass logic. In TLBs (ITB or DTB), short term faults (transient or intermittent) are compensated. However, when there is a permanent fault in TLB, it is harmful for the whole architecture. Finally, the PC is the most vulnerable structure in these structure that any fault in the PC result in either error or system crash.

In Figure 3.4, we present the effects of multiple faults on the processor. Note that we can inject the vertical multi-bit faults only to the buffer structures (i.e. register files and TLBs). In the figure, it is seen that the multi-bit transient faults in the horizontal direction are not more harmful than a single bit transient fault. However, in the vertical direction, multi-bit transient faults become significantly more harmful in the register files since it affects more than one entry in the buffer. Bridging faults (dominant-0 and dominant-1) affecting the vulnerability in the similar way. Because the final result of the bit values changes if two bits are different from each other meaning that they are effective in the same conditions.

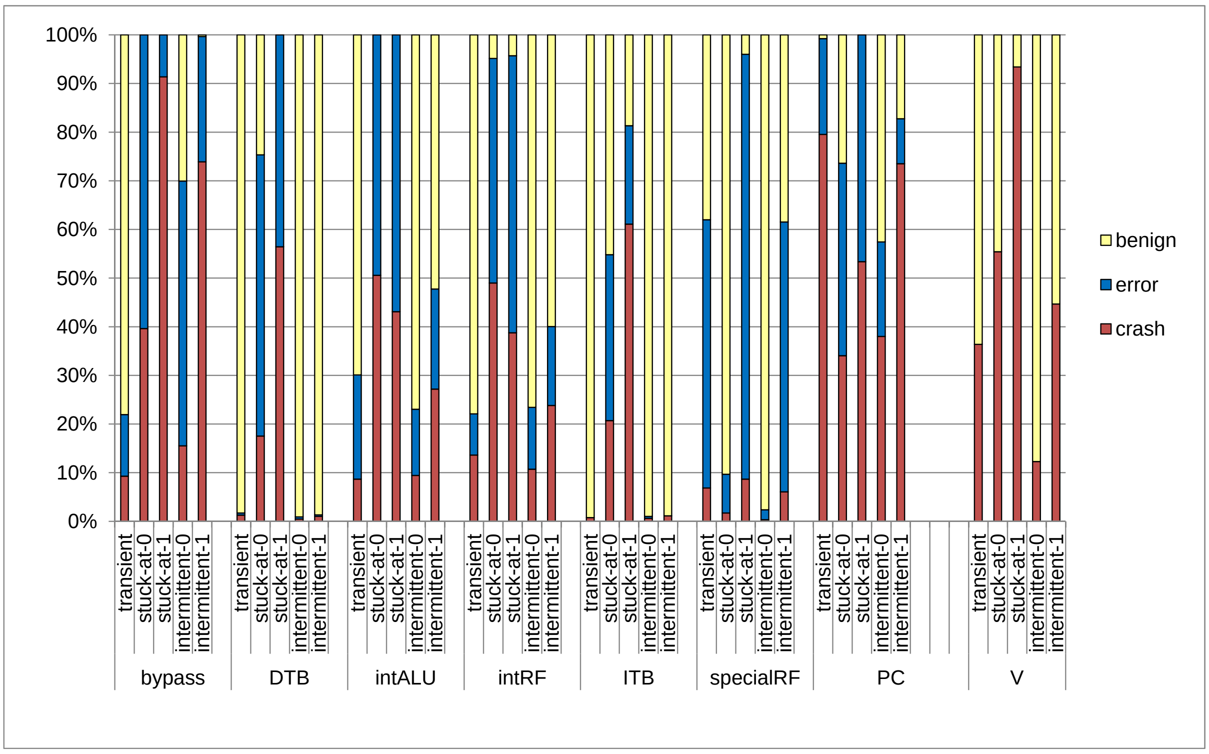


Figure 3.3: Single Fault Injection to Spec2006 Benchmark.

3.3 Related Work

Many fault tolerant system design have been proposed by the increasing importance of the reliability since scaling of new technologies leads to reliability issues. Architectural Vulnerability Factor (AVF) is one direction to measure the reliability of a microarchutecture which is proposed by Shubu Mukherje [59]. The main advantage of AVF analyse is that it is fast to consulate in the early in the design time without needing the hardware prototype. Although AVF analysis can be used to estimate the vulnerability of the system against single-bit transient faults, it is not adequate to be used for multi-bit transient faults or intermittent and permanent faults. Thus, AVF only gives the lower bound for reliability [58].

Another well known reliability measurement technique is fault injection which is being increasingly consolidated and applied a wide range of fields. Besides supplying an infrastructure to measure the reliability performance of a computer system, fault injection schemes can also

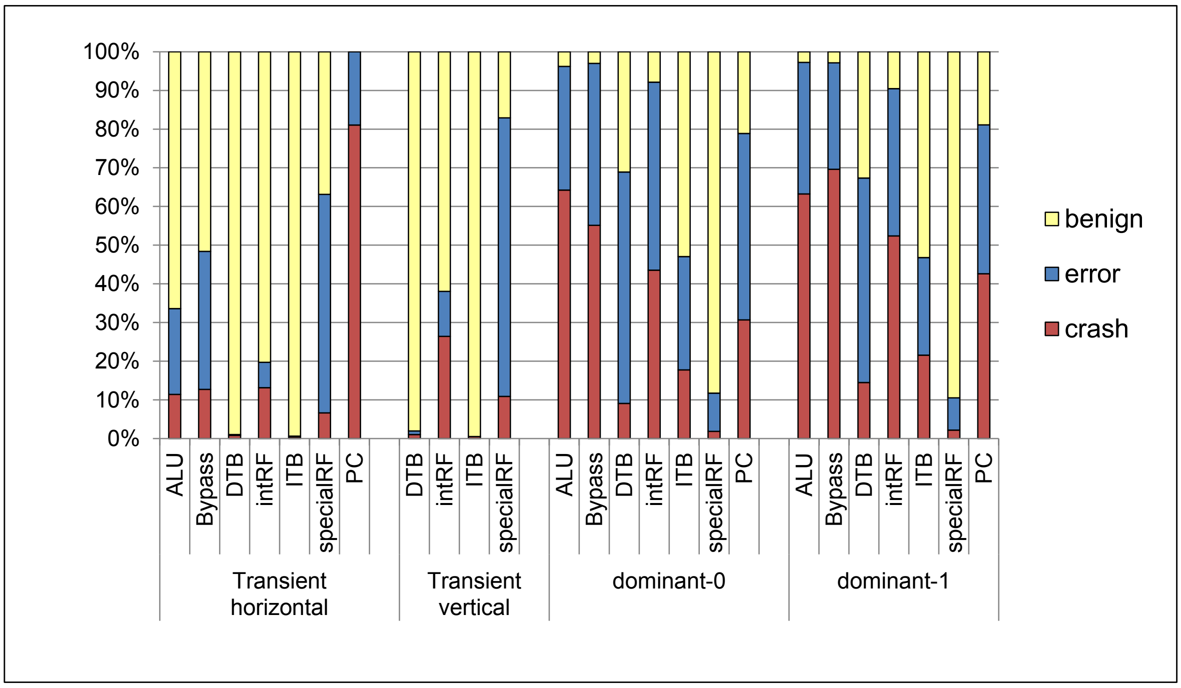


Figure 3.4: Multi-bit Fault Injection to spec2006 benchmark.

help to measure the error recovery time of a reliability scheme. In this section, we present previous fault injectors implemented in simulators.

Many simulation based fault injectors simulate the models in Very high speed integrated circuits Hardware Description Language (VHDL). Gil et. al. [50] studied and compared these fault injection methods. Nicholas J. Wang et. al. [51] implemented The Illinois Verilog Model (IVM) for injecting transient faults and characterizing the effects on a high performance processor pipeline. They created a highly detailed RTL model of microprocessor. The results are traced by using uniform sampling. Later in another study, Nicholas J. Wang et. al. [58] injected transient faults to IVM and they compared fault injection results with AVF analyse.

Although low-level simulators would provide the ability to use more accurate fault models, they present a trade-off in speed and the ability to model long running workloads with OS activity. Given our emphasis on understanding the impact of faults on the OS and the need to simulate for long periods, gate level simulation is not feasible

Michail Maniatakos et. al. [60] pinpointed some limitations of IVM (e.g. can not implement the floating point instructions, inefficient to inject permanent or intermittent faults). They developed an extensive fault simulation infrastructure by interacting IVM with a functional simulator (simplescalar). Also, Man Lap Li et. al. [55] combined microarchitecture simulation with RTL level fault injectors to inject permanent faults to ALU, Decoder and Address Generation Unit (AGEN) to gain accurate fault injection results. However, these schemes are complex and slow for microarchitecture designers since they include RTL model.

Nishant J. George et. al. [25] injected single and double transient faults due to single particle strike. They injected faults to the register file and to the reorder buffer by using a functional simulator (PtlSim). The effects of the injected faults are classified according to the final results of the applications. Therefore, they could make their experiments only on short applications due to time constraints. Also, they can not distinguish whether a fault is benign or hidden in the architecture.

Man-Lap Li et. al. [57] injected permanent faults to a microarchitectur level full system simulation environment (GEMS+Virtutech Simics). They injected both single (stuck-at) and double (bridging) faults. Also, they pointed out the advantages of fault injection into microarchitecture level simulator (e.g. presenting a trade-off in speed and the ability to model long running workloads with OS activity).

So far, very few tries have been done in order to study the effects of intermittent faults by fault injection. Layali Rashid [21] injected intermittent faults in software level to understand their affects on the software. However, they made their experiments only on two applications (matrix multiply and insertion sort). Gracia et. al. injected intermittent faults in a VHDL based simulator [61].

3.4 Summary

Fault injection is a widely-used experiment-based reliability evaluation approach in which faults are injected either (1) to the real hardware, (2) to the simulator or (3) to the software (operating system or application). While hardware fault injection requires at least the physical prototype of the system, software fault injection is limited in the sense that they cannot inject faults into locations that are inaccessible to the software. On the other hand, simulation based fault injection is applicable early in the design time and it can inject faults to the processor structures that cannot be excited by injecting faults at the software level.

In this chapter, we present FIMSIM, a fault injection infrastructure in the microarchitecture level. FIMSIM is a compact tool that can inject transient, permanent or intermittent faults either as a single bit fault or as multi-bit faults. Therefore, it provides opportunity to analyse the affects of the faults on the software since it is on a full-system simulator. In this chapter, we also evaluate the vulnerability of in-order Alpha microarchitecture by using FIMSIM.

4 SymptomTM: Symptom-Based Error Detection with Transactional Memory

It is important to develop simple (in terms of complexity-effectiveness) and powerful (in terms of error detection and recovery coverage) reliability and availability mechanisms to address the increasing number of transient and permanent faults. In order to provide low-cost error detection, Symptom-Based error detection has been proposed which monitors the execution to determine if there is a symptom of hardware error such as fatal traps, high number of OS activities, mispredictions in high confidential branches or too many misses to caches [39, 40]. Although these symptom-based error detection schemes provide acceptable fault coverage (except Silent Data Corruptions) with minimum performance degradation, they require a simple recovery mechanism to roll back to an error-free state after detecting an error [62].

Error recovery is the process of restoring the system's integrity after the occurrence of an error. One of the well-known error recovery scheme is global checkpointing [43, 47]. However, as we move towards many-core systems, it is foreseen that global checkpointing will not be scalable due to two reasons. First, global checkpointing schemes should implement synchronization mechanisms (either at checkpoint creation or checkpoint validation) to guarantee that all structures (e.g. cores) rollback to the same state in case of an error. Second, when an error is detected in one core, all the other cores communicating with it (faulty or not) have to roll back since errors could have propagated to the error-free cores through shared variables. Thus, assuming that error rate will be higher for higher core counts, it is foreseen that in future many-core processors, global checkpointing may take even more time than the execution of the application itself [63].

In addition to the performance degradation in the error-free execution, recovery schemes require supplementary hardware structures (e.g buffers to save checkpoints) which are non-functional for performance but they are only utilized for reliability. These structures increase system verification and test complexity. Thus, most of the academic reliability proposals have not been implemented in real hardware. Although software based reliability schemes have been proposed in order to avoid new hardware design, these schemes present high overhead in the execution time and they require the recompilation of the system software or the application code.

In this chapter, we introduce SymptomTM, an architectural reliability scheme which leverages Hardware Transactional Memory (HTM) mechanisms for error recovery and detects errors by monitoring error symptoms. HTM systems provide mechanisms to abort transactions in case of a conflict, thus they discard or undo all the tentative memory updates and restart the execution from the beginning of the transaction. Thus, transaction start can be viewed as a checkpointed stable state. SymptomTM uses the TM abort mechanisms for error recovery. SymptomTM executes vulnerable code in a special transaction and it monitors if this transaction presents an error symptom. In case of an error symptom, SymptomTM aborts the transaction to recover from the error. Thus, SymptomTM provides a simple recovery mechanism.

We designate fatal traps (e.g attempting to execute an undefined instruction code) as our error symptom which is examined by SWAT group for permanent faults in detail [39, 62, 64]. They conclude that it has high error coverage with no false positive impact. Although other symptoms (e.g. mispredictions in high confidence branches or high OS activity) can be considered to improve the error coverage, these symptoms may result in false positives that increase the performance degradation.

In Section 4.1, we present a background information for Transactional Memory. In Section 4.2, we explain the design principles of SymptomTM with its benefits. In Section 4.3, we provide our experimental results for SymptomTM.

4.1 Background about Transactional Memory

Transactional Memory (TM) is a promising technique which aims to simplify parallel programming by executing transactions (sequences of instructions) atomically and in isolation [65, 66]. Atomicity means that all the instructions in a transaction either commit as a whole, or abort and roll back their changes. When a transaction commits, its tentative updates are made permanent. Transactions record their tentative reads and writes in a read-set and write-set respectively. TM provides mechanisms to abort transactions in case of a conflict. In order to abort a transaction, TM systems discard or undo all the tentative memory updates and restart the execution from the beginning of the transaction. Thus, a transaction's start can be viewed as a checkpointed state.

TM systems can be implemented in the software, in the hardware or in a hybrid fashion. Hardware Transactional Memory is already implemented in mainstream processors and available from large system integrators [67, 68].

All TM proposals implement three key mechanisms: data versioning, conflict detection and conflict resolution.

Data versioning manages all the writes inside transactions until transactions successfully commit or abort. It can be implemented in eager or lazy manner. Eager data versioning systems put the new versions of the data values in-place and keep the old versions of the data values in an auxiliary structure. On the contrary, in lazy data versioning systems, old values are kept in-place while new values are stored in separate buffers. In both data versioning system, in transactional aborts, old values are restored while in transactional commit new values are made visible to the entire system.

Conflict detection tracks addresses of transactional reads and writes to identify concurrent accesses that violate consistency. Conflict detection can also be implemented in lazy or eager manner. In the eager manner, every memory accesses are inspected while in lazy manner conflict detection is deferred until the end of the transaction.

Conflict resolution aborts one or more transactions to resolve conflicts.

We believe that building reliable systems on HTM systems is an appealing approach. First, the hardware structures required for reliability are also implemented in HTM systems for optimistic concurrency. For instance, HTM systems provide mechanisms to abort transactions in case of a conflict, thus they discard or undo all the tentative memory updates and restart the execution from the beginning of the transaction. Thus, a transaction's start can be viewed as

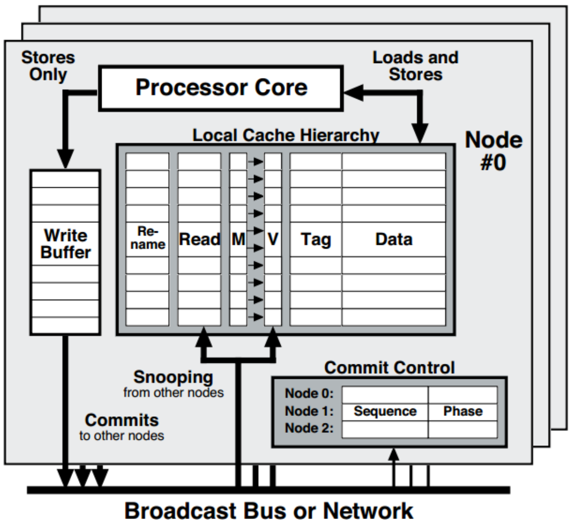


Figure 4.1: Sample TCC Hardware, the figure is taken from [1]

a locally checkpointed stable state. Second, a reliable system should ensure that faulty tasks do not negatively affect other tasks in the system. Hence, it should provide a failure isolation which is not easy to achieve since tasks need to communicate. TM executes transactions atomically and in isolation which also supports the isolation of failures.

One of the main challenges in TM is how to cope with external actions such as system calls or I/O operations. Note that external operations are an issue for reliable systems as well and they are mostly deferred after validating that all operations are error-free. Besides external actions, TM systems have inefficiency at executing large transactions. However, when transactions are not used for concurrency control (i.e. reliability purposed transactions), transaction demarcation can be changed and these two disadvantages of TM can be eliminated for reliable systems. Thus, the size of reliability-purposed transactions can be limited and those transactions can be committed before system calls and I/O operations wait too much to be handled.

We build SymptomTM on top of an HTM system that features lazy conflict detection and lazy data versioning. We present the main hardware components of Transactional Coherence and Consistency system, a conventional lazy-lazy HTM system, in Figure 4.1 .

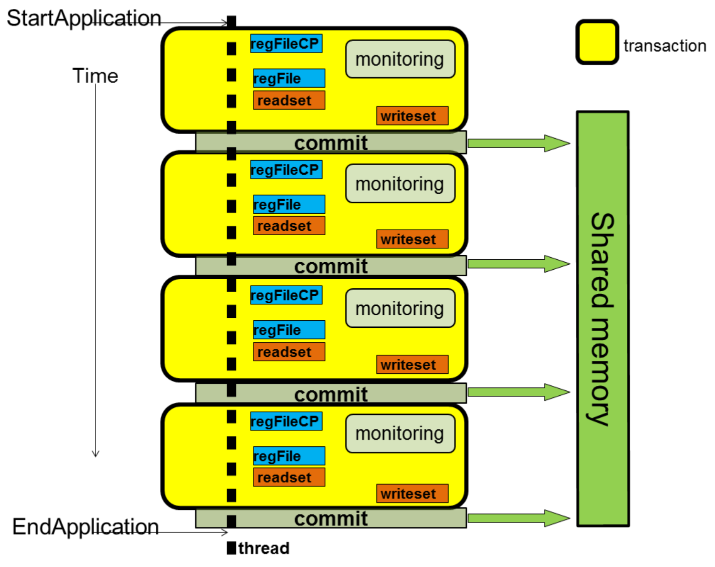


Figure 4.2: Design of SymptomTM

4.2 Basic Design of SymptomTM

We build SymptomTM on top of an HTM system that features lazy conflict detection and lazy data versioning. The SymptomTM system starts a single special transaction at the beginning of the application (see Figure 4.2). Here, the applications requiring to be executed reliably can be annotated by the programmer. The transactions are started by the hardware automatically. From now on, we call this transaction as `availableTX`. Note that, in lazy-lazy HTM, creating a transaction means starting to write the values to the local buffer area instead of shared memory, thus it does not present a transaction creation overhead. This special transaction is executed atomically and isolated from the system until commit. Hence, possible error does not propagate out of the availableTX until commit. Register file is also checkpointed at the beginning of the availableTX as in transactional memory which would be used for error recovery if necessary. In SymptomTM, applications are executed in back-to-back availableTXs. Also, the execution of the availableTX is monitored to detect if there is any symptom of hardware errors, in this case fatal traps (e.g., undefined opcode). Unless any fatal trap exception is raised in the availableTX, the write-set is committed to the shared memory at the end of the availableTX. Commit process starts whenever write-set size equals to the transactional log size.

As illustrated in Figure 4.3, if a symptom is detected, the availableTX aborts and restarts the execution from the beginning of the availableTX. If there is no symptom at the end of the second

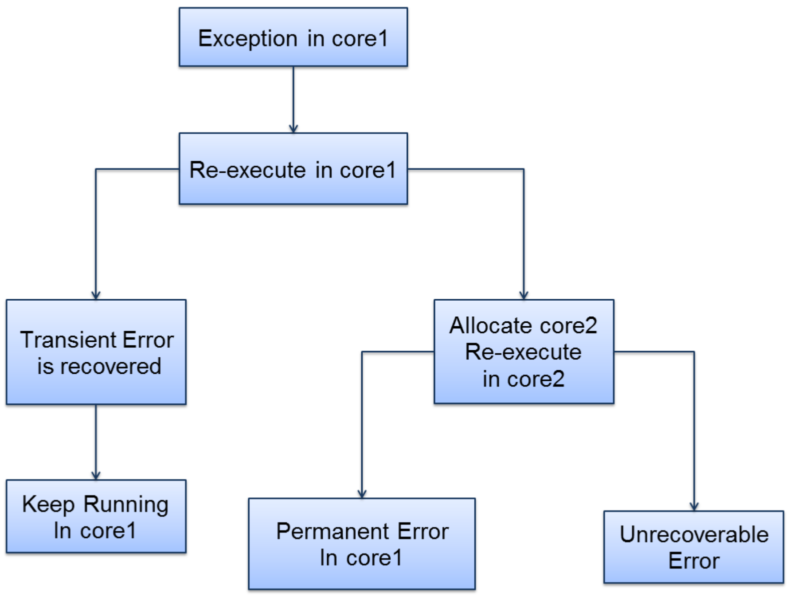


Figure 4.3: Error Detection Algorithm of SymptomTM

restarted execution, that means that the error was transient and that it was corrected. If the second execution raises the fatal trap exception signal again, this could be due to a permanent fault. In this case, SymptomTM allocates another core, copies the checkpointed state of the availableTX to the second core and re-executes the availableTX. If the second core does not raise an exception, that means that the first core had a permanent fault and finally it should be disconnected from the system. Otherwise, either the error is caused by the software or SymptomTM can not recover from the error. The algorithm used to diagnose the faulty core is quite similar to the one utilized in TBFD [64]. Fatal traps can be either because of a software bug or because of an error triggered by a transient or permanent fault. SymptomTM can not recover from an error if it raises the exception after the transaction commits. Therefore, the transaction size is a critical parameter for the error coverage.

SymptomTM does not have a perceptible performance degradation in the error free execution. It only increases pressure on memory bandwidth since the write operations are convoyed on commit. However, SymptomTM has limited error coverage since it cannot detect silent data corruptions (SDC) and, further, exceptions can be raised after the commit of the transaction. Thus, we argue that SymptomTM is an attractive scheme to provide high availability to the systems such that web-search engines and it is not convenient to be used for high reliability

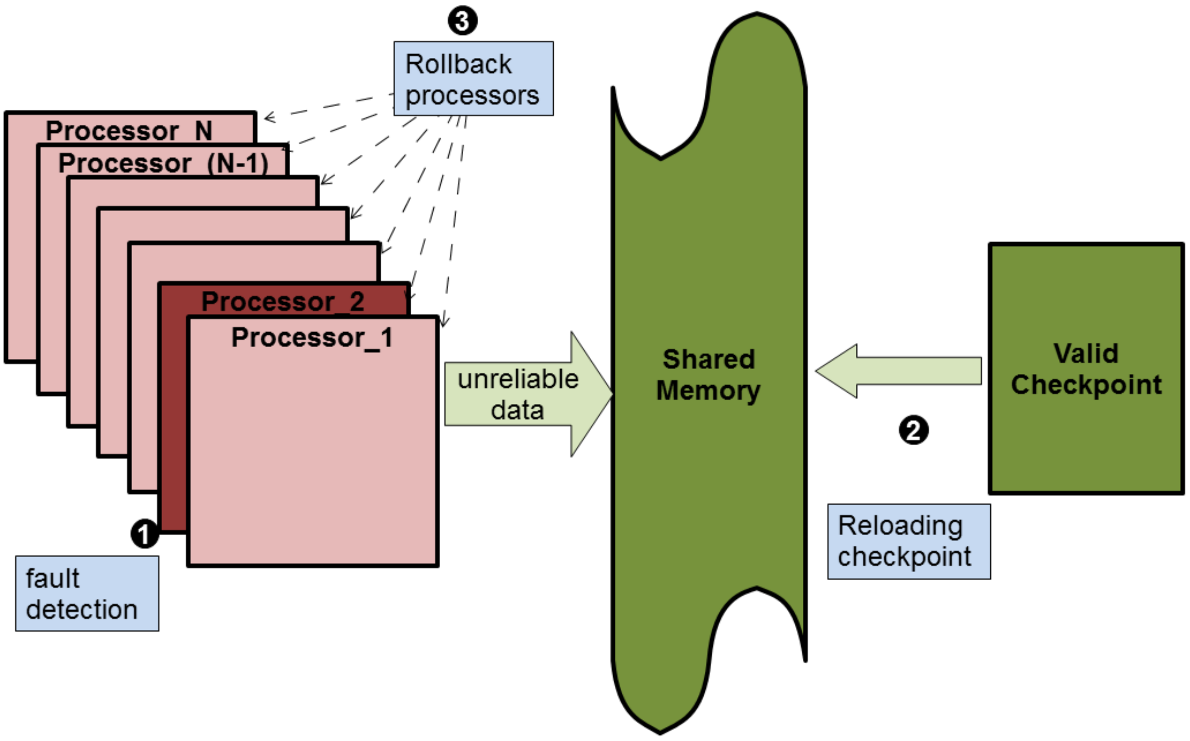


Figure 4.4: In global checkpointing, when an error is detected in one processor, all the communicating processors should rollback.

requiring systems such as the ones executing mission critical applications.

Some symptoms can be observed very efficiently (e.g., catching exceptions) and symptom-based error detection can be easily combined with other error detection mechanisms. Some other symptoms such as mispredictions in the high confidential branches can also be used as symptoms of errors. However, they may cause false positive impact (i.e. a misprediction which are not due to a fault) unlike fatal traps, thus, they are not convenient to be used for permanent fault detection. Similarly, those symptoms (e.g., infinite loops due to a corruption of the stop condition) may require an instrumentation of the source code or a support by the operating system (e.g, adding timeouts).

4.2.1 Lightweight Checkpointing with Transactional Memory

The main benefit of SymptomTM is that it provides an error recovery mechanism by leveraging the abort mechanism of TM. SymptomTM presents a lightweight checkpointing scheme with TM hardware. It presents less overhead compare to global checkpointing. In Figure 4.4, we demonstrate the global checkpointing scheme. In global checkpointing, all processors in the system agree on a valid checkpoint by passing several messages or synchronization primitives. When a fault is detected in one processor (1), this valid checkpoint is reloaded to the shared memory for recovery (2). Also all communicating processors in the system rollback even if they are error-free (3) which causes additional performance loss and power dissipation. For system-wide checkpointing and data sharing systems, once a fault is detected, no core in the system can be assumed to be fault-free. Due to this problem mSWAT [62] replays all cores in the system up to three times to diagnose the faulty core. SymptomTM replays one core once.

SymptomTM uses HTM with lazy data versioning in which shared memory keeps only validated data, therefore any error occurring in a core does not propagate to other cores through memory. Thus, only the erroneous core rolls back while the rest of the system keeps running without wasting any error-free work done. SymptomTM recovers from errors using the abort mechanism of TM which rolls back to the beginning of the availableTX in the erroneous core. We argue that this is a simpler method than system-wide checkpointing [43, 47] that requires complex synchronization mechanisms to guarantee that all structures rollback to the same state. Moreover, system-wide checkpointing requires long recovery time due to its long checkpoint interval. SymptomTM flushes the local log area and loads back the checkpointed register file for recovery only in the faulty core which has a negligible overhead. Moreover, SymptomTM does not require any additional hardware structures on top of transactional memory hardware to save checkpointed state. Also availableTX sizes are very short compared to system-wide checkpointing.

4.2.2 Architecture of SymptomTM

SymptomTM extends a popular lazy-lazy HTM design [1] with minor modifications. In this section, we explain these extensions.

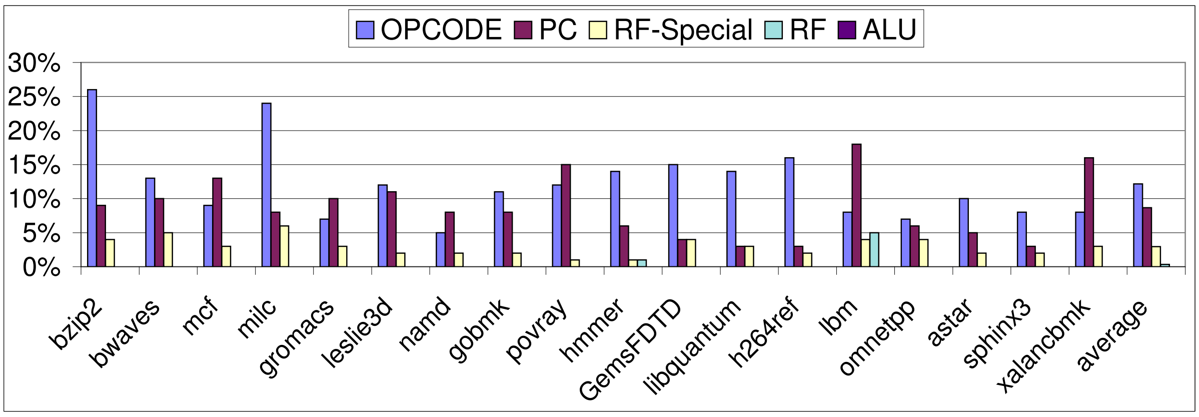


Figure 4.5: Fatal Trap Rate of Injected Transient Faults

Splitting the availableTXs

In HTMs with lazy data versioning, local log buffers can emerge as a bottleneck for large transactions. Therefore, it is infeasible to execute the entire application in one availableTX. Fortunately, we can split availableTXs since they are for error recovery not for parallelism and they can be validated when it is desired. SymptomTM can commit when the write-set size reaches the transactional log size. Therefore, in run-time, the whole application is split into finer grained back-to-back availableTXs with constant-size log buffers. When the whole write-set is committed, a new availableTXs starts by clearing the write-set. Splitting availableTXs allows us using a very small fully associative special transactional cache which holds the write-set and read-set of the whole availableTX.

Protection of Checkpoint

HTM systems checkpoint the register file at the beginning of a transaction. This checkpoint is utilized to be able to roll back to the beginning of the transaction when the transaction aborts. However, this buffer area is vulnerable to hardware errors, as well. Thus, we protect register checkpoints of availableTXs by ECC as in ReStore [40] in order to cover the worst case scenario in which both the instruction execution and the checkpoint of the register file are faulty due to multiple faults. Note that this ECC protection is to reduce the error rate from the checkpoint storage rather than to reduce the error rate from the instruction execution pipeline. Encoding and decoding of these ECC do not present an extra time overhead to the common error-free case. This is because ECC is generated only at the beginning of the availbaleTX, in parallel with the execution and it is decoded only in the erroneous case to safely rollback to the beginning of the availableTX. As opposed to SymptomTM, in several previous fault recovery schemes [30, 36], it is ensured that the architectural state is error-free by utilizing ECC on the architectural register file which presents encoding/decoding overhead every time the architectural register file is accessed. On the other hand, since shared memory is not in the coverage of SymptomTM, ECC values of the write-set entries are generated during the execution of store instructions.

Watchdog Mechanism

Assuming that an error in one availableTX leads to an incorrect execution path (i.e. an infinite loop), the availableTX may not reach the write-set limit or end of the application. SymptomTM employs a watchdog mechanism which records the time passed since the commit of the last availableTX in the processor. So that it enforces the availableTXs to abort and recover if the watchdog threshold has been exceeded since the last commit. As we stated in Figure. 4.3, the fault can be permanent and the second execution of availableTX after the recovery may exceed the watchdog threshold. In this case, the availableTX is enforced to abort again and it is re-executed for the third time in another core. If the third execution of availableTX finishes successfully that means that the first core had a permanent fault and finally it should be disconnected from the system. Otherwise, either the error is caused by the software or SymptomTM can not recover from the error. Here, we set the watchdog threshold long enough (i.e. couple of minutes) not to cause the termination of the application erroneously.

Handling Input/Output Operations

In reliable systems, only error-free data can be communicated outside of the sphere which is called output commit problem. For example, the system can not print unvalidated data to the user screen. Also, input commit presents problem in reliability since input messages should be replayed after recovery. In SymptomTM, we adopt the practical solution of both TM and checkpointing in which output values are deferred until validation (end of availableTXs in our case), and input values are logged to replay after recovery. Note that the size of the availableTXs are small enough for output delay.

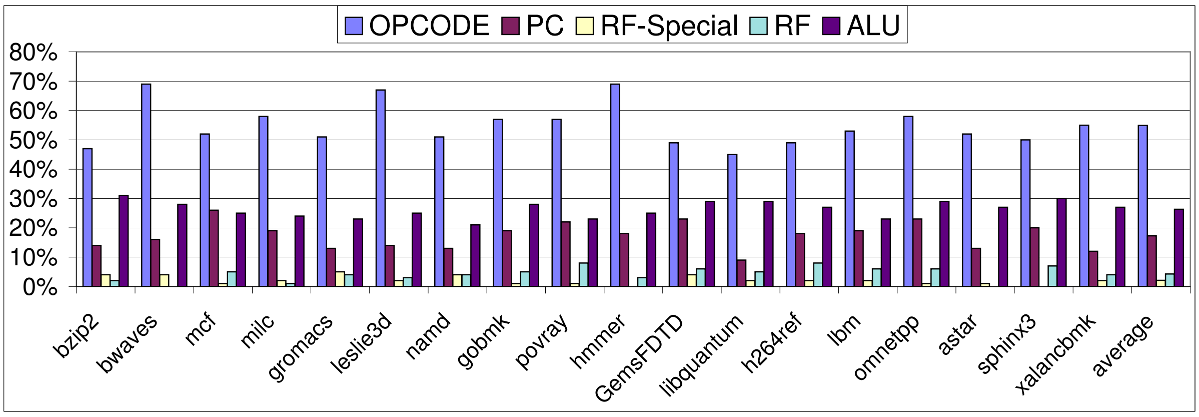


Figure 4.6: Fatal Trap Rate of Injected Permanent Faults

4.3 Evaluation

In this section, we evaluate the reliability performance of SymptomTM. We use the M5 full-system simulator [52] with an implementation of a Hardware Transactional Memory system that uses lazy data versioning and lazy conflict detection [69]. We extend this simulator with our SymptomTM implementations. We evaluate our schemes in in-order Alpha 21264 cores [53] having L1D and L1I caches and a unified L2 cache. Each L1 cache is 64KB with four-way set associativity, and a two-cycle hit latency. The L2 cache is 2MB with eight-way set associativity, and 10 cycles of hit latency. All caches are write-back with a line size of 64B and a local HTM write-set buffer with 32 entries. Main memory latency is 100 cycles. We evaluate SymptomTM by using spec cpu2006 [54] benchmark suite with test data-set by executing either 2 billion instructions or until application termination.

We use FimSim fault injection infrastructure that we explain in Chapter. 3 to measure the reliability performance of SymptomTM for both transient and permanent faults. We inject the faults to five different structures in a core; instruction opcodes, program counter (PC), integer register file (int-RF), special purposed register file (RF-special) and arithmetic logic unit (ALU). We did not inject faults to TLBs since our experimental results in Chapter. 3 shows that TLBs are not vulnerable to transient and intermittent faults. Also we did not inject faults to caches and main memory since these structures are out of the coverage of SymptomTM. We inject 100 faults per structure in each application to a random location in each structure at a random time after warming up 200M instructions by performing one injection per simulation.

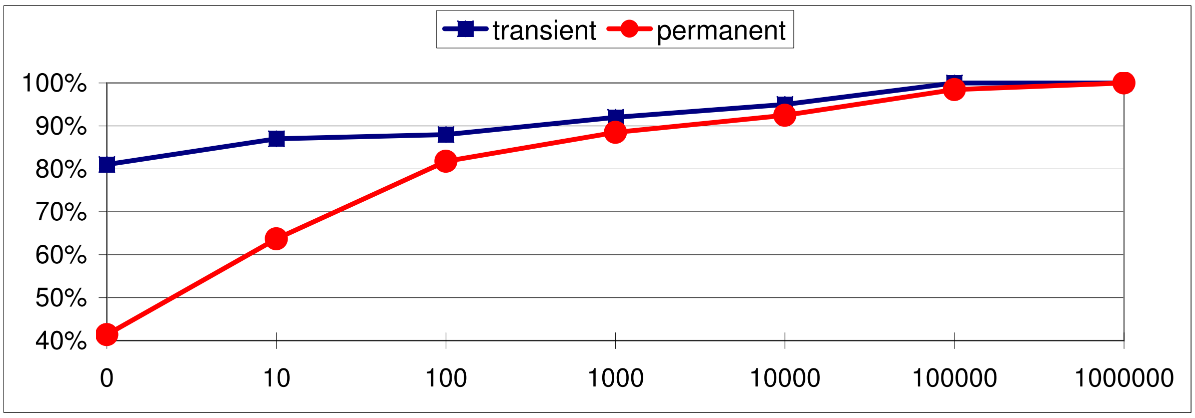


Figure 4.7: Number of Stores Executed between Fault Injection and Exception Raise

While we flip the chosen bit for transient fault injection, we use stuck-at-0 and stuck-at-1 models for permanent faults. We simulate 10M cycles after fault injection to observe the effect of the injected fault. Although we are aware of the fact that a fault may manifest itself as an error even after 10M-cycles, it is not feasible to simulate many cycles at every fault injection, thus we limited our tracking window to 10M-cycles.

Figure 4.5 and Figure 4.6 shows the hardware exception rate of transient and permanent faults according to our fault injection experiments. As expected, permanent faults are more likely to induce catastrophic failures than transient faults (21.6% vs. 4.8%). We present the exception ratio for different microarchitectural structures, because, in terms of system failures, each structure in the microarchitecture is not equally vulnerable. Note that our fault injection structures are different from SWAT [39] so that our exception ratio for permanent faults is different. Also, SWAT does not present these results for transient faults in detail.

SymptomTM can detect all these critical faults. However, it is able to recover from an error if it raises the exception before the end of the transaction. Therefore, transaction size is a critical issue for recovery. Also, the transaction size is limited by the size of the local buffer area that is correlated with the maximum number of store instructions within a transaction. Figure 4.7 depicts the number of store instructions executed between the bit flip and exception raise. 87% of catastrophic failures induced by transient faults raise an exception within 32 store instructions (Transaction size in SymptomTM). However, fewer amount of the permanent faults (70%) raise exceptions within 32 store instructions. SymptomTM with a write-set of 32

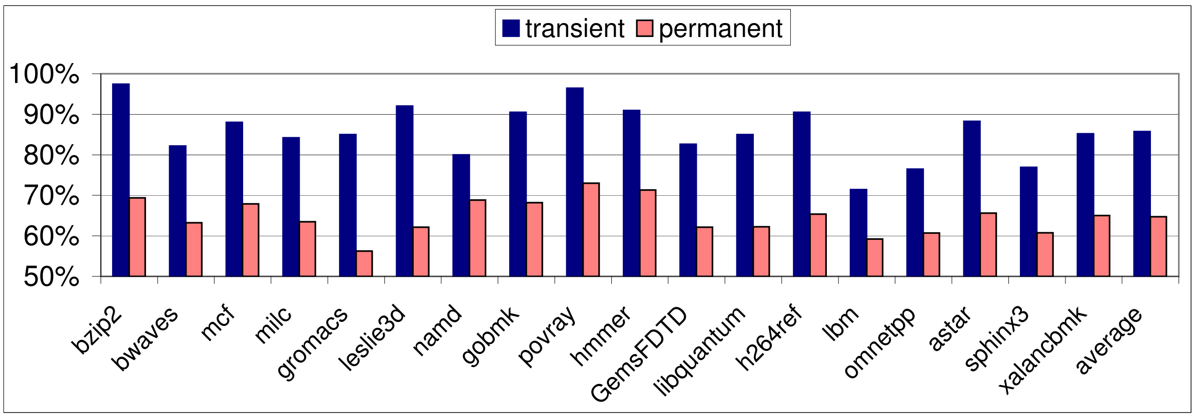


Figure 4.8: Error Recovery Performance of SymptomTM

entries recovers, on average, 86% and 65% of catastrophic failures caused by transient faults and permanent faults respectively (Figure 4.8). Note that, our evaluation is pessimistic for permanent faults assuming that the fault can be recovered only by going back to the first bit flip state which may be in an earlier transaction. This coverage can be increased with larger write-sets. As it is seen from Figure 4.7 within 1M stores (~10M instructions which is the checkpoint interval in SWAT) all catastrophic failures can be recovered.

4.4 Summary

In this chapter, we introduce symptomTM, an availability approach that leverages lazy-lazy hardware transactional memory (HTM) system for both transient and permanent faults. SymptomTM avoids catastrophic failures by monitoring fatal traps without any performance or core overhead in the general error-free case. Also, the hardware overhead of SymtomTM is negligible compared to lazy-lazy HTM system. SymptomTM leverages the lightweight checkpointing mechanism of Hardware Transactional Memory (HTM) for error recovery. Also, it avoids error propagation to the whole system by utilizing the isolation property of transactions.

5 FauTM: Redundancy-Based Error Detection with Transactional Memory

Reliability is a first-class design constraint for processor designers, especially in mission-critical domain (e.g. automotive cruise-control systems or financial applications). A reliable system should include two key capabilities: 1) error detection and 2) error recovery.

In the previous section, we present that the lightweight checkpointing mechanism of transactional memory can be utilized for error recovery. In this way, hardware structures implemented for optimistic concurrency in HTM systems can be leveraged in order to provide low-cost error recovery. However, symptom-based error detection as in SymptomTM, despite its very low performance degradation, is not convenient to be used for the systems requiring high reliability.

To satisfy the strict reliability requirement of mission-critical systems, various redundancy-based error detection solutions have been proposed [30, 31, 32, 33, 36, 37]. Executing instruction streams redundantly in chip multi processors (CMP) provides high reliability since it can detect both transient and permanent faults. Additionally, it also minimizes the Silent Data Corruption rate. In particular, lockstepping is a popular hardware based error detection scheme and is widely implemented in systems requiring high-reliability such as the IBM S/390 G5 [32] or the HP NonStop servers [33]. Lockstepping executes an instruction stream redundantly in two synchronized and lockstepped processors and checks if both produce identical results.

Error detection presents a performance overhead in reliable systems every time it is triggered. For instance, the comparison of execution results in order to detect divergent execution of redundancy-based error detection causes synchronization and comparison overheads in the execution time especially if the inter-processor communication channel has a limited bandwidth. Moreover, if error-detection is frequently triggered, the possibility that a benign fault causes an error recovery increases. In TM, transactional semantics allow the error detection to be deferred until a transaction commits (or the value becomes externally visible), so that the cost of error detection can be reduced while its efficiency can be increased.

In this chapter we present FaulTM, an error detection and recovery proposal based on Hardware Transactional Memory (HTM). We design FaulTM for mission-critical systems that require high reliability (e.g. financial or health applications) in which redundancy is essential to provide high error-detection rate. We have **two main goals** in designing FaulTM:

(1) to minimize the implementation complexity by utilizing existing hardware with minor changes,

(2) to reduce the performance degradation of redundancy-based error detection.

FaulTM utilizes an HTM which detects conflicts at the end of transactions (lazy conflict detection) and commits the validated version of the data at the end of the transactions (lazy data versioning). The main advantage of using lazy-lazy HTM is that comparison of write-sets for error detection reduces the comparison overhead of redundancy-based fault detection due to multiple writes to the same address. In this chapter, we explain how a generic HTM could be minimally modified so that it can also support reliability in addition to HTM's intended purpose of supporting optimistic concurrency. FaulTM executes applications in two redundant threads (i.e it creates a backup thread) and in special-purpose reliable transactions (From now on, we will call these reliable transactions as *reliTX* in order to avoid any confusion with regular TM transactions). FaulTM classifies any mismatch between the write-sets and register files of reliTX pairs (i.e. original reliTX and backup reliTX) as a hardware error (transient or permanent), and aborts both reliTXs, which are then restarted. In the case of a complete match, original reliTX commits the changes to the shared memory. FaulTM utilizes an HTM which detects conflicts at the end of transactions (lazy conflict detection) and commits the validated version of the data at the end of the transactions (lazy data versioning). HTM systems already have well-defined comparison mechanisms of read-/write-sets in order to detect if there is any conflict between transactions. While comparison of addresses is sufficient for conflict detection in TM, some systems also send data along with addresses. FaulTM adapts these already existing conflict detection mechanisms for error detection.

FaulTM is a self-contained reliability proposal in which error detection and recovery is integrated without requiring any external mechanism. It provides reliability for sequential and parallel (non-TM and TM) applications against transient and permanent faults. Also, compared to lockstepping [32], it reduces the execution time overhead. It avoids error propagation out of the processor. It leverages the checkpointing mechanism of TM that provides local checkpointing for fault tolerance. It eliminates the requirement of separate input replication mechanism.

In this chapter, we first explain our motivation to use transactional semantics for error detection in Section 5.1. Then, we present the design of FaulTM for sequential applications in Section 5.2. We extend the FaulTM design for parallel applications by addressing the challenges of traditional lock-based and TM applications in Section 5.3. After evaluating FaulTM in Section 5.5, we present other reliability schemes utilizing transactional memory which are published after FaulTM and SymptomTM are published in Section 5.6. We also present the pros and cons of the different design parameters of transactional memory from the point of reliability in Section 5.6.1.

5.1 Motivation

In Figure 5.1(a) and Figure 5.1(b), we show a simple loop written in C and in pseudo-assembly-language respectively. In several fault detection schemes [31, 32, 33, 70] only the results of the store instructions are compared assuming that an error is benign unless it propagates out of the core. With these schemes, there should be 2*N comparisons/synchronizations in the example (Figure 5.1(c)). Alternatively, with FaulTM, the processor splits the code into series of reliTXs, and performs validation of replicas at the granularity of complete reliTX, rather than at the granularity of individual stores. If reliTX correspond to loop iterations as in Figure 5.1(d), then it reduces the comparison and synchronization overhead significantly (from 2*N to 2). Also, the register file can be compared at this synchronization state to guarantee that it is error-free since the last validation.

```
for(index=0;index<N;
            index++)
{
    counter++;
}
```

(a) Simple Loop in C

```
     LOAD    index
     MOV     index,0
     LOAD    N
     JMP     EL
     STORE   index
BL:  LOAD    counter
     ADD     counter, 1
     STORE   counter
     ADD     index,1
     STORE   index
     LOAD    index
EL:  CMP     index,N
     JL      BL
```

(b) Pseudo Assembly Code of the Loop

```
     LOAD    index
     MOV     index,0
     LOAD    N
     JMP     EL
     STORE   index
—validate index—
BL:  LOAD    counter
     ADD     counter, 1
     STORE   counter
—validate counter—
     ADD     index,1
     STORE   index
—validate index—
     LOAD    index
EL:  CMP     index,N
     JL      BL
```

(c) Validation After Every Store Instruction

```
begin ReliTX
     LOAD    index
     MOV     index,0
     LOAD    N
     JMP     EL
     STORE   index
BL:  LOAD    counter
     ADD     counter, 1
     STORE   counter
     ADD     index,1
     STORE   index
     LOAD    index
EL:  CMP     index,N
     JL      BL
end ReliTX
—validate index—
—validate counter—
```

(d) Validation At the End of the reliTX

Figure 5.1: Example of Error Detection in Fine Granularity vs in Coarse Granularity

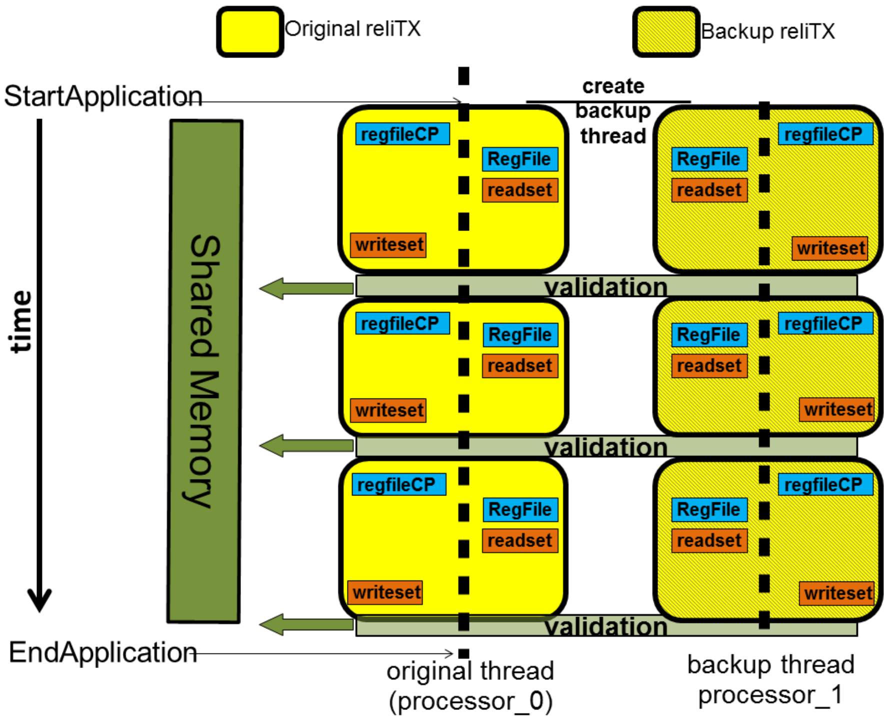


Figure 5.2: Design of FaulTM for Sequential Codes

5.2 Basic Design of FaulTM

In Figure.5.2, we demonstrate the basic design of FaulTM that we explain in this section. FaulTM provides high reliability based on 4 steps:

Creating reliTXs:

At the beginning of the execution, FaulTM hardware creates a backup thread which executes the identical instruction stream to the original thread. Then the original and backup threads are executed as two separate reliTXs. In lazy-lazy HTM, creating a transaction means starting to write the values to the local buffer area instead of shared memory, thus it does not present a transaction creation overhead.

Executing reliTXs:

Both original and backup reliTXs are executed atomically and in isolation. Each reliTX independently sends load requests to shared memory or read-sets. In the FaulTM approach, there are no conflicts between the original and the backup reliTXs, because the backup reliTX is only for validation of error-free execution and it does not modify shared memory. Note that,

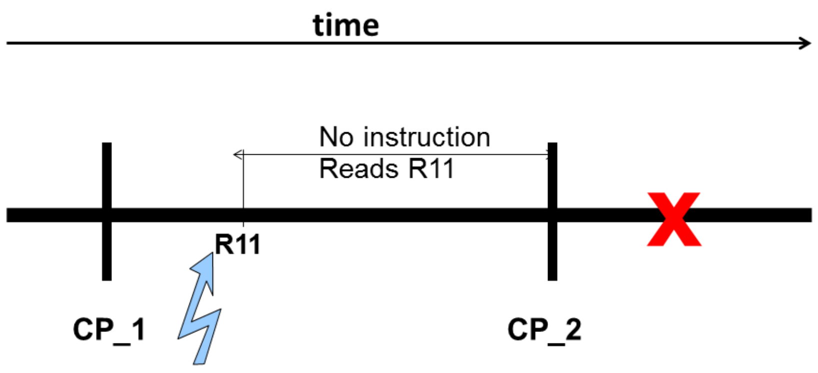


Figure 5.3: Full-State Comparison Ensures the Error-Free Operation Since the Last Validation

an original reliTX may conflict with normal transactions or other reliTXs except its pair reliTX. When a reliTX aborts to resolve a conflict, its pair reliTX also aborts and restarts.

FaulTM executes original and backup threads in different cores in order to detect both transient and permanent errors. This cause 100% core and energy overhead which is paid by all previous redundancy-based reliability schemes [30, 31, 36, 37] on CMPs for high reliability. While reading from memory is done redundantly by each core, writing to main memory is achieved only by one core after validation. Thus, FaulTM can detect the errors on the bus during the read operation from the memory. However, it can not detect errors during the write operation to the memory.

Ending reliTXs:

In FaulTM, we start commit processes whenever the write-set size reaches the transactional log size. Since both threads have identical instructions in the absence of a fault, their read-/write-sets have to be identical as well. Note that, we left the randomness and non-determinism problems out of the coverage of FaulTM as previous redundancy-based reliability methods do [30, 31, 37].

Original and backup reliTXs wait for each other (spin) to reach the commit stage. Then, the reliTX pair compare their write-sets and register files through the comparators in the backup reliTX. (From now on we call this operation as *validation*). Note that, the comparison of read-set is not necessary since it can be recovered by re-loading from shared memory. If they match, the original reliTX commits its changes to memory, and the backup reliTX is cleared as if it aborts and it does not re-execute. Mismatch means an error due to a hardware fault in one of the reliTX and it starts the recovery.

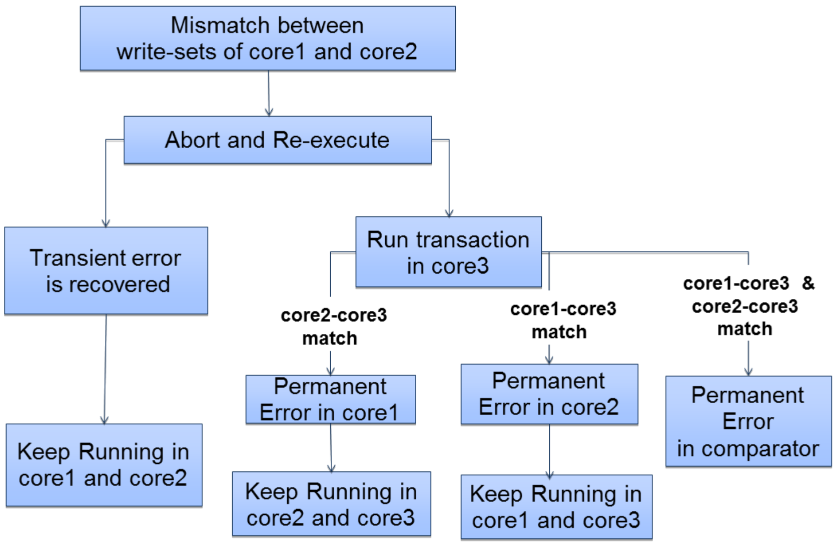

Mismatch between write-sets of core1 and core2
Abort and Re-execute
Transient error is recovered
Run transaction in core3
core2-core3 match
core1-core3 match
core1-core3 & core2-core3 match
Permanent Error in core1
Permanent Error in core2
Permanent Error in comparator
Keep Running in core1 and core2
Keep Running in core2 and core3
Keep Running in core1 and core3

Figure 5.4: Error Detection Algorithm of FaulTM

Redundancy based fault tolerance methods detect errors by comparing either only the results of store instructions [27, 31] or results of all instructions [29, 30, 37] due to the assumption that a fault is benign unless it propagates to the architectural state. However, only full-state comparison (e.g. checking the register file) guarantee the error-free operation since the last validation. We demonstrate an example that presents the importance of full-state comparison in Figure 5.3. In the example, a register (R11) becomes faulty after the validation of a checkpoint (CP_1). However, until the validation of the next checkpoint (CP_2), the fault does not affect the output of any instruction nor an instruction writes over R11. When this unmasked fault causes a detected error after the validation of CP_2, the system rolls back to the CP_2 which, unfortunately, is not fault-free and cannot recover from the error. FaulTM compares register file (including special purposed registers such as PC) at the end of reliTX (instead of after every instruction) with acceptable performance degradation. Note that, whole inner-state comparison can be supported as well.

Error Recovery:

If either write-sets or register files do not match at validation, both the original and backup reliTXs abort and they restart execution (Figure 5.4). If they match in the second execution, that means that there was a transient fault either in one of the cores or in the comparators in the first execution.

Two successive mismatch signals between the same original and backup reliTXs signify that either one of the cores or the comparators has a permanent fault. FaulTM executes the reliTX in a third core to detect the source of the permanent fault by comparing the results with this third core's comparators. If the write-set of the third reliTX matches with either the original or the backup reliTX, the mismatched core is marked as the faulty core. If both the original and backup reliTXs match the third reliTX, the comparators are marked as faulty. The algorithm that is utilized to diagnose the faulty core does not have a performance impact in the common error free case as it is explained in mSWAT [62] and TBFD [64]. Note that, FaulTM can not detect more than one faulty component (e.g. both cores are faulty) at a time.

5.2.1 Architecture of FaulTM

FaulTM extends a popular lazy-lazy HTM design [1] with minor modifications. First of all, we apply the extensions required for SymptomTM to FaulTM as well. To this end, to handle the I/O operations, we deferred output values until the end of reliTXs and we logged input values to reply after recovery. Similarly, we protect the checkpoints of the register file via ECC. Also, we split the reliTXs into finer granularity and start the validation process when the write-set size reaches the transactional log size. Finally, we also elaborate the watchdog mechanism to avoid the infinite loops.

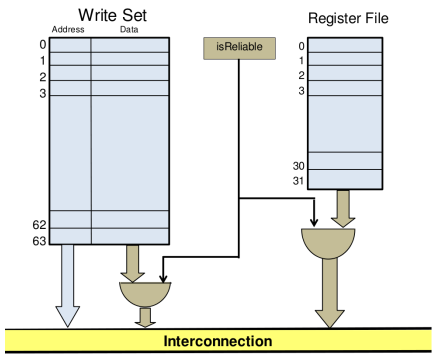
In this section, we explain the additional extensions required for FaulTM.

Microarchitecture Extensions for FaulTM implementation

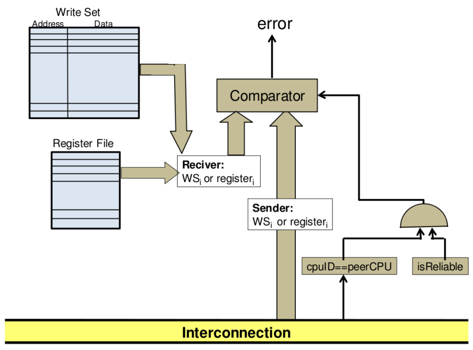
We need a simple controller having the list of processors in order to manage reliTXs. Besides coreIDs, we add several bits per core to this list such as *isReliable, isOriginal, peerCPU, wasFaulty and controlCore* bits, to account for mechanisms to deal with reliable execution.

`isReliable/isOriginal`: We may have additional TM applications running concurrently with our reliability-critical applications. Therefore, we use the isReliable bit to distinguish between the transactions and reliTXs. This bit is also sent together with the write-set at validation. If isOriginal bit is set, it indicates that the reliTX is the original one otherwise it is the backup reliTX.

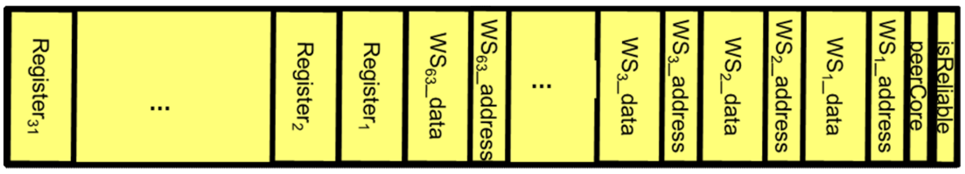
`peerCPU`(($\lceil \log_2 n \rceil$)-bits for an n-core system): reliTX pairs have to be aware of each other to compare their results. The `peerCPU` bits point to the processor that has the peer reliTX of the reliTX that runs in the current processor.



(a) Processor Executing Original reliTX



(b) Processor Executing Backup reliTX



(c) Data Sent to the Interconnect During Comparison

Figure 5.5: The Validation Mechanism of FaulTM (Modified Structures are Shown in Grey)

`wasFaulty`: This bit records if the core had a fault at the last time it executed a reliTX, hence it distinguishes if there might be a permanent fault. The `wasFaulty` bit is reset every time the core executes an error-free reliTX.

`controlCore`: When FaulTM detects two consecutive mismatches, it allocates a new core to recover from a permanent fault. The `controlCore` bit is set in the allocated core which is reset when an error-free reliTX is committed.

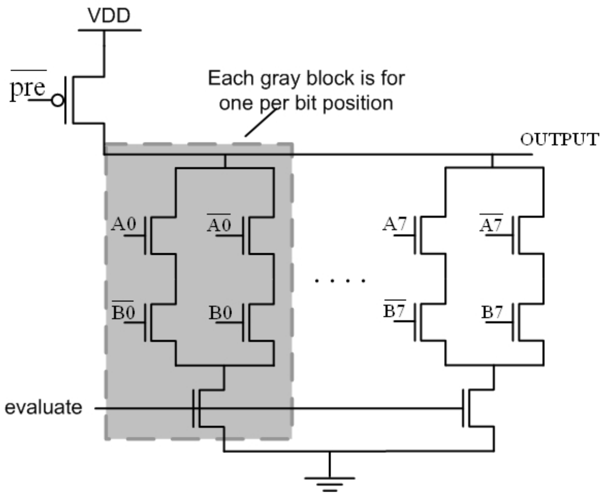
Adapting Conflict Detection for Error Detection

In Figure 5.5, we demonstrate the paired processors running original and backup reliTX as well as the interconnect during error detection. Lazy-lazy TM systems compare address values of stores in both read-sets and write-sets for conflict detection. FaulTM compares address values as well as data values of stores in write-sets to detect if an error has manifested itself either in the address or in the data. This comparison is essential for reliability and FaulTM reduces the overhead of this comparison compared to prior reliability techniques [30, 31, 32]. Furthermore, only reliTXs that set the `isReliable` bit send store data to their peer reliTX. Reliability purposed comparison can be performed as in-order buffer comparison without requiring the associative address search of HTM since, write-sets are identical in the absence of faults. Also, normal parallelism purposed transactions are not affected by this overhead.

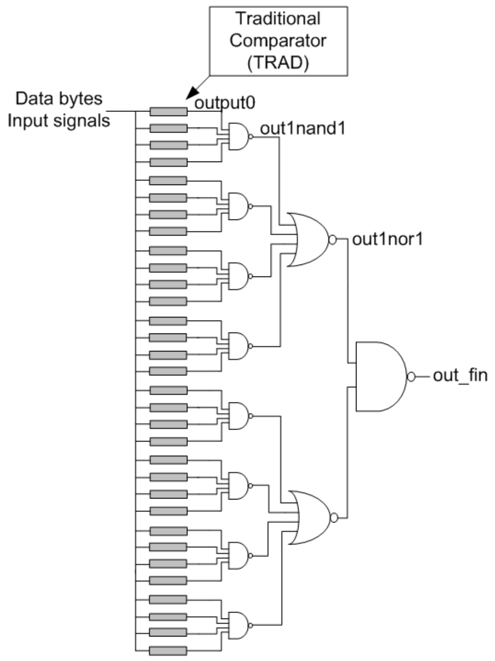
In order to improve the performance and energy consumption, we propose two extensions for the comparison operation: 1) Designing comparators for error detection 2) Generating signatures of the comparison data.

Comparators for Reliability Comparison

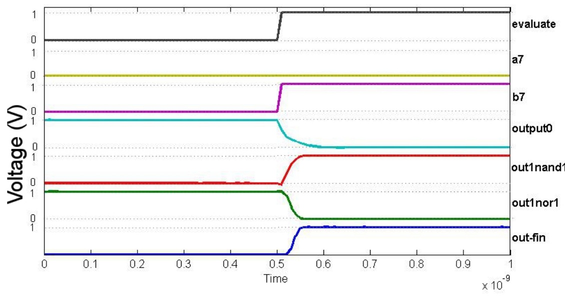
In most transactional memory implementations, only address values of both read and write sets are compared and match means conflict and reason to abort. In FaulTM, although data comparisons present an overhead over a usual HTM, this overhead is paid by most of the redundancy based fault tolerance techniques [30, 31, 32, 33]. Note that, writesets of reliability purposed transactions are in order without necessitating CAM logic. In FaulTM, first, the write set addresses are compared, a match signifies that there is no error in the store address. Once a match in the addresses is found, then the data are compared; in case of a match there is no error in the store data. Since the error case is rare, most of the time data comparison will register a match in FaulTM. Therefore, we use dissipate-on-mismatch comparators [71, 72] for data comparison. However, when the address comparisons is conducted in normal HTM, it mostly registers a mismatch since a store address is compared against the whole write set. Here special comparators that dissipate little energy on mismatches or partial matches as proposed by Ponomarev et al. [72] could be used to save energy. Figure 5.6-a shows an 8-bit dissipate-on-mismatch comparator with a pull-down design that compares two 8-bit comparands with each other. The output is precharged high in the precharge phase. It remains high unless any bit mismatch occurs in the evaluation phase. If there are any mismatches in bit positions, the output will discharge via one of two parallel branches of NMOS transistors in the related gray block. Thus the significant amount of energy is dissipated on a mismatch. As in our case, mismatches occur less frequently than matches, more energy saving can be realized by using these kind of comparators. Figure 5.6-b depicts a new scheme for comparing longer arguments of 32 bytes. An obvious way might be to extend the traditional dissipate-on-mismatch comparator bits to 32 bytes. However, this method leads to significant increase in delay time and power consumption. Therefore we use the evaluation tree design that is the better approach [72]. Normally final output is in low level. If there is even one mismatch bit position in 32 bytes, the final output will be charged to high level. We design and simulate this 32 bytes comparator with Hspice 2003.03 using HP 45nm Predictive Technology Model [73] for VDD=1V, 1GHz



a) Dissipate on mismatch comparator

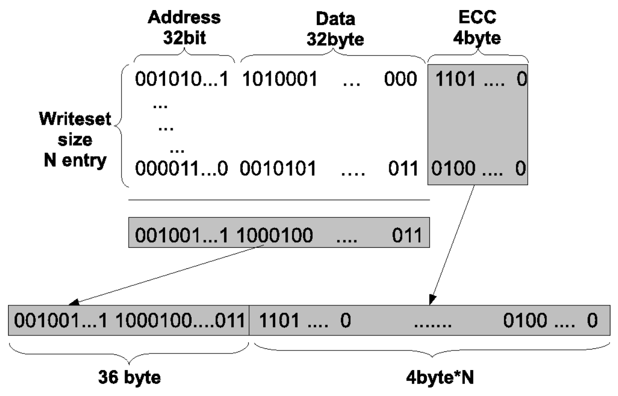


b) Comparator for 32 bytes

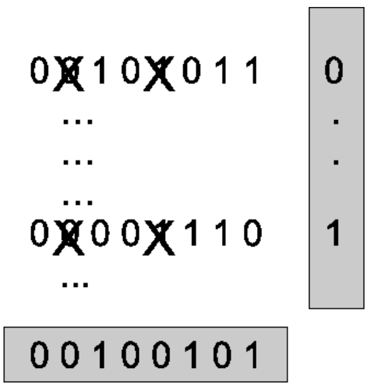


c) Timing Diagram of the Comparator

Figure 5.6: Comparator Design for Data Comparison



(a) Composition of the signature of the writesets including addresses and data



(b) A Case that signature can not detect the error

Figure 5.7: Signature of Writeset

frequency, and T= 25_C. Figure 5.6-c shows the simulated timing diagram of the internal nodes when there is only one bit mismatch in the first TRAD, bits a7 and b7, which is the worst case scenario for delay time. At the evaluation time, when signal evaluate goes high, a7 remains low but b7 goes high. In fact, the worst case for discharging the output of TRAD is only one bit mismatch occurring. The output of the first TRAD, output0, goes low; the output of the first NAND, out1nand1, goes high; Our timing analysis confirms that this operation can be performed in one cycle at our target frequency of 1GHz.

FaulTM-sig: Comparing Signatures of Write-Sets

We can also reduce the cost of error detection with signature comparison. There are Transactional Memory systems using signatures (i.e. bloom filter) to compress a transaction's read- and write-sets to reduce the cost of conflict detection. False positives (detecting conflict although there isn't) are possible in these signatures but they never give a false negative signal (missing the detection of conflict). Similarly, FaulTM reduces the comparison cost significantly by utilizing an XOR-based signature as shown in Figure 5.7(a). This signature is known as overlapping parity [74] in which each bit is covered by more than one parity. The ECC values are calculated during the composition of writesets, because the bus between processors and shared memory is not protected by FaulTM. The parity bits of the columns are calculated at the end of the transactions.

We call this mechanism as FaulTM-sig in order to separate it from the base design. This mechanism is similar to Fingerprinting [37] in which a cumulative signature of the execution trace is generated after every instruction instead of after every reliTX.

Fatal Trap Exceptions

Hardware faults may lead a fatal trap exception. For instance, a fault may change the opcode of an instruction to an undefined opcode which is a defined hardware exception. If one reliTX raises a hardware exception while its pair reliTX comes to commit, the recovery process starts and both reliTXs abort and restart execution. In Chapter 4, we explain that SymptomTM provides a similar mechanism using a single reliTX to recover from catastrophic failures. In SymptomTM, a reliTX is aborted and re-executed on encountering a fatal trap. Although this enables to recover from fatal crashes, the scheme has very limited error coverage since it cannot detect silent data corruptions and, also, exceptions can be raised after the commit of reliTX. SymptomTM can not distinguish if the fatal trap is caused by a hardware fault or a software bug. In FaulTM, software induced fatal traps cause both reliTX pairs to raise an exception while hardware fault makes only the faulty reliTX raise the exception.

Possible Programming Model

We argue that availability or reliability requirements of systems depend on the application. Some applications require a highly-reliable environment that can detect any error including silent data corruptions. For instance, a financial application should prevent transferring an incorrect amount of money to a bank account. These systems prefer seeing Segmentation Fault messages on the screen and rebooting their system rather than making an incorrect money transfer. We argue that FaulTM should be utilized for those systems that require high reliability. Some other applications demand high availability. For example, a web search engine is expected to be serving 7/24 while an error in the search result is not vital. We believe that SymptomTM can be utilized for high availability demanding applications. Another group of applications, i.e. airplane applications, require both reliability and availability since they can not boot the application to solve an error and also the correct result of the application is vital. For these applications, we utilize SymptomTM and FaulTM together.

FaulTM and SymptomTM can be extended with the programming model in order to provide reliability/availability in the finer granularity instead of duplicating the entire application. For this purpose, we add the keyword "`reliable`"to denote sections of code that should be protected by FaulTM against hardware errors including silent data corruptions. Similarly, in SymptomTM approach, programmer defines the vulnerable section by using the keyword "`available`"which means only catastrophic failures will be detected and recovered in the section to provide high availability. Using these keywords, programmers only need to define the vulnerable sections in their applications. They can insert vulnerability boundaries as if they define atomic sections in TM applications. The vulnerable sections can be either fine-grained, lasting for a few instructions or coarse-grained such as the entire application. While the fine-grained approach causes less performance degradation, coarse-grained approach provides more reliability/availability. For instance, for an airplane control application, the programmer could identify that the code that is responsible for controlling the flaps should be protected coarsely, whereas the code regarding the on-flight entertainment system is not protected at all. Alternatively in the fine grained version of flap controlling code, the programmer decides to protect only the calculation of desired flap angle but he leaves the graphic user interface unprotected.

5.3 Extending FaulTM for Multi-threaded applications

In this section, we present how FaulTM can be extended for multi-threaded applications running on transactional memory hardware. We provide the details of FualTM for both traditional lock-based parallel applications and TM applications.

5.3.1 Non-TM Parallel Applications

There are two main challenges in redundantly executing multi-threaded applications: (1) handling instructions dedicated to maintaining synchronization such as locks, barriers or create/join threads and (2) maintaining identical instruction streams of redundant threads.

Synchronization Instructions

In FaulTM, until reliTX commits its write-set to the shared memory, the rest of the system can not be aware of the instructions in reliTX. However, the thread management (e.g. create/join thread), coherency (allocate/release lock) and synchronization (e.g. barriers) instructions should be committed to the system to avoid deadlocks. Also, before the execution of these instructions, all the older instructions on the thread should be validated from the reliability point of view and committed. In this way, it is confirmed that the synchronization instruction is not

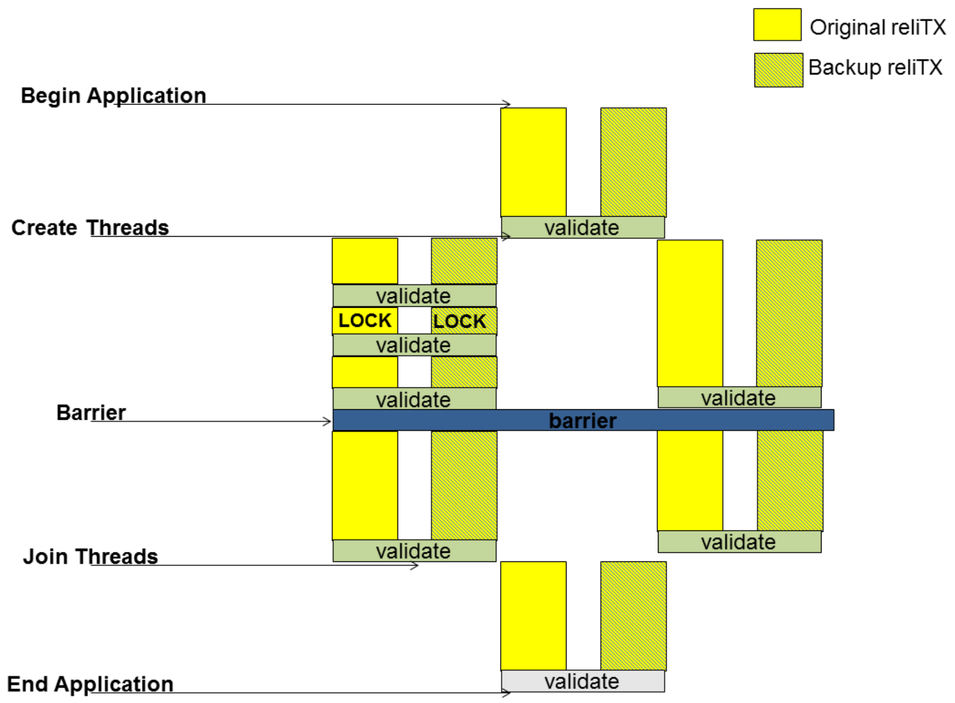


Figure 5.8: FaulTM for traditional, lock-based parallel applications

in an erroneous path in the execution. When FaulTM (Figure 5.8) encounters one of these special instructions (e.g. test-and-set, compare-and-swap) in a reliTX, before executing the instruction, it first validates and commits the current reliTX pair and starts a new reliTX pair. As soon as the special synchronization instruction is executed, FaulTM starts the commit process in order to make the system aware of the instruction. FaulTM executes a special instruction in a reliTX pair to keep it fault tolerant. Also, in this way, even if one of the reliTXs aborts due to a detected error after acquiring the lock, it will not try to reacquire the lock that it already held, which could have led to a deadlock. Note that lock release operations are accomplished by simple store instructions which can not be detected by the hardware so that it does not force reliTX to commit. However, delaying lock release instructions until the end of reliTX does not present any forward progress issues. Lock allocation/release is accomplished by only the original reliTX since the backup reliTX does not write anything to the shared memory. However, backup reliTX is allowed to operate on the data locked by its original pair.

In Figure 5.8 we demonstrate how FaulTM handles special instructions dedicated to maintain synchronization. In the figure, the application starts with a single thread and FaulTM

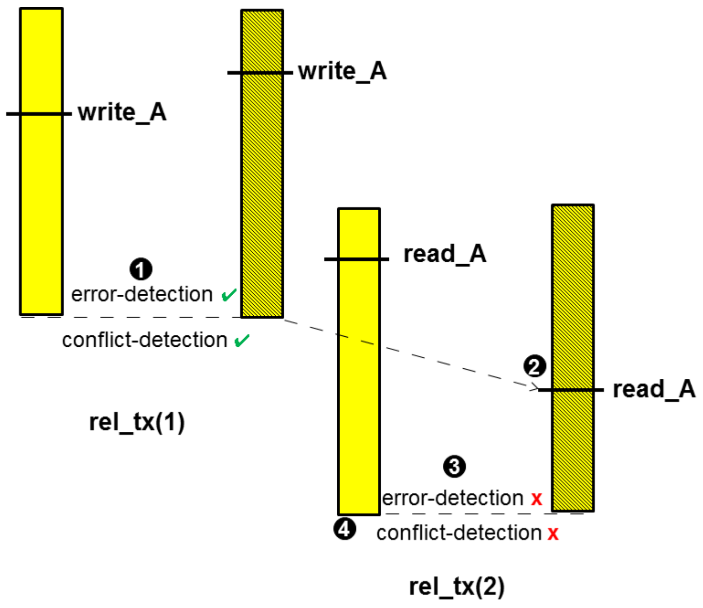


Figure 5.9: Addressing input replication in FaulTM

creates a backup thread in order to make it fault tolerant. Also these two threads are executed in reliTX pairs. When the application attempts to create a thread to execute some parallel operations, FaulTM forces the reliTX pair in the system to start validate/commit process. After the thread creation, there are 4 threads in the system (i.e. 2 originals, 2 backups) running in reliTXs. Later on, the first reliTX pair attempts to acquire a lock. Before executing the lock acquire instruction, the reliTX pair validates and the original reliTX commits its changes to the shared memory. After that, they execute the lock acquire operation reliably and only the original reliTX issues the instruction to shared memory. After the lock is acquired, original and backup reliTX pair operates on the data and writes the changes to their write-sets. After some point, when the write-set is full according to the FaulTM design, the reliTX pair compares their write-sets for validation and original reliTX commits its write-set to the shared memory. Since backup reliTX does not commit any data to shared memory, it does not present any correctness issue that it operates on a locked data by its original reliTX. Similar to the thread creation and lock acquiring, reliTX pairs are also forced to validate/commit at barriers and thread join instructions.

Input Incoherence

The second challenge for multi-threaded applications is maintaining identical instruction streams to replicated threads called input incoherence problem. Input incoherence occurs in redundancy-based reliability schemes when a value is changed by another thread in the system between the time it is read by the original thread and it is re-read by the backup thread. In Figure 5.9, we demonstrate an input incoherence in FaulTM and how the conflict detection mechanism of TM solves it. In the example, there are two original threads and two backup threads in the system which are executed in reliTX pairs namely reliTX(1) and reliTX(2). The original thread in reliTX(2) reads the value **A** before the reliTX(1) pair commits the new value of **A** to the shared memory. After reliTX(1) commits the new value of **A**, the backup thread in reliTX(2) reads **A**. Thus, in reliTX(2) the original thread reads the old version of **A** while the backup thread reads the new version written by reliTX(1). reliTX(2) detects an error since the original and the backup reliTXs operate on different values. Note that this case occurs in an application with a data race. However, the fault tolerance scheme treat this kind of race as an error and it calls the recovery scheme. If the race occurs repetitively in a fault tolerant architecture, the system treats the race as a permanent error and disables the hardware structure although it is not faulty.

In the example, although error-detection is adequate to abort and restart reliTX potentially solving the input incoherency, conflicts should also be detected after error detection to distinguish input incoherence, transient faults and permanent faults. For instance, two consecutive input incoherences on the same processor is considered as a permanent fault if conflict detection is not done. Note that both the original and the backup reliTXs should accomplish conflict detection since the late reader can be either the original or the backup reliTX.

5.3.2 TM Applications

Many researchers have developed applications using TM with the purpose of benchmarking different implementations, and studying whether or not TM is easy to use [75, 76]. We believe that, since HTM systems are implemented and by the time the TM programming model is adopted by programmers, there will be many TM applications to be executed on HTMs. Providing reliability to TM applications on HTM by using the conventional fault tolerance schemes is infeasible since they require additional comparison and checkpointing over TM itself. In this study, we provide error detection and recovery for TM applications in lazy-lazy HTM by leveraging the existing structures on the hardware (Figure 5.10).

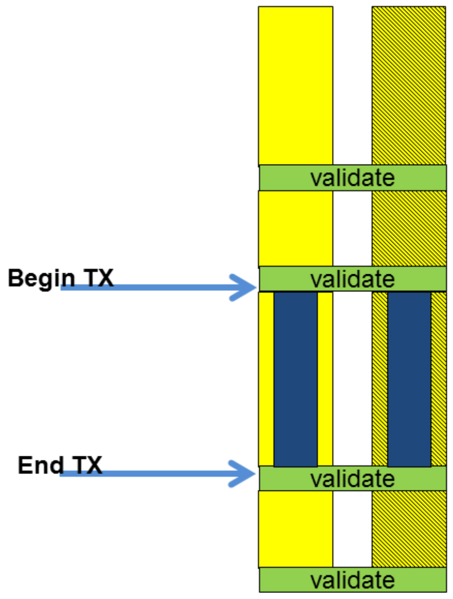


Figure 5.10: Design for TM Applications

When a thread encounters the instruction that starts a transaction, FaulTM first validates/-commits the reliTX before it starts the transaction. Then transaction starts both in the original and the backup processors and they are executed in reliTX. Since transactions already write the produced values to their write-sets, reliTXs do not need to register the write values again. ReliTXs do not start the validation until the transactions commit. If the TM transaction commits, the reliability validation is carried out before the TM transaction publishes its write-set to avoid any error propagation out of the core. In case the TM transaction aborts due to a conflict, reliTX is also aborted to avoid any correctness issues. After both transactions reach the commit stage, their write-sets and register files are compared before collision detection in order to abort erroneous transactions before they cause other transactions abort erroneously.

There are two key implementation questions for FaulTM for TM applications: 1)irrevocable operations in transactions and 2) nested transactions.

In TM, if there is an irrevocable operation in a transaction, TM marks this transaction as such and the transaction does not abort. There can be only one irrevocable transaction in the system. If a transaction conflicts with an irrevocable transaction, the conflicting revocable transaction aborts and the irrevocable one commits. When an irrevocable transaction is executed with reliTX, reliTX is validated before the irrevocable operation. reliTX creates a checkpoint of the register file and the write-set in order to ensure that if an error is detected after the irrevocable operation in the irrevocable transaction, it can roll-back after the irrevocable operation.

In TM systems, there can be nested transactions that begin and end within the scope of surrounding transactions. There are two types of nested transactions: closed and open. In a closed-nested TM system using flattening, either all or none of the transactions in a nested region commit. In contrast, in an open nested TM, when an inner transaction commits, its effects become visible for all threads in the system. In FaulTM, validations of the close-nested transactions are performed at the commit of outer most transaction while the validations of an open-nested transactions are performed when the nested transaction commits.

5.4 Benefits and Overheads of Using TM for Reliability

FaultTM also presents a lightweight checkpointing scheme by utilizing the abort scheme of TM as similar to SymptomTM. In this section, we explain other benefits and overheads of using Transactional Memory for fault tolerance.

5.4.1 Less Comparison Overhead

FaulTM reduces the comparison overhead compared to the previous redundancy-based fault-detection schemes [30, 31] due to two reasons. First, it compares the write-sets (instead of each store values) which have a fewer amount of entries than the total number of store instructions due to multiple stores to the same address (e.g incrementing a counter in a loop causes several store instructions but only one entry in the write-set). Also, it spins only at the end of reliTXs instead of after each store instruction. Second, register file comparison is done only at the commit stage of reliTXs (instead of after each instruction). Furthermore, comparison only at the commit point reduces the probability of detecting benign faults; because if a fault is masked within the reliTX, its effect is eliminated before the end of the reliTX. The comparison overhead of FaulTM can be reduced further by comparing hash-based signatures of write-sets and register files of reliTX pairs as in FaulTM-sig.

5.4.2 Full-state Comparison

Redundancy based reliability methods detect errors by comparing either only the results of store instructions [27, 31] or results of all instructions [29, 30, 37] by assuming that a fault is benign unless it propagates the architectural state. However, only full-state comparison (e.g. checking the register file) guarantees the error-free operation since the last validation. FaulTM provides full-state comparison at commit point with acceptable performance degradation.

5.4.3 Eliminating the requirement of separate input replication mechanisms

Previous redundancy based methods [30, 31] require input replication mechanisms such as a load value queue, because a value can be changed by another thread in the system between the time it is read by the first thread and it is read by the second thread. In FaulTM, the conflict detection mechanism of TM guarantees that there would not be any modifications in the loaded values by other threads. In this sense, FaulTM is similar to ReUnion [36] which does not require an extra hardware component for input replication. ReUnion recovers from the input incoherence by benefiting from the fact that if redundant threads read different values, they produce different results, otherwise the difference is benign. Therefore, it solves the input incoherence by utilizing an error detection mechanism. However, ReUnion is not convenient for permanent fault detection since it can not differentiate if two successive mismatch of results are due to a permanent fault or an input incoherence. FaulTM, on the other hand, solves the input incoherence issue between redundant reliTXs by benefiting from the lazy conflict detection of HTM which identifies concurrent accesses by tracking the addresses in read-sets and write-sets. Note that loading data redundantly by original and backup reliTXs can detect faults in memory buses as well.

5.4.4 Overheads

FaulTM presents 100% core overhead as previous redundancy based reliability schemes on CMPs [30, 31, 36, 37]. During the execution of a sequential application in a multi-core architecture, only one core is occupied and the others stay idle. FaulTM leverages one of the idle cores for the reliability purpose which supplies the capability of detecting both transient and permanent errors.

FaulTM does not present any overhead for the creation of ReliTXs since in a lazy-lazy HTM, creating a transaction means starting to write the values to the local log area instead of writing to the shared memory. However, the backup reliTX is obliged to copy the register file and TLBs from the original thread to be able to produce the same results. Fortunately, this copy operation is not required to be done when the transactions are back-to-back. Nevertheless, this copy operation is not vulnerable to hardware errors because, if a strike changes the value of some data on the bus, this would cause the final results of pair transactions to be distinct thus the error is detected.

In FaulTM, out-of-order memory operations may cause false positives. Although we detail FaulTM for in-order executions, a straightforward solution for out-of-order cores is storing only retired instructions to writesets.

5.5 Evaluation

In this section, first we explain the evaluation environment then we present our experimental results. In the experimental results, we first evaluate the performance overhead of FaulTM for sequential and parallel application in the error free execution. Second, we evaluate the reliability performances of FaulTM by utilizing FimSim fault injector. Finally, we evaluate the recovery overhead of lightweight checkpointing mechanism of transactional memory.

5.5.1 Simulation Setup

We use the M5 full-system simulator [52] with an implementation of a Hardware Transactional Memory system that uses lazy data versioning and lazy conflict detection [69]. We extend this simulator with our FaulTM implementations. We evaluate our schemes by using in-order Alpha 21264 cores [53] with L1D and L1I caches and a unified L2 cache. Each L1 cache is 64KB with four-way set associativity, and a two-cycle hit latency. The L2 cache is 2MB with eight-way set associativity, and 10 cycles of hit latency. All caches are write-back with a line size of 64B and a local HTM write-set buffer with 32 entries. Main memory latency is 100 cycles. We evaluate FaulTM using spec cpu2006 [54] benchmark suite with test data-set by executing either 2 billion instructions or until application termination in 1 (original) thread (i.e for FaulTM with also 1 more backup thread). For the evaluation of FaulTM for multi-thread applications, we use 4 original and 4 backup threads using splash2 [77] and stamp [78] benchmark suites which are the representative of lock-based parallel and TM applications.

For the time overhead of error detection in the error-free execution, we compare FaulTM system against off-core lockstepping, a standard error detection method. In the lockstepping approach, after every store instruction, two threads synchronize and the results of the store values are compared as we explained in Chapter 2. Note that, validation of store values is common in several recent redundancy-based error detection techniques [27, 30, 31].

We use FimSim fault injection infrastructure that we explain in Chapter. 3 to measure the reliability performance of FaulTM for both transient and permanent faults. We inject the faults to five different structures in a core; instruction opcodes, program counter (PC), integer register file (int-RF), special purposed register file (RF-special) and arithmetic logic unit (ALU). We did not inject faults to TLBs since our experimental results in Chapter. 3 shows that TLBs are not vulnerable to transient and intermittent faults. Also we did not inject faults to caches and main memory since these structures are out of the coverage of FaulTM.

We inject 100 faults per structure in each application to a random location in each structure at a random time after warming up 200M instructions. We perform one injection per simulation. While we flip the chosen bit for transient fault injection, we use stuck-at-0 and stuck-at-1 models for permanent faults. We simulate 10M cycles after fault injection to observe the effect of the injected fault.

For the time overhead of error recovery, we compare transactional checkpointing with Rebound [46], the state of the art checkpointing scheme. In Rebound, whenever a core checkpoints its execution, it writes the dirty lines to the shared memory (or main memory) which is protected by other means such as error correcting codes. This scheme is quite similar to our scheme in which write-sets are written to shared memory in every checkpoint interval (i.e. in every reliTX). In Rebound, when core A checkpoints, all other cores which produce data used by core A checkpoint (i.e. all *producers* of core A checkpoint). Also, when core A needs to recover due to a detected error, all the cores which has consumed any data produced by core A also recovers (i.e. all *consumers* of core A recovers). For comparing TM-based error recovery with Rebound, we execute splash applications in 32 threads by using PIN [79], a dynamic binary instrumentation tool to instrument the execution.

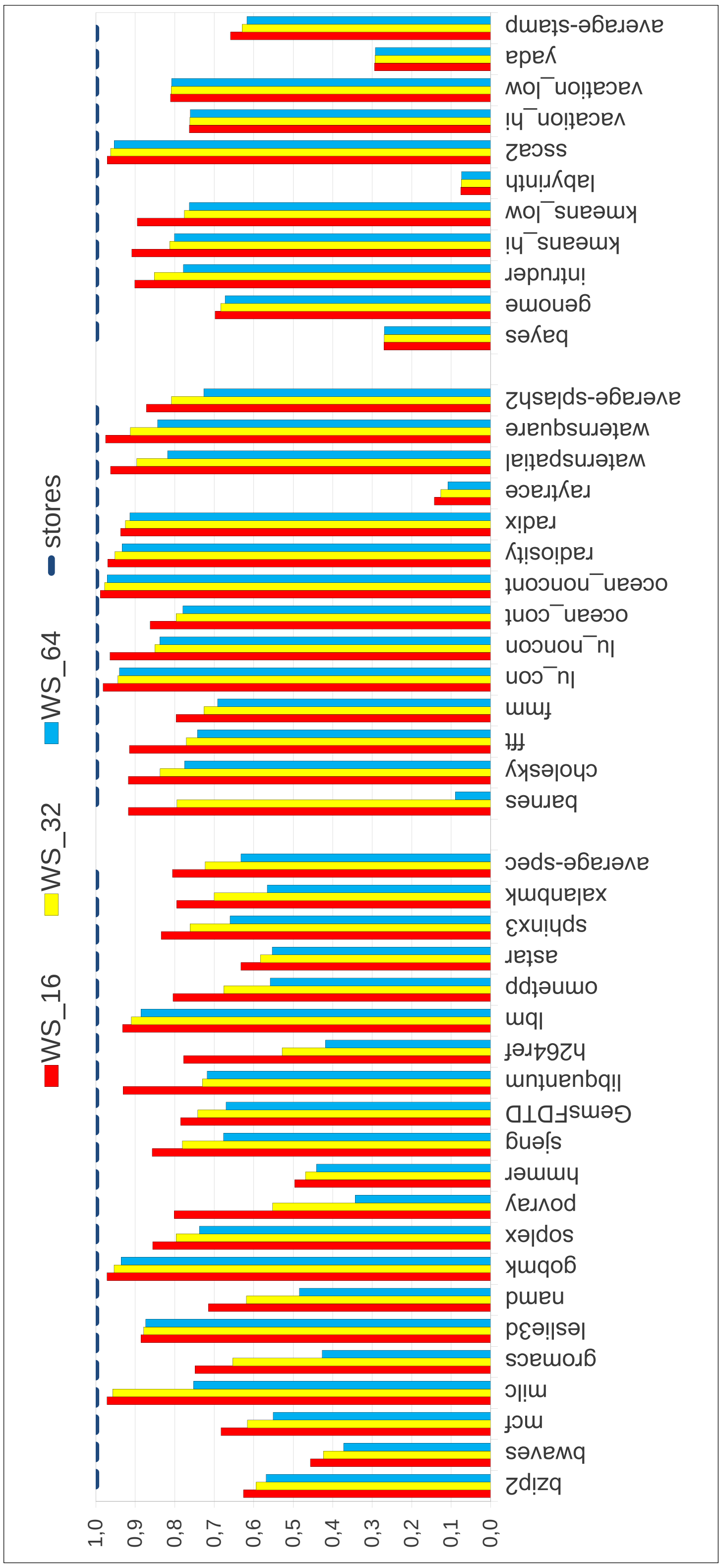


Figure 5.11: Total entries in write-sets normalized according to the total number of store instructions in FaulTM.

5.5.2 Performance Overhead in the Error-Free Case

FaulTM reduces store value comparison overhead since it compares the write-sets (instead of each store values) which have fewer amount of entries than total number of store instructions in transactions due to multiple stores to same addresses. In Figure 5.11, we present the normalized value of the total amount of entries in write-sets according to total number of store instructions in each application on different write-set sizes (i.e. 16, 32 and 64 entries which is determined during the design time of an HTM). In the figure, each bar represents the normalized value of the total amount of entries in write-sets as compared to total stores. The dashed line in the graph shows the normalized value of stores in applications. Note that, for the rest of the experiments for FaulTM we use 64 entries for the write-set.

We find that, in our benchmarks, the percentages of entries in write-sets are smaller than the percentages of all stores (on average, 35% less among all benchmarks when WS=64 entries), because some store instructions in transactions write to the same addresses multiple times. For instance, in raytrace, labyrinth and barnes with WS_64, write-sets have around 90% less entries than stores. When we increase the size of the write-set, we reduce the total number of entries in write-sets since the temporal locality of the store addresses increases. Due to the temporal locality of the stores, our FaulTM technique requires fewer comparisons compared to lockstepping in order to check errors.

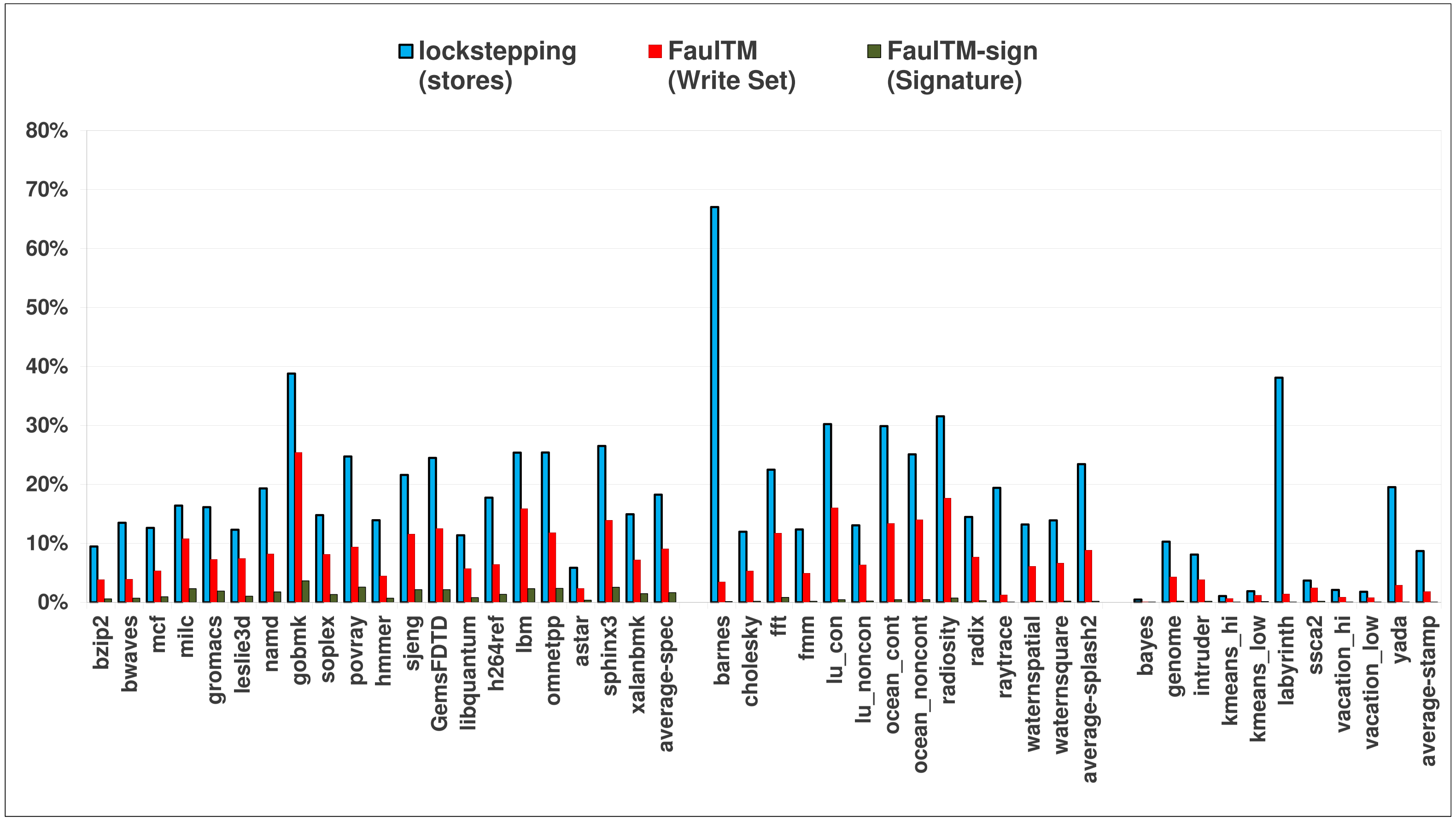


Figure 5.12: The Error Detection Overhead of FaulTM vs Lockstepping in the Execution Time

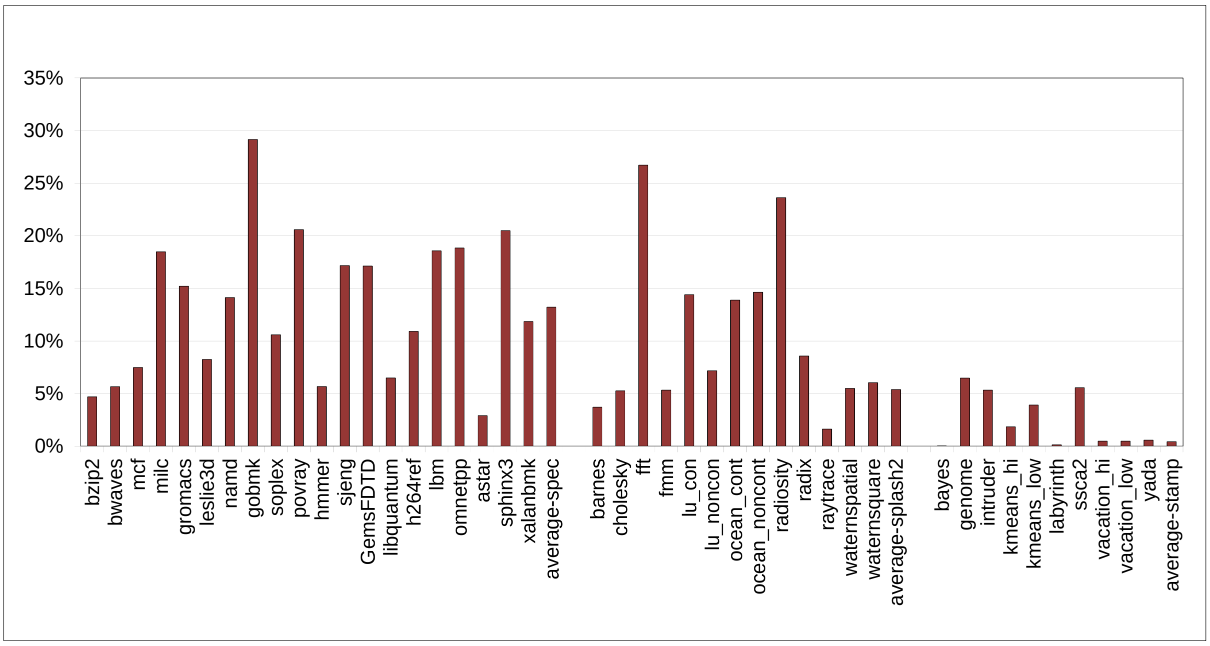


Figure 5.13: Register File Comparison Overhead

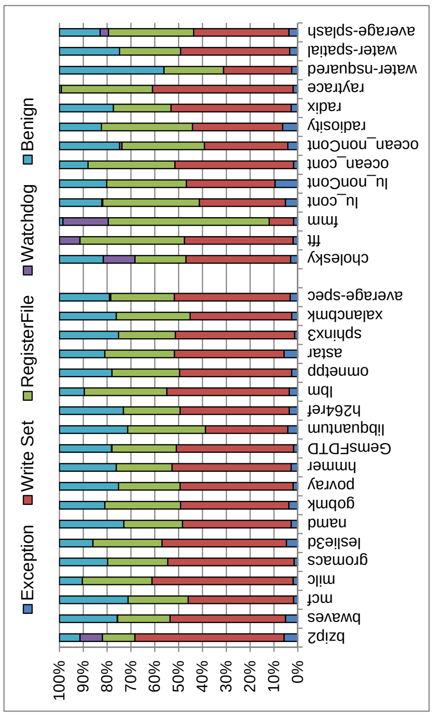
In Figure 5.12, we compare the performance overhead of error detection of FaulTM in error-free case with lockstepping including comparison and spin overheads. Note that we did not include the error recovery overhead in the graph. We use 64 entries in the write-sets of FaulTM. We assume that each comparison (e.g. comparison of either a store value or an entry in WS) can be accomplished in one cycle on an idealized bus where collisions are not modelled. This favours lockstepping since lockstepping is penalized more when the latency of the bus is higher. Spin overhead is, on average, 1 cycle for lockstepping and 4 cycles for FaulTM according to our simulation results.

Compared to lockstepping, FaulTM reduces the performance degradation by 2.5X for SPEC2006 benchmarks. We find that, on average, the performance degradation of our approach in error detection is 8% for spec2006, 10% for splash and 2% for stamp applications which are 56% and 75% less than lockstepping for splash2 and stamp applications respectively. In Figure 5.12, lockstepping fares poorly for two applications in particular: Barnes from splash2 and Labyrinth from stamp. Barnes is a particularly resource-hungry application with respect to stores; Kirman et al. [80] also comment on this issue and describe the challenges they faces when trying to optimize the store queue design for Barnes. Labyrinth, on the other hand, features large transactions writing to local copies of a 3D grid, and exhibits high contention, mostly due to false conflicts on the same cache line [81]; this increases the comparison overhead for lockstepping. In the figure, FaulTM-sig shows the performance degradation of FaulTM when hash-based signatures are compared instead of the entire write-sets. Note that, error coverage of signature comparison is not 100%.

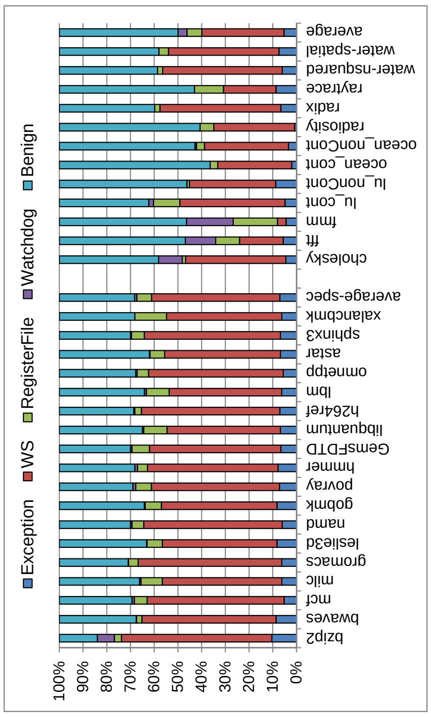
In Figure 5.13, we present the comparison overhead of the entire register file in FaulTM (13%) in order to guarantee the error-free execution since the last validation. Register value comparison after each instruction in lockstepping would cause extremely high overhead (not included in the evaluations here). For instance, in the LiVe design which accomplishes register file comparison on top of lockstepped cores, the overhead can be very huge when the register values are compared in the short interval to detect every fault immediately after they occurred. Note that the register file can also be included in the signature value in FaulTM-sig which compares signatures instead of the entire write-set and register file.

5.5.3 Reliability Performance

In this section we evaluate the reliability performances of FaulTM.



(a) Transient Faults



(b) Permanent Faults

Figure 5.14: The figure presents the error detection performance of FaulTM for transient and permanent errors.

Figure 5.14 presents the reliability performance of FaulTM. For parallel applications, the potential problem is shared variables that might propagate errors to other cores. We avoid this problem by performing the reliability verification just before publishing the write-set to shared memory. According to our fault injection experiments with a 10M-cycle tracking window (we wait 10M-cycle after injecting a fault), our FaulTM design provides 100% error coverage for both transient and permanent faults with the comparison of entire write-set and register file. In the figure, we present the rate of the faults which is detected in the relevant mechanism. These mechanisms are detecting

1) a fatal trap exception in a reliTX,

2) a mismatch between write-sets of reliTXs,

3) a mismatch between register files of reliTXs,

4) an error which cause a time-out in the watchdog mechanism.

FaulTM, also, avoids detecting benign faults which is 20% of injected transient faults and 39% of injected permanent faults. There are several cases that end up as a benign fault for transient faults. One such case is when a fault (e.g. in a register) is overwritten by a new value before being read, this case is not detected by any reliability scheme. In a second case, the faulty value is read by an instruction however the fault does not change the result of the instruction, an example would an arithmetic operation that masks the fault. A third case occurs when an instruction stores the faulty value but this value is then overwritten by another store to the same location before it is read by another instruction, this is the case of the so-called silent store. Note that lockstepping would detect the silent store as an error and produce a spurious error signal; while FaulTM would filter it out since it only sends the overwritten store value to main memory. We present a similar example for register instructions.

```
Ex:  I1:   R3(faulty) = R1(faulty) + R2
     I2:   R1 = R7 + R5
     I3:   R3 = R3(faulty) AND 0x00
       ---commit---
```

In the example, I1 reads R1 which is faulty and it propagates the fault to R3. I3 then reads faulty R3 and by AND ing with zero removes the fault on R3. Also, I2 writes over R1 which masks the transient fault on R1 (unnecessarily detecting a benign fault can be considered as a "false positive" from a reliability view point). We find that 4% of transient faults (not in the graph) are treated as error in Fingerprintig [37] while FaulTM does not try to recover these

Benchmark	Size	Benchmark	Size	Benchmark	Size	Benchmark	Size	Benchmark	Size
401bzip2	1139	456hmmer	980	410bwaves	1120	458sjeng	379	429mcf	823
459GemsFDTD	352	433milc	408	462libquantum	772	435gromacs	607	464h264ref	683
437leslie3d	591	470lbm	277	444namd	536	471omnetpp	373	445gobmk	173
473astar	1876	450soplex	543	482sphinx3	317	453povray	469	483xalancbmk	612
barnes	2548	cholesky	909	fft	149	fmm	1634	lu_con	516
lu_noncon	387	ocean_cont	642	ocean_noncont	623	radiosity	110	radix	705
raytrace	5801	waternspatial	458	waternsquare	421	bayes	152716	genome	567
intruder	591	kmeans_hi	1886	kmeans_low	1043	labyrinth	50314	ssca2	505
vacation_hi	6545	vacation_low	6520	yada	7668				

Table 5.1: Average Number of Instructions in a reliTX.

benign faults and it does not cause a false positive signal since they are masked before the end of reliTXs. We believe that using larger transactions is effective to reduce the false positives since it increases the probability of that some instructions may mask benign faults within transactions. For permanent stuck-at faults, they are benign if a same bit value is written with the stuck value (e.g. a bit cell which is generally 0 is stuck at 0).

In case of comparing signatures instead of comparing entire write-sets, 5 transient errors (among 155000 transient fault injection in spec2006 applications) are not detected in FaulTM-sig (not in the graph), thus, in extremely high reliability requiring systems, signature usage might not be appropriate.

The error detection rate of register file comparison is 30% for transient faults which presents the importance of register file comparison.

5.5.4 Error Recovery Overhead

For error recovery, reliable systems require additional checkpointing mechanism which presents checkpoint creation and recovery overhead. Table 5.1 shows the average number of instructions in transactions when the write set size is 64 entries. In FaulTM, the number of instructions in transactions is very low (generally less than 10K instructions) compared to system-wide checkpointing mechanisms (about 10M instructions) such as ReVive [43] or SafetyNet [47]. Note that, smaller checkpoint interval is essential to support I/O operations [37].

The error recovery overhead of FaulTM comes from two reasons. First, it needs to write the write-sets to the shared memory. Second, after an error is detected, it re-executes the instructions from the beginning of the reliTX. Similarly, in Rebound, each core writes the dirty lines to the shared memory whenever they create a checkpoint at every 100M cycles.

In Figure 5.15, we compare the number of checkpoints created in FaulTM and in Rebound.

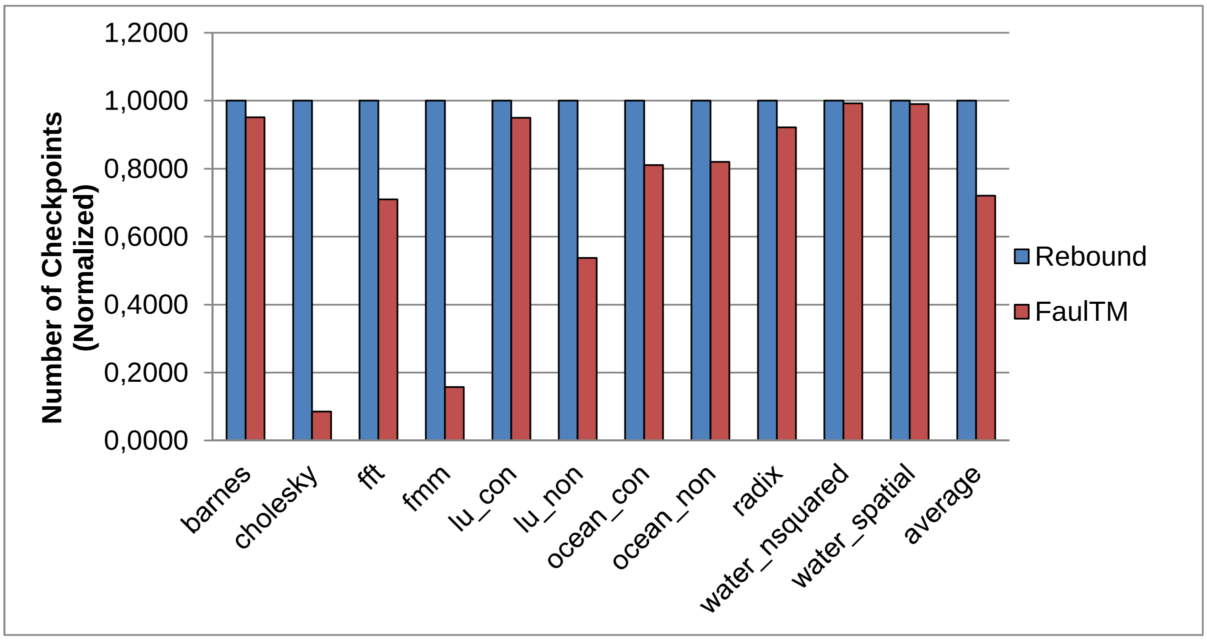


Figure 5.15: The figure compares the number of checkpoints created in FaulTM and Rebound in the error free execution

To be able to provide a fair comparison, we create checkpoints in Rebound and set the size of reliTX according to number of instructions executed. Thus, FaulTM sets the size of reliTXs as 10000 instructions and Rebound checkpoints a core when it executes 10000 instructions since the last checkpoint creation. Note that, in the normal case, FaulTM checkpoints a core when the write-set is full. In Rebound, when a core checkpoints, all producers of the core also checkpoint thus some checkpoints include less than 10000 instructions. FaulTM, on the other side, presents a lightweight checkpointing scheme which reduces the number of checkpoints by 28% compared to Rebound. Note that, besides writing back the dirty lines, before and after each checkpoint creation Rebound has an additional overhead of synchronization.

In order to provide a close look, in Figure 5.16, we present the number of instructions executed in each checkpoint for cholesky (In cholesky, $\backsim$800M checkpoints are created). As it can be seen from the figure, many checkpoints are created with the instruction numbers much lower than the normal checkpoint interval.

After the error is detected, FaulTM re-executes the instructions only in **one** reliTX pair. Rebound, on the other hand, recovers all the consumers of the erroneous core if there is any.

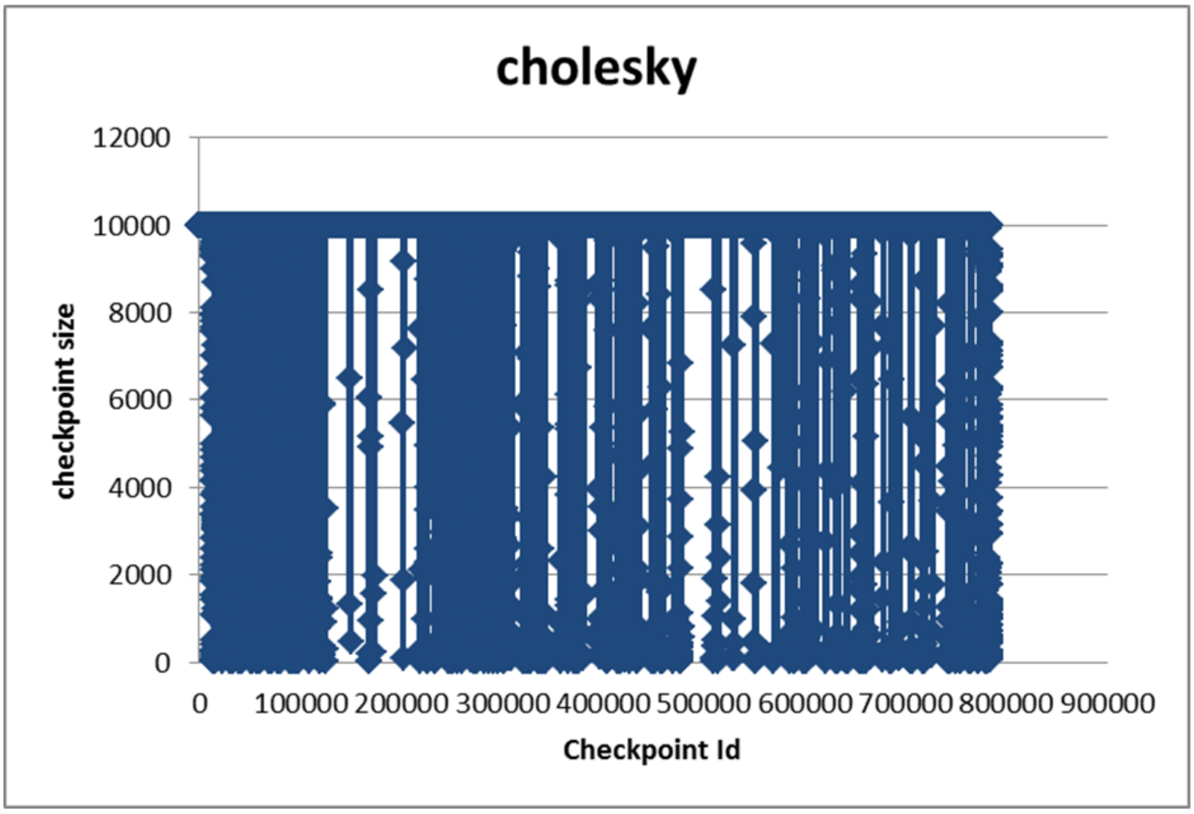


Figure 5.16: The figure presents the size of the checkpoints created by Rebound for cholesky. In Rebound, when a core checkpoints, its producers also checkpoint, thus, some checkpoints have less instructions than the checkpoint interval (i.e. 10M instructions).

Obviously, FaulTM presents less recovery overhead than Rebound. In Figure 5.17, we present this reduction visually for one application, barnes. In the Figure, we present the number of cores which requires recovery for barnes application in case of an error is detected in the related checkpoint. At some point, all cores (i.e. 32) need to be rolled back due to communication within checkpoint interval while FaulTM rolls back only the erroneous core.

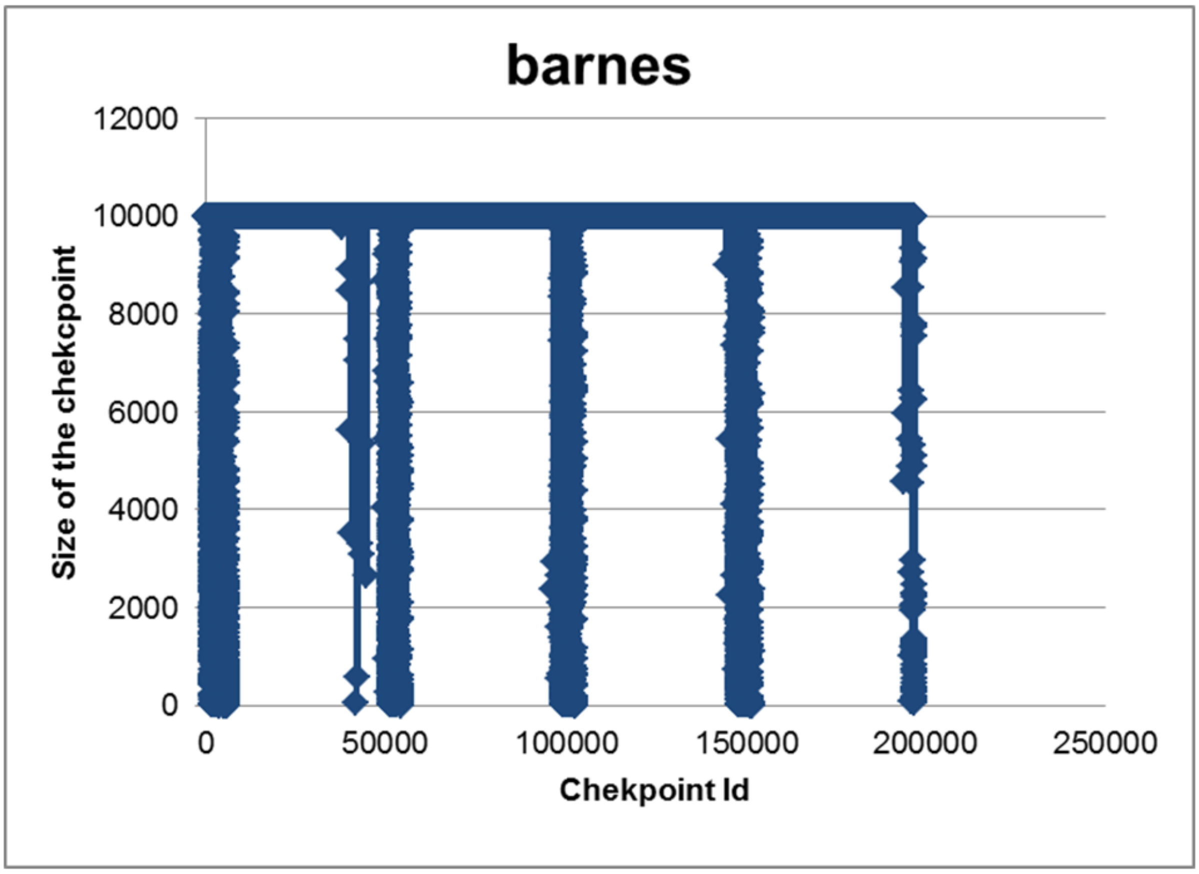


Figure 5.17: The figure presents the number of cores which require recovery in case of an error detection in barnes application. At some point, all cores(i.e. 32) need to be rolled back due to communication within checkpoint interval while FaulTM rolls back only the erroneous core.

5.6 Other Reliability Schemes utilizing Transactional Memory

After we introduce SymptomTM and FaulTM, there have been further reliability proposals leveraging Transactional Memory semantics for reliability. In this section, we present these further studies.

In FaulTM the execution is stalled at the commit stages of the transactions since transaction pairs are tightly coupled. Also, after the verification of the correct execution of transactions, buffered values must be visible to the rest of the system. This commit process presents a pressure over the memory hierarchy and the performance of the system. In order to provide an alternative design overcoming these limitations, Log-Based Redundant Architecture (LBRA), a reliability scheme build on an eager data versioning HTM system is proposed [82].

In LBRA, the thread pairs are termed master-slave threads akin to original-backup threads of FaulTM. The master thread executes the transaction but, additionally, it keeps the results of its progress (i.e. Verification Signature which summarizes the computation performed during the execution of the transaction) in a pair-shared log. By means of this log, the slave verifies that the results produced by master are correct. Using the eager-eager HTM, LBRA decouples the pair transactions so that master transaction can commit without being stalled for the execution of the slave transaction to be finished. However, it requires additional implementation of three mechanisms: i) *input replication*: Since master and slave transactions are not decoupled, the execution of redundant memory instructions would probably lead to input incoherence. Input replication is required to solve this issue. LBRA extends the log area provided by the TM to contain the history related to memory operations for the purpose of keeping track of the data values that the master thread accesses. The load instruction of the slave transaction is served through this log (in program order), thus, slave thread obtain the same value as its master-pair. ii)*Providing a stable recovery state*: As memory values are allowed to be shared and shared memory is eagerly updated in LBRA, potential faults could be propagated across the system. Thus, when a fault is detected in a transaction and this transaction aborts, all other transactions using the data produced by this faulty transaction (i.e. consumer of the faulty transaction) should also abort. To this purpose, in LBRA, master thread track the producer/consumer dependencies with other threads in the system by means of the conflict detection support provided by TM. Thus, when a faulty transaction aborts, it also sends an abort request to all its consumers. iii) *output comparison*: If a thread is the consumer of another thread, the validation of the consumer thread is accomplished after the validation of its producers.

Wamhoff at el [83] proposed Transactional Encoding which combines arithmetic codes for error detection and Software Transactional Memory for error recovery. They provide the design for both lazy and eager conflict detections. They use AN arithmetic codes together with symptom-based error detection. Besides the advantage of encoding processing that allows executing non-deterministic applications, the software-based Transactional Encoding also provide achieving reliability using unreliable commodity hardware. However, software-only solutions can not guarantee the detection or the recovery of permanent errors since replicas or recovered executions can be issued to the same hardware structures.

	Data Versioning - Conflict Detection		
	lazy-lazy	lazy-eager	eager-eager
Checkpointing Overhead	High	High	**Low**
Recovery Overhead	**Low**	**Low**	High
Error Containment	**High**	**High**	Low
Error Detection Latency	High	**Low**	**Low**
Error Detection Overhead	**Low**	High	High

Table 5.2: Reliability attributes of different TM implementations (Bold is Better).

In the transactional encoding, first, the application written in C is transformed to the encoded version by using an encoding compiler. During this transformation, the main module initializing the application is not encoded. Also, encoded versions of public functions and wrappers of those functions are added to the encoded application. Wrappers encode the parameters for the function, call the encoded version of the function and then decode the returned value (if there is any). In the second step of the transactional encoding, transactional memory semantics are added to the encoded version. In this sense, Transaction Begin/End instructions are added to the wrapper. Also, all accesses to the state are redirected to the TM by invoking read and write operations from the encoded functions. Also, before reading/writing to/from memory, the memory address is decoded so that the correct memory location can be accessed.

The error detection can be accomplished in either lazy or eager manner similar to the conflict detection of TM. If the error detection is accomplished eagerly, all transactional writes conduct decode operation of encoded processing for the written data. Otherwise, if the error detection is deferred until transaction commit, the entire write-set is decoded only in the commit stage. If transactional encoding detects a divergence of any value from the valid state, it aborts the transaction and starts it from the beginning. Otherwise, the transaction commits and all the memory operations can visible by the system.

5.6.1 Discussion: Pros and Cons of TM design parameters for Reliability

TM proposals implement two key mechanisms: data versioning and conflict detection. Each of these mechanisms can be implemented either in lazy or eager policies. Out of four possible combinations of these policies, the lazy-lazy [1], lazy-eager [84], and eager-eager [85] schemes are the most popular implementations. In the rest of this section, we provide a succinct discussion of the impact of TM policies on reliability by considering five desirable features for a reliable system:

(1) Low checkpointing overhead,

(2) low recovery overhead,

(3) high error containment, to limit the propagation of errors in the system,

(4) low error detection latency, to detect errors as soon as possible, and

(5) low error detection overhead.

In Table 5.2, we summarize effects of the data versioning and conflict detection policies on reliability. As we show in the table (bolds typeface denotes the desired properties), none of the possible three TM policy combinations has all these features.

The cost of providing checkpoint/rollback behavior depends mainly on the data versioning strategy. *Lazy data versioning* works in two stages, a pre-commit phase and a commit phase. In the pre-commit phase, the modifications are made on private copies and at the commit phase, these modifications are written to the memory. Since the modifications within transactions are repeated—at least once for the private copy and once for the shared memory—a significant overhead is introduced for checkpointing even for error-free executions. However, it provides a very fast error recovery. *Eager data versioning* performs in-place memory updates during transaction execution and introduces overhead only upon abort, i.e., upon error recovery. The abort overhead is caused by the replacement of modified versions of data with their versions prior to the transaction. Thus, eager data versioning presents less overhead for checkpointing compared to lazy data versioning, however, its recovery overhead is much higher than the lazy data versioning. Eager data versioning is preferable in terms of performance and energy efficiency when the error rate is low and the system presents few rollback. On the contrary, when the error rate is high (e.g in low Vdd or when the hardware is located in a high attitude), using lazy data versioning is preferable since the system would require many rollbacks and a rollback for lazy data versioning is cheaper in comparison to eager data versioning.

In TM implementations with eager data versioning, main memory keeps the latest speculative version of the data. If we use eager data versioning for reliability, some data in the shared memory which is not validated for being error-free, can be read by other cores. Assuming any of these data or any address is erroneous, this error might then easily propagate to concurrently executing tasks. Therefore, error propagation in eager data versioning is high while lazy data versioning presents high error containment. Thus, eager data versioning requires additional synchronization mechanisms for error recovery in order to rollback the communicating tasks when an error is detected in a transaction. Due to the error propagation, all transactions executing in the systems may require recovery. On the contrary, in lazy data versioning, only error-free data is written to the shared memory, therefore any error occurring in a certain transaction does not propagate to the other transactions through memory. Thus, only the erroneous transaction rolls back while the rest of the system keeps running without wasting any error-free work done.

For Error Detection, from time to time the normal process should stop and the error detection operations (e.g. in redundancy-based error detection, comparing the results of instructions) should be carried out. Thus, the higher the number of error detection is triggered, the higher the potential performance degradation due to error detection is presented. In TM systems, error detection is accomplished during the conflict detection time of TM. For instance in a redundancy-based reliable system utilizing TM with lazy conflict detection (e.g. FaulTM), the comparison operation is carried out at the commit stage of the transaction. On the contrary, for eager conflict detection, the error detection should be carried out at every time the shared memory is updated (i.e. every write operation). Therefore, we could conclude that potential performance degradation of lazy conflict detection is lower.

On the other hand, in lazy conflict detection any error occurring earlier in the transaction will only be detected at the commit stage, so error detection latency will be higher. In eager conflict detection, however, the error could be detected earlier when a transactional store containing the error is compared.

5.7 Summary

In this chapter, we introduce FaulTM, a redundancy-based error detection and recovery approach leveraging a lazy-lazy hardware transactional memory (HTM) system. FaulTM provides an efficient error recovery mechanism by utilizing the local checkpointing mechanism of TM. Also, it reduces the comparison overhead significantly by comparing the redundant execution streams at the end of the transactions instead of after every store instruction while avoiding error propagation to the whole system by utilizing the isolation property of transactions. Moreover, it eliminates the requirement of a separate input replication mechanism by utilizing the conflict detection scheme of TM.

Comparatively, FaulTM has a negligible transaction creation and abort overhead. The average number of instructions in reliTXs are generally less than 10K instructions which is very low compared to system-wide checkpointing mechanisms (10M instructions) [43, 47] that reduces the re-execution overhead of the instructions from the beginning of the reliTX for the error recovery. Moreover, smaller checkpoint interval is essential to support I/O operations [37].

6 Energy Reduction in Microprocessor

The increasing power and energy consumption of modern computing devices is perhaps a large threat to technology minimization and associated gains in performance and productivity. For instance, current scaling trends have led to multi-core processors at the architectural level, and higher core counts are expected in the following years. Yet, it will not be possible to keep the whole chip powered-on at the same time due to power envelope issues, a problem also known as the dark silicon phenomenon [86].

As the power envelope becomes one of the key design concerns, a dramatic improvement in the energy efficiency of microprocessors is required in order to keep the power under control. Since energy consumption grows proportionally to the square of supply voltage (i.e., $Energy \approx C_L \times V_{dd}^2$), a very effective approach in reducing the energy consumption is to reduce the supply voltage (V_{dd}) below the safe margin, close to the transistors' threshold (near-threshold execution). Voltage downscaling can offer substantial energy savings by trading off performance. To take advantage of potential power savings, microprocessors have started to provide high-performance and low-power operating modes [87]. While the processor runs at a high frequency by using high V_{dd} in the high-performance mode, in the low-power mode, the processor reduces V_{dd} and the frequency.

However, the energy reduction in the low-power mode comes with a drastic increase in the number of failures [8]. As V_{dd} decreases, failure rates for yield loss, hard errors, erratic bits, and soft errors increase.

Dan et al. proposed Razor [2], a hardware solution for tolerating failures caused by lowering supply voltage. Razor extends the pipeline stages with a new circuit design in order to detect timing errors. Obviously, this circuit extension presents a hardware overhead which is not used under nominal voltage. EnerJ [88], a software framework for approximate computing, is proposed by Sampson et al. EnerJ executes the non-critical sections of applications in the low-power mode while the rest is executed under nominal voltage. Programmers specify, which parts should be approximate, thus programming language support is required. The architecture has to support separate areas in memory and CPU for which the voltage can be reduced. In this chapter, our goal is seeking architectural solutions for energy minimization without requiring any substantial hardware changes or programming mechanisms.

In order to fully exploit the dynamic energy savings of voltage downscaling, a potentially attractive idea is to implement reliability solutions that allow a system to operate below the safe margin of V_{dd}. In this chapter, we investigate the usefulness of TM-based reliability schemes for reducing energy consumption.

Dependable multiprocessor systems combine error detection and error recovery to provide reliability in the safe margin. These systems such as embedded systems and supercomputers, mostly rely on checkpointing/recovery for error recovery [89]. However, when the voltage of processor is reduced, a poorly implemented checkpointing/recovery mechanisms may consume more energy than the saved one. Therefore, for error recovery, we are interested in utilizing Transactional Memory which provides a lightweight mechanism.

Transactional Memory (TM) was originally introduced to simplify the process of parallel programming [65]. It also provides automated checkpointing/rollback, hence, it is also used for implementing reliability as we present in Chapter 4 and Chapter 5 as well as shown in [82, 83, 90]. So far, it has been shown that the use of software TM (STM) consumes less energy than traditional lock mechanisms [91]. The authors investigated typical parallel benchmarks and measured the energy consumed, which motivates us to investigate further the pros and cons of combining error detection mechanisms with TM if processors work on low voltage levels. We believe that TM can simplify the process of energy-efficient reliable programming in a similar way. Researchers showed that the use of TM can consume less energy than traditional lock-based mechanisms, for micro benchmarks [92] and also for the more sophisticated STAMP benchmarks [91]. Moreover, we also show that the implementation complexity of reliability schemes can be minimized by using existing TM hardware with minor changes in Chapter 5. Hence, it is worthwhile to further investigate TM in combination with different error detection mechanisms for processors working at low voltage levels.

In this chapter, we investigate the edge cases on voltage reduction while the error recovery still leads to a reduced energy consumption. In the following section, we present the error rate in combinational logic under scaling voltage. In Section 6.2, we summarized how TM can be utilized for error recovery. In Section 6.3, we survey different existing error detection mechanisms. In Section 6.4 we evaluate the energy reduction provided by these mechanisms.

6.1 Voltage Scaling and Error Rate

The energy usage of computer systems is becoming more important, especially for battery operated systems such as mobile devices. One solution for efficient energy consumption is power-down-when-it-is-idle which is used in many laptops. For instance, in mobile devices it is preferable using "turbo mode" [93], where Vdd is increased to run faster and switch off not only the processor, but the screen, antennas, etc. as soon as possible. Also, those systems power-down the disk when it is not accessed. In the other option, the chip speed is dynamically varied. In this way, the energy consumption also varies and at the peak moments, the user sees the high number of instructions per second while when the user is not active, less number of instructions is executed per second with less energy consumption.

Voltage reduction is currently the most promising way to save energy [94, 95]. The intuition behind the power savings come from the basic energy equation that is proportional to the square of the voltage. To lower the voltage and still operate correctly, the frequency (the cycle time) should also be lowered. Since, frequency and the voltage should be adjusted together, obviously lower voltage and frequency dissipate less energy per cycle.

The question here is whether the voltage reduction is better for energy saving. Suppose that a task has a deadline in 100 miliseconds but it takes 50 miliseconds to complete when the system is running at full speed. In the first option, the system runs at full speed for 50 miliseconds and then stops for the next 50 miliseconds by putting into a hibernate mode that wakes up upon an interrupt. Note that the energy consumption in the hibernate mode can not be zero. In the second option, the system runs the task at half speed also by reducing the voltage by half and completes it just before its deadline. In this case, the task would consume 1/4 the energy of the normal system. However, when the supply voltage is reduced below the safe-margin (i.e. A voltage is in the safe margin if the circuit runs without errors in more than 90 percent of the baseline clock period), the error rate increases drastically [2]. This is because, as the voltage drops, more internal circuit paths cannot complete in the clock period. Clearly, if pipeline can tolerate more errors, it can operate much lower supply voltage level.

6.2 Error Recovery

In order to reduce the supply voltage lower than the safe margin, it is required to include a reliability scheme in the system. The more error is tolerated by the reliability scheme, the more the supply voltage can be reduced. However, although lowering the voltage provides energy saving, the reliability scheme presents additional energy consumption.

Reliability has two main aspects: error detection and error recovery. As we explain in Chapter. 2, checkpoint/recovery provides a backward error recovery scheme. There are three main strategies in checkpoint creating: global, coordinated-local and uncoordinated-local. As we presented in Chapter. 4 and Chapter. 5, checkpointing can also be supported by transactional approaches. In this way, error recovery is build on top of an existing hardware with minor modifications so that the implementation complexity is reduced. Initially, Transactional Memory (TM) have been proposed as a data synchronization mechanism. While TM provides checkpoint/rollback behaviour for free, it can easily be adopted for recovering from transient faults. The semantics of transactions proposed by TM inherently support failure atomicity, i.e., TMs guarantee that an operation, which cannot complete due to an unexpected event, can rollback to a state prior to its invocation. Several researchers have exploited this fact to enhance exception handling mechanisms [96, 97, 98, 99]. Similarly, Oplinger and Lam [100] encapsulate error-prone parts of the code within special transactional blocks in order to recover from faults. However, these studies mainly target faults induced by software development. In this section, we discuss how TM can be adopted for error recovery.

6.2.1 Adapting TM for Recovery

Although TM (and especially STM) is known to have a high overhead for certain workloads, a significant portion of this overhead is due to data synchronization when detecting whether different threads accessed common data. For error recovery purposes, however, only the checkpoint/rollback behaviour is necessary and the synchronization requirement is therefore largely reduced. Hence, it is possible to design cost-effective TM for error recovery by providing minimal synchronization (e.g., Chapter5). Such TM designs can easily provide coordinated-local checkpointing.

The cost of providing checkpoint/rollback behaviour depends mainly on the logging strategy. *Redo-logging* (lazy data versioning) works in two stages, a pre-commit phase and a commit phase. In the pre-commit phase the modifications are made on private copies and at the commit phase these modifications are written to the memory. Since the modifications within transactions are repeated—at least once for the private copy and once for the shared memory—a significant overhead is introduced even to error-free executions. *Undo-logging* (eager data versioning) performs in-place memory updates during transaction execution and introduces overhead only upon abort, i.e., upon error recovery. The abort overhead is caused by the replacement of modified versions of data with their versions prior to the transaction.

While having lower time overhead, undo-logging can easily result in the propagation of a fault between concurrently executing tasks, because speculative changes become effective immediately on shared memory locations. Therefore, a synchronization mechanism is needed for error recovery. High fault rates can cause important overheads and it may be more interesting in such cases to use redo-logging. This would reduce the synchronization costs because fault propagation can only occur during transaction commit. Furthermore, a rollback for redo-logging is cheap in comparison to undo-logging. Thus, in the highly faulty environments, it is better to use redo-logging. Since the fault rate under the scaling voltage is drastically high, we limit the scope of this chapter to redo-logging.

Using undo- and redo-logging simultaneously can increase the recovery and reliability provided by the TM. The rationale is to use undo-logging for enabling rollback, while using redo-logging to build a (correct) history of updates done by the transaction until the error is detected. The introduced overhead is limited to an additional replication of write operations, while the operand is already in the cache. Since transaction rollback cannot be guaranteed to be error-free when operating at low voltage, the history can be used as FER during the re-execution of the transaction to mask any faults. Doudalis and Prvulovic [101] proposed combining undo- and redo-logging to support both bidirectional debugging and error recovery, which can be another research direction for TM that we will not further discuss about it.

6.2.2 Executing with TM for Recovery

In order to integrate TM within an error recovery scheme, the code that requires recovery should be executed within transactions regardless of whether the original code includes transactions or not. We name the process of executing a code within a transaction as *transactification*. The need for transactification raises two issues: (1) determining the scope of transactification, (2) choosing the right transaction granularity (the decision of when transactions start and commit).

Within the context of voltage scaling, a TM should be capable of taking control of the executed code at any time, since the voltage level can be reduced at an arbitrary moment. This implies that the scope of transactification should span the entire code, except the code that explicitly declares that transactification is not needed, e.g., non-critical sections in approximate computing. If STM is used, all code (not only applications but also operating system code) running on a machine should also have a transactional version to switch to transactional execution at any time. For a hardware TM (HTM), transactification is done transparently in hardware, but the size of a transaction is limited. If the size of transaction can be kept small, it is possible to use an HTM alone. Otherwise a hybrid TM is required, i.e., where the HTM limitations are exceeded, STM is used as a fall-back.

Determining where the transactions start and end during low voltage operation is also an important issue. Inserting the executed code inside a single transaction (during the whole time the core operates at low voltage) is not feasible, since this requires an unbounded buffer in order to store the unmodified states of all modified data (the transaction size cannot be known in advance). Hence, once a core starts operating at low voltage we need to execute the code inside back-to-back transactions with known write-set sizes. It is further important to take care not to miss errors if there is a delay between occurrence of a fault and its detection. Otherwise, an instruction might be already committed even though the execution was faulty. It is possible to introduce delays to ensure that all the instructions within a transaction completed without errors. At this point, the choice of the transaction granularity is critical. Small transactions permit efficient TM implementations (e.g., HTM) but may introduce too many artificial delays, slowing down the error-free execution. Large transactions can hide the artificial delays, but make it difficult for the TM to be efficient (e.g., requiring STM at least as a fall-back). Also, the probability of detecting an error in a transaction increases for larger transactions thus it increases the recovery overhead.

We focus our study on light-weight transactions that are supported by hardware and target reliability rather than regular transactions with the purpose of concurrency control (see Chapter. 4 and Chapter. 5). Thus, they do not require code transformation and they can be committed when it is required. For instance, in regular HTMs, commit points of transactions are statically defined in the source code. On the contrary, reliability purposed transactions can be committed flexibly, for example they can commit when the HTM structures are full 5.

6.3 Error Detection Schemes

In this section, we review several lightweight error detection mechanisms and discuss their applicability for energy efficient computing. Typical error detection mechanisms in the literature (1) run the code redundantly and compare the outputs, i.e., rely on replication, (2) use assertions/invariants, (3) use encoded processing, or (4) monitor the error symptoms. We further discuss how these error detection mechanisms can be combined with TM (irrespective of whether TM is built in software or hardware). A qualitative comparison is provided at the end of the section.

6.3.1 Replication

To satisfy the strict reliability requirements of mission-critical systems, various redundancy-based error detection solutions have been proposed [32, 33]. These solutions are also termed as N-Modular Redundancy (NMR) Techniques in which N represents the number of replication. Triple Modular Redundancy (TMR) is one of the most popular NMR technique which executes the instruction stream three times. TMR compares the results of the replicas via a voting circuit and upon results divergence decides that one of the copy is correct. Since it expects a single result, it does not require a recovery and it presents an forward error correction. Dual modular redundancy (DMR) schemes, another NMR technique, execute an instruction stream redundantly in two synchronized processors and check if both produce identical results. If the results diverge, a recovery mechanism can be triggered. The comparison of execution results causes synchronization and comparison overheads in the execution time.

In Chapter 5, we present FaulTM which utilizes Transactional Memory for dual replication so that it reduces the comparison overhead. This is because, FaulTM efficiently compares the write-sets instead of comparing each individual store operation. The write-set has typically less entries than the total number of store instructions, because multiple stores to the same address are mapped to a single entry. Also the comparison is done only at the commit stage of the transactions, which presents less time overhead for synchronizing the replicated executions. The comparison overhead can be further reduced by comparing hash-based signatures of write-sets and register files of the transactions. It is trivial to apply FaulTM for triplication in which instruction streams are triplicated with transactions and those three transactions synchronize and compare their results from time to time.

Although redundancy provides a very high error detection capability, it suffers from 100% (or 200% for triplication) energy and space overhead in the error-free execution. However, the energy consumption is reduced exponentially when V_{dd} is lowered, in comparison to the linearly increasing energy consumption of replication. Hence, it is worth to further investigate replication as error detection mechanism.

6.3.2 Assertions/Invariants

Using assertions is a common technique for detecting software or hardware errors [102]. Assertions are conditions referring to the current and previous state of the program. If the states do not match the expected results, an error is detected. Upon such event, the typical behaviour is to issue a warning, but corrective actions can also be triggered [103].

An approach based on a coprocessor (watchdog) is proposed in [104]. It uses annotations in the first phase of the error detection, where processes provide some information. In the second phase the processes are continuously monitored and the collected information is compared with the information previously provided. The authors claim an error coverage of 90% of transient and permanent errors by control-flow and memory access checking.

As pointed out in [90], combining assertions with transactions is an interesting approach as one can implicitly create the latter based on the invariants provided by developers. Inserting invariants manually into the program has the drawback that the resulting assertions might be unsound (lead to false positives) or incomplete [105] and might be inefficient because too many evaluations are needed. The alternative is to add them automatically to a program, as proposed in [106]. To detect faulty assertions Knauth et al. [107] use random mutation to detect the most likely software bugs. This mechanism could be extended to improve the combination of assertions and TMs. Another possibility is to automatically add invariants to a program, as described in [106].

The authors of [108] propose an extension of STM Haskell with invariants that concentrates on C like consistency from the ACID characteristics. Consistency is ensured by dynamically-checked invariants that must hold if the system is in a consistent state. The authors identified that the frequency of invariant evaluation represents a tradeoff between overhead and detection rate. In their work, they reduce the overhead by the following measures. The invariants are (1) garbage-collected if their watched data structure does not exist any more, and (2) invariants are only checked if a transaction wrote a variable read by the invariant.

6.3.3 Symptom-Based Error Detection

In order to provide reliability at a low cost, some recent error detection solutions [40, 57] monitor program executions to inspect if there is a symptom of hardware faults. These symptoms can be mispredictions in high confidence branches, high OS activity, or fatal traps (e.g. attempting to execute an undefined instruction code).

In Chapter 4, we present SymptomTM, a symptom-based error detection mechanisms using transactions to recover from application crashes. Also a similar scheme has been disclosed in a patent filed by IBM [109]. In this approach, applications are executed in back-to-back, reliability purposed transactions which are monitored to detect if there are any symptoms of hardware errors, which typically result in fatal traps. Unless any fatal trap exception is raised in the transaction, the write-set is committed to shared memory at the end of the transaction. Otherwise, the system aborts and re-executes the transaction. SymptomTM avoids catastrophic failures induced by transient or permanent faults without any perceptible performance degradation. Since there is no replication, the scheme has virtually no area/energy overheads. It has, however, limited error coverage since it cannot detect silent data corruptions (SDC) and, further, exceptions can be raised after the commit of the transaction.

Since some symptoms can be observed very efficiently (e.g., catching exceptions), symptom-based error detection can be easily combined with other error detection mechanisms. Other symptoms (e.g., infinite loops due to a corruption of the stop condition) require an instrumentation of the code or support by the operating system (e.g, adding time-outs to operations).

6.3.4 Encoded Processing

Error correcting codes (ECCs) are commonly used to detect and correct soft errors in memory by adding redundancy. ECCs usually provide single bit error correction and double bit error

Method	Memory	Processing	Error Detection	Complexity
Replication	high	high	**high**	**low**
Assertions	medium	high	medium	high
Encoded Processing	medium	high	**high**	high
Symptoms	**low**	**low**	low	**low**

Table 6.1: Comparison of error detection schemes (bold is better)

detection [110]. However, soft errors might also be introduced during data transport and processing in the logic building blocks. One way of applying the principles of ECC to runtime errors is encoded processing [111]. The redundancy is added by applying arithmetic codes to the values processed by the application. This can be done either using custom hardware or in software by an encoding compiler [112]. All operations must preserve the encoding, which results in more computations and higher energy consumption.

The level of error detection that can be achieved using encoded processing depends on the selected type of arithmetic code, e.g., AN codes can detect value errors while ANBDmem codes [113] can additionally detect lost updates in memory, but at the expense of a higher processing overhead [4]. The observed rate of undetected errors is 9% and 0.5%, respectively.

Wamhoff at el. [83] proposed Transactional Encoding which combines arithmetic codes for error detection and Software Transactional Memory for error recovery. In this design, value is validated when it is read or written by checking its arithmetic code. If the code is incorrect, the transaction must be aborted. Wamhoff at el. provide the design for both lazy and eager conflict detections. For higher efficiency, they propose deferring the validation of a code word until a transaction commits or the value becomes externally visible (lazy checking). This avoids the costly check on each access of the value because the error propagates in the employed arithmetic code. Eager checks can allow the application to identify the first occurrence of an invalid value and to react more pro-actively.

6.3.5 Qualitative Comparison

Error detection is a critical step for enabling low voltage operation but it does not come without cost. The energy efficiency is highly dependent on the selection of the right technique. In the following, we summarize the aforementioned schemes and provide a comparison regarding factors that influence the design decision. We concentrate on (1) the overhead introduced in memory and processing, (2) the error detection coverage, (3) the requirements for setting up the error detection. Table 6.1 compares the overhead of the single error-detection schemes in processing and in memory, when applied to an error-free system.

Memory and Processing Overhead. Replication has high memory and processing overheads because the whole application executes in parallel. With assertions/invariants, the overhead depends on the programmer or the automated tool that generates them. It can be medium to high in memory, depending notably on the support for garbage collection. The annotations have to be evaluated in any case (even if there are no failures). Encoded processing needs only a small amount of additional memory to keep the arithmetic codes, but all executed operations incur the significant overhead of maintaining the encoding. For the symptom-based error detection only the symptoms have to be stored and checked, therefore the overheads are low.

Error Detection Coverage. There is usually a tradeoff between error detection coverage and overhead. For example, whereas replication provides 100% error detection, it requires many resources and hence might not be usable for energy efficient computing when the processor runs in high performance mode. The assertion-based mechanism is highly dependent on the implementation. Transient errors might not be detected, because they are simply not covered. However, there are implementations that claim to reach 97% coverage with only 5-14% performance overhead [114]. The detection capabilities of encoded processing depend on the applied arithmetic code. Its complexity introduces linearly increasing runtime costs, while the error detection rate increases exponentially [4]. Symptom-based error detection provides a limited error coverage with a very-low performance overhead.

Requirements for Using the Mechanism. Although the energy efficiency of a system is the main goal, a mechanism can only be successful if it can be easily applied, especially if the error-detection mechanisms have to be combined with recovery. Replication has few requirements for the checking of the output and the application does not have to be changed. Similarly, symptom-based error detection requires only the detection of exception which already exists in the hardware. Assertions usually need language support to be defined and the implementation must verify them during the execution. Encoded processing can be implemented as a combination of compiler and library, and integration with the application is straightforward.

6.4 Analysis

In this section we analyse the feasibility of applying the aforementioned error detection schemes with TM-based error recovery. We are specifically interested in how much we can lower the

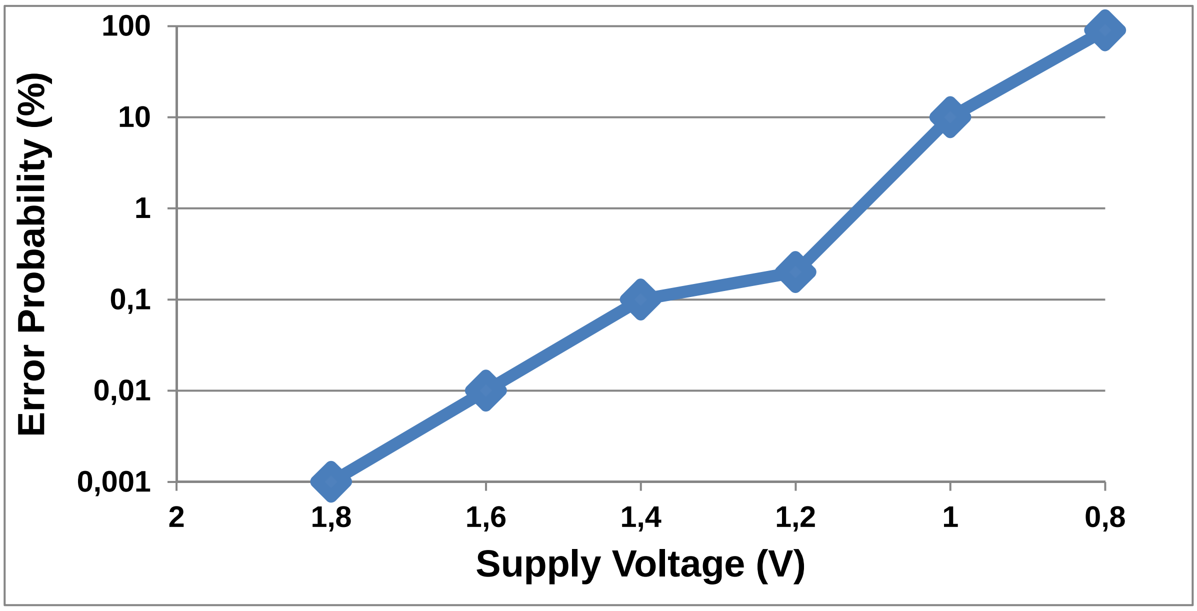


Figure 6.1: Error rate as analysed by Ernst et al. in [2]

voltage while still providing high error detection capability.

For the evaluation we consider the following scenarios: 1) We show the fault rate of execution units under scaling voltage; 2) We compare the error detection capability of each of the schemes; 3) We investigate the energy overhead of the error detection schemes and the combined error detection and recovery; 4) Finally, we consider combination of different error detection schemes.

6.4.1 Fault rate

To the best of our knowledge, only Ernst et al. performed a study in [2], evaluating the error rate of the execution units for the voltage levels below safe V_{dd}. Although there are newer technologies such that Intel Atom Processors which can execute between 1.1V to 0.75V safely [115], to the best of our knowledge, there hasn't been a study for the error rate in Atom Processors below 0.75V. However, the study presented in this chapter still applicable for these new processors in which only the scope shifts beyond 0.75V. The results presented by Ernst et al. [2] are shown in Figure 6.1. For this experiment a circuit-level design of a 64-bit Kogge-Stone adder has been implemented, assuming an 870 MHz clock and an ambient temperature of 27 C. As an input,

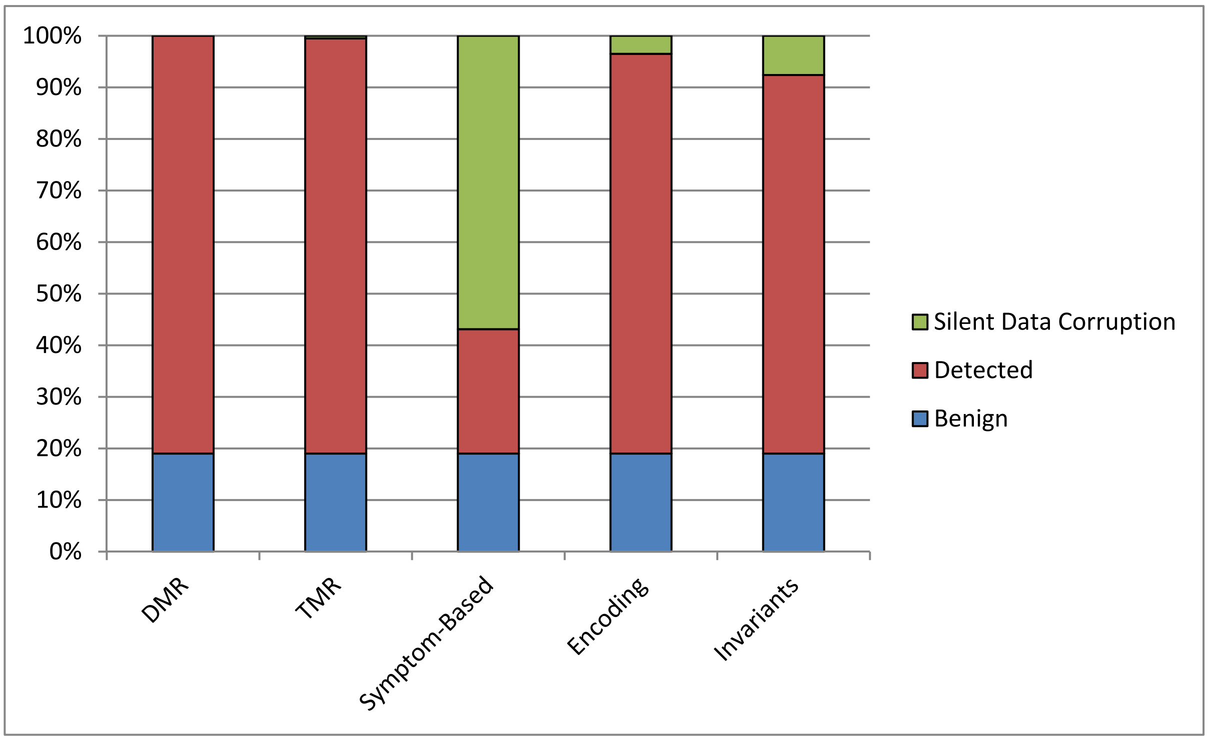


Figure 6.2: Fault coverage of each scheme according to the results presented in Chapter 5 and [3, 4].

random sample vectors are generated. The error rate is computed as the fraction of sample vectors that do not complete within the clock period of the current voltage and frequency. In this study, we conservatively use the error rate of the random input sequence by knowing the fact that the resulting error rate is higher than in a real application.

6.4.2 Error detection capability

For showing the error detection capability of each scheme, we summarize results found in the literature (Chapter 4 for symptom-based error detection, Chapter 5 for dual-modular redundancy (DMR), [3] for invariants and [4] for encoding) and display them in Figure 6.2. On average, 20 % of the injected faults are benign since they are masked before the transaction ends. Note that this is lower than the results reported in [3], because we conservatively classify faults as not being benign, if the injected fault still exists at the end of the transaction.

DMR provides 100 % fault coverage since it detects all injected faults unless they are be-

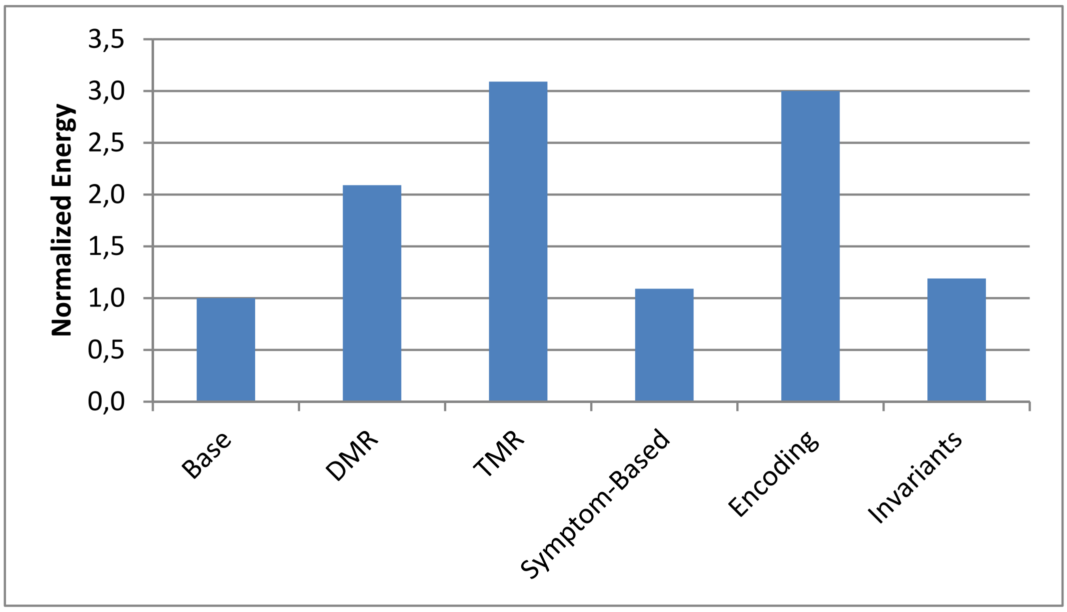


Figure 6.3: Normalized energy spent by each error detection scheme when the V_{dd} is 2 V according to the results presented in Chapter 5 and [3, 4].

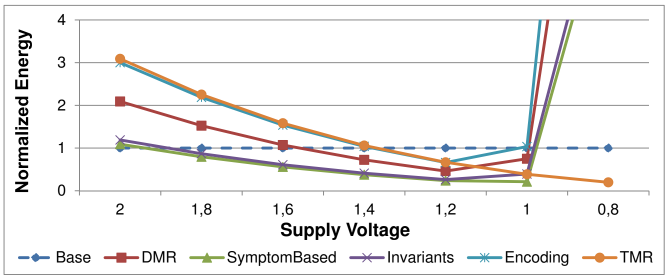
nign. The coverage of TMR is slightly lower than DMR, because if there are two faulty copies, TMR can only detect a failure, but not correct it. Note that hardware based replication (DMR or TMR) provides higher reliability since it does not have a vulnerability window as software-based replication. Symptom-based error detection only covers 35 % of the faults, if transactions are short (i.e., less than 10,000 instructions). Similarly, encoded processing provides 97 % fault coverage, while invariants cover 93 % of the faults. Note that we include the benign faults into fault coverage and leave only the harmful data corruptions out of the coverage.

6.4.3 Energy consumption

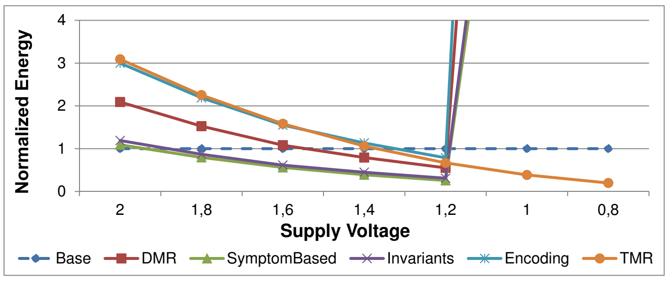
In the following section, we show the energy consumption of the error detection schemes under full voltage and under scaling voltage.

Energy overhead of the error detection schemes.

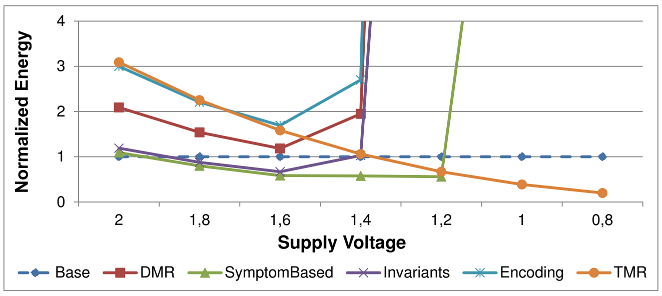
We calculate the energy overhead under full voltage as the number of additional instructions (memory instructions, integer/floating point instructions) by relying on the results found in the literature (Chapter 4, Chapter 5, [3], [4]). The results are shown in Figure 6.3, where we



(a) Energy (TX=10 inst)



(b) Energy (TX=100 inst)



(c) Energy (TX=1000 inst)

Figure 6.4: Energy consumption under scaling voltage

normalized the values to the base, error-free energy consumption. Basically, high reliability requires more energy. For example, DMR and TMR have an overhead of more than factor two and three, respectively. The symptom-based error detection only has a negligible overhead, but also only provides a low error detection capability.

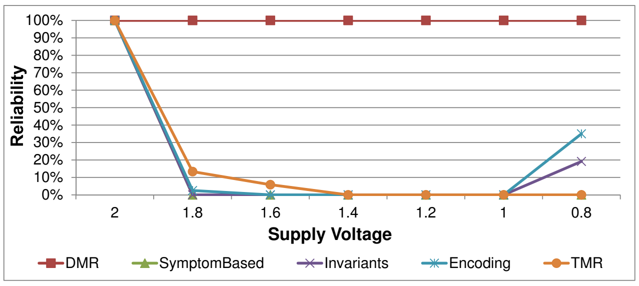
Energy consumption under scaling voltage.

For the voltage scaling experiments we use the Gem5 full-system simulator [52] and run the SPLASH2 [77] benchmark suite. The simulator runs with in-order cores executing X86 ISA running at 1 GHz with private 64KB L1D and L1I caches and a unified 2MB L2 cache. During the simulation, we inject faults to the output of each instruction with the theoretical error probability for the applied V_{dd} (given in Figure 6.1). Then, we assume that the faults are detected with the error detection probability as shown Figure 6.2 (or the fault is benign). If an error is detected, the corresponding transaction aborts and rolls back. If all transactions in the application execute without any error (but with the possibility of re-executing the transaction several times), the application completes reliably. We repeat this fault injection experiment 40 times for each application for each V_{dd} level and we calculate the reliability of the application as the rate of the executions in which the application completes reliably. To estimate the energy spent under the given V_{dd}, we calculate the total number of instructions executed (for error detection given in Figure 6.3 and for re-execution in the error recovery) and multiply it with the energy spent for one instruction under the given V_{dd} level. We repeat this fault injection experiment for the transaction sizes of 10, 100 and 1000 instructions. Regarding the TM, we do not make any assumption about the type of transactions (i.e., lazy/eager conflict detection or hardware/software transactions).

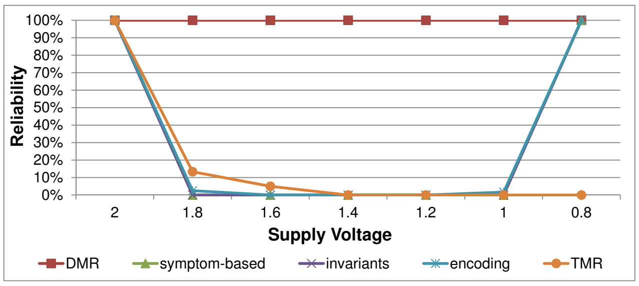
In Figure 6.4, we summarize the normalized energy consumption of all applications in the SPLASH benchmark by averaging their energy consumption. The energy consumption is normalized to the error-free base case in which 2 V supply voltage is used. In Figure 6.5, we display the averaged applications' reliability of the combined error detection and recovery under the given V_{dd}.

From these graphs we can make several observations: DMR provides the highest reliability, because it is very unlikely to have two failures affecting the replicated copies of the same instruction at the same time such that both replicated executions result in the same faulty value. However, DMR presents a high energy overhead for high V_{dd}s. When a transaction consists of 10 or 100 instructions, DMR starts to outperform the base-case, when V_{dd} is 1.4V (i.e. up to 28% reduction) or 1.2V (up to 54% reduction). For larger transactions (i.e., 1,000 instructions), the overhead of DMR cannot be covered by lowering the V_{dd} any more. It is because, due to the increase of the transaction size the number of faults causing rollbacks repeatedly becomes considerably higher.

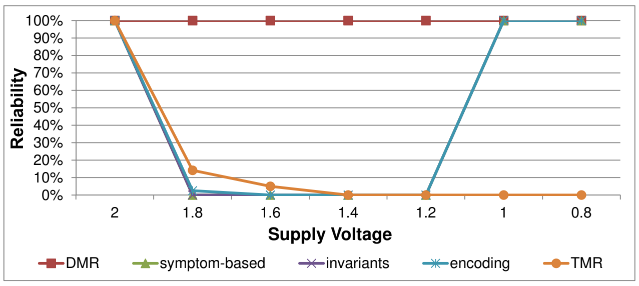
Error detection schemes other than DMR only provide a lower reliability. One reason is the high number of transactions in the SPLASH applications, which can be up to 2 billion (barnes with transaction size of 10 instructions). In this case, although the probability of a transaction fail (i.e. in the transaction, at least one instruction fails but the transaction do not recovers) is very low, the probability that having at least one transaction which produced undetected faulty result increases drastically for long applications even for high reliability providing schemes



(a) Reliability (TX=10 inst)



(b) Reliability (TX=100 inst)



(c) Reliability (TX=1000 inst)

Figure 6.5: Reliability under scaling voltage

such that TMR.

For recovery requiring schemes (i.e. all schemes except TMR), when the failure rate is very high, the reliability provided by these schemes starts to increase again. It is because, transactions starts to encounter many failing instruction and it is enough to detect only one failure in the transaction in order to trigger the recovery. For instance, assume that there are two faulty instruction in a transaction and the reliability scheme detects only one of them. In this case, the transaction rolls back anyway although it did not detected all failures.

Symptom-based error detection and invariants have a low overhead and start to outperform the base case very fast. They require up to 88% less energy than the base case for a transaction size of 10 and V_{dd}=1V. However, these schemes are not enough to complete the whole application reliably at this voltage level. When the failure rate gets very high, due to very low V_{dd} values, the error detection capabilities increase again. At these low levels it is very likely that more than one instruction per transaction fails. In this case it is easier to classify a faulty transaction, because it is only necessary to find one failed instruction to roll back. However, symptom-based error detection starts to be energy-inefficient in the moment the reliability increases.

TMR and encoded processing (schemes presenting high overhead in the base case less than 100% error detection capability), can only lead to a lower energy consumption than the base case when V_{dd} is lower than 1.4V. Since TMR does not present any recovery overhead, the supply voltage can be reduced to 0.8V (up to 80% less energy consumption than the base case). However, at these voltage levels there will be at least one transaction that produces a non-benign, faulty result.

6.4.4 Combining Error Detection Schemes

There is a tradeoff between energy efficiency and reliability, as we have seen for DMR and symptom-based error detection and TM recovery. However, there are many applications that are implicitly fault tolerant (e.g., from the area of multimedia and artificial intelligence). To reduce the overhead of the error detection schemes, the programmer can define parts that are less strict regarding outcome precision. Thus, we can for example combine symptom-based error detection and DMR for consuming less energy, but providing full reliability for critical parts. In Figure 6.6, we depict the energy overhead in comparison to the base case and DMR only for a transaction size of 100 instructions. We assume that 30, 50 or 70 % of the application are only secured by symptom-based error detection. With this combination it is possible to lower the V_{dd} to 1 V (in comparison to 1.2 V with DMR only) and still be more efficient than the base case. Specifically, we reduce the energy consumption by 66 % in comparison to the base case. However, at these voltage levels the reliability of symptom-based error detection is at 0 %, thus might be omitted at lower voltage levels. At a voltage level of 1.6 V the reliability is around 10 %, but the energy consumption 20-50% lower than the single usage of DMR.

Another possibility would be to use approximate computing as e.g., studied by Sampson et al. in [88]. The programmer can define approximate parts and these are considered as fault tolerant. The rest can be protected by DMR and transactional memory. In this study, we prefer

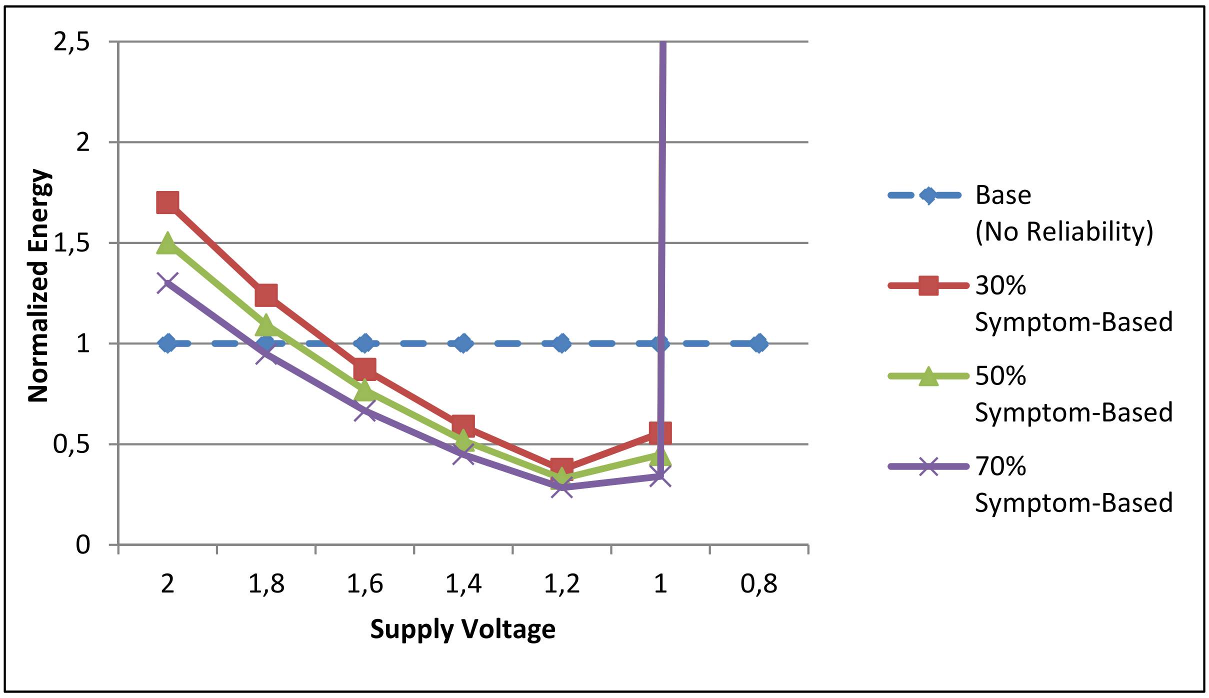


Figure 6.6: Possible normalized energy consumption of the combination of DMR and symptom-based error detection (TX-Size=100 instructions).

utilizing symptom-based error detection instead of approximate computing for the non-critical sections of applications, because with symptom-based error detection and TM recovery we can still avoid system crashes. So that, if there is a failure causing system crash in these sections, symptom-based detection can detect and recover the failure and can avoid the system crash.

In summary, the combination of error detection and TM-based error recovery can be used when lowering voltage. The energy reduction depends on the size of the transactions. The decision on which schemes should be selected is dependent on the required level of reliability (i.e. if applications are mission-critical or not) and the targeted supply voltage. A possible solution would be to provide all mechanisms and decide dynamically based on hints by the application (e.g., configuration) or runtime on which mechanism to use.

6.5 Summary

To improve the energy-efficiency of modern CPUs, one can reduce the supply voltage of cores. Reducing the supply voltage increases, however, the likelihood for wrong executions of programs. In this chapter, we proposed using transactional memory (TM) for rolling back the effects of wrong executions. To reduce the energy consumption, one needs an error detection scheme that has both a sufficient coverage and a low overhead. We discussed multiple error detection alternatives. Based on our evaluation, we conclude that one can reduce the energy consumption of CPUs, in particular, if we have efficient hardware support for TM and for error detection.

7 Energy Reduction in Memory Structures

Technology minimization and associated gains in performance and productivity are in jeopardy as the power density and energy consumption of modern computing devices increase. For instance, smaller transistors allow computer designers to pack more chips in each technology node in the same area. However, within the given power budget, not all portions of the chip can be powered. The area of the chip which are not powered is termed as *dark*, and this is a recent problem known as Dark Silicon Phenomena [86].

Downscaling the supply voltage (V_{dd}) close to the transistors' threshold (near-threshold execution) or lower than the threshold (sub-threshold execution) is a quite effective approach for minimizing the energy consumption of the computer systems [116]. Therefore, as energy becomes a key design concern for computer systems, processors started provide 1) high-performance and 2) low-power operating modes. Processors run at a high frequency by using the nominal supply voltage (V_{dd}) in the high-performance mode, and they reduce V_{dd} in the low-power mode to reduce the energy consumption by trading-off performance [117, 118]. However, this energy reduction comes with a drastic increase in the number of failures especially in memory structures (i.e on-chip SRAM memories such as L1 and L2 caches) [119, 120].

These memory failures can be persistent (i.e. yield loss or hard errors) or non-persistent (i.e. soft errors or erratic bits) while rates of both failures increase as the V_{dd} is decreased. Moreover transistor scaling increases the vulnerability of transistors to radiation events since it increases the likelihood of having multibit soft errors on adjacent bits [121]. Thus, it is essential to implement reliability solutions addressing both persistent and non-persistent failures in caches in order to reduce the V_{dd} and provide reliable cache operation for future technology nodes. There are two main techniques to deal with high fault rates stemming from the above issues. The first mechanism utilizes coding techniques such as parity or ECC. The second mechanism features in-cache replication. While they are effective, both mechanisms have issues.

Utilizing Error Correcting Codes (ECC) in the low-power operating mode is an appealing and prevalent solution for reducing the safe operating margin for V_{dd} of memory structures [122, 123, 124]. ECC is the most widely used techniques for detecting and correcting both persistent and non-persistent failures with additional area, power and encoding/decoding time overhead [122, 123, 124]. ECC extends data lines with additional parity bits. The encoder of the ECC generates parity bits when the data line is updated. In the reading of the data line, the decoder regenerates the parity bits to check and correct any existing fault. However, the increase in the error correction capability of ECC is much lower than the increase in power and area consumption. For example, in 8-byte data, correcting a double-bit error costs 19% area overhead while three-bit error correction requires a stronger and a more complex ECC with 100% area overhead [121]. Intel's latest 22nm 15-core Xeon processor uses Double Error Correction, Triple Error Detection (DECTED), a very strong ECC, for its L3 cache data tag array; however, the computational cost of DECTED ECC impacts the L3 data accesses, whose latency is variable, thus significantly complicating the micro-architecture [125]. Due to the diminishing benefits of stronger ECCs, providing reliability in an environment with a very high fault rate (i.e. more than 10^{-3} failure probability for each bit) such as when the processor is operating in a very low power mode, is not trivial. Thus, only a few ECC solutions address large-scale multibit errors in a line [122, 124]. However, they require a complex encoder/decoder with a high energy consumption which diminishes the energy saving potential of the low-power mode execution.

The second mechanism, in-cache replication such as triplication, is a conventional way of providing high reliability with a minimum correction latency in which replicated cache lines are corrected via bitwise majority voter [126, 127]. However, replication schemes have two main problems: (1) Writing/reading more than one cache line increases access latency and energy consumption of caches. (2) When processors operate with a very low V_{dd}, the number of uncorrectable lines increases due to the multiple failures in the same bit-position. (i.e. for a 512-bit cache line with near-threshold V_{dd}, 14% of the triplicated lines are uncorrectable (We used 420mV for V_{dd} and simple probability theory analysis to calculate the fault rate.).

In this chapter, our goal is designing on-chip SRAM memories which can tolerate very high bit failure rates of ultra-low voltage execution with minimum overhead, and without harming the access time and read/write energy of the cache in the nominal mode. To this end we propose two schemes:

1. ECC-Based Solution: We Adopt a Single Error Correcting - Multiple Adjacent Error Correcting ECC (SEC-MAEC code) - a fast and energy efficient ECC with a high error correction capability - in the faulty lines of L1 caches for below safe voltage operations. The SEC-MAEC code used [9] is designed for multi-bit soft errors occurring in the adjacent bits in order to correct them in one cycle. It can accomplish the error correction via only 4 level gate pass

2. Redundancy-Based Solution: We present a simple, circuit-driven solution, Flexicache, that duplicates or triplicates all the available lines in the cache and achieves each read or write access to multiple lines without increasing access latency. Flexicache, automatically configures itself for different supply voltages in order to tolerate different fault rates. It works in one of the three modes:(1) Single Version Mode (SVM), (2) Double Version Mode (DVM) or (3) Triple Version Mode (TVM). Flexicache also divides each cache line into single-parity-protected partitions in order to increase the error correction capability of replication schemes.

We compare the selected SEC-MAEC code with the Orthogonal Latin Square Code (OLSC), a state-of-the-art fast ECC scheme utilized in L1 caches in order to lower the supply voltage [123]. We present that SEC-MAEC reduces the area overhead of decoder by 10X while reducing the encoding and decoding latency into half. We also show that the energy spent for encoding and decoding can be reduced up to 80%.

We compare Flexicache with conventional triplication schemes [126, 127] besides OLSC. Flexicache can provide a higher error correction capability (i.e. providing a cache with a higher capacity in low power mode) with significantly lower error correction latency. Flexicache can continue to operate reliably up to 10% bit failure rate which provides operational capability in

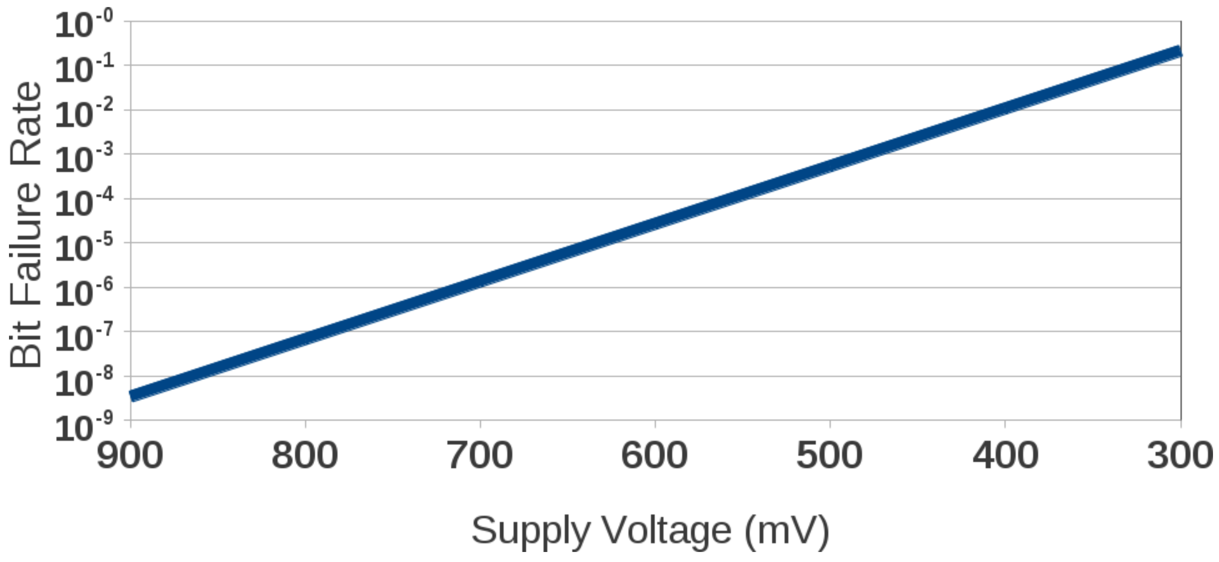


Figure 7.1: Bit Failure Rate vs Supply Voltage

320 mV by reducing the energy consumption of cache operations by 68%. Also, Flexicache can operate until 9% bit failure rate without harming the uniform view of the cache. The area overhead of Flexicache is only 8.6% compared to the typical single parity protected L1 cache.

In Section 7.1, we explain bit failures occurring in SRAM memory structures and we present the previous proposal for correcting those memory failures. In Section 7.2, we explain the principles of the SEC-MAEC code that we utilized in this study and we present how we extend the faulty cache lines with SEC-MAEC codes. In Section 7.3, we elaborate the basic principals of Flexicache and its circuit design. In Section 7.4, we present the evaluation of SEC-MAEC and Flexicache in terms of area, power, time and reliability by comparing OLSC, the state-of-the-art ECC scheme. We also compare Flexicache with typical TMR scheme.

7.1 Background

In this section, we first explain the bit failures occurring in memory structures due to scaling voltage. Then, we present the related work which allows running memory structures under ultra-low voltages. We also give details of Orthogonal Latin Square Codes which is the state of the art ECC scheme for level-1 memory structures when V_{dd} is below the safe margin.

7.1.1 Failures in Memory Structures due to Voltage Scaling

Bit failures in the memory structures below the safe operating margin are classified into two broad categories [122]: 1) Persistent Faults and 2) Non-Persistent Faults.

Persistent Failures:

The random variation in the number and location of dopant atoms in the channel region of the device leads to the random variations in transistor threshold voltage. It causes threshold voltage mismatch between the transistors close to each other. In a SRAM cell, a mismatch in the strength between the neighbouring transistors caused by intra-die variations can result in the failure of the cell [128]. A cell failure can occur due to the following reasons:

1. **An increase in the cell access time:** When the differential voltage developed across the bit-lines during a read operation is not sufficient for sense amplifier to identify the correct value, an access failure occurs.

2. **Unstable read operation:** When the stored value flips during a read operation, a read failure occurs. This happens when the noise developed on the node storing 0 is larger than the trip point of inverter.

3. **Unstable write operation:** When the cell contents cannot be toggled during write operation, a write failure occurs.

4. **Retention failure:** When the stored value in the cell is lost during standby, a retention failure occurs.

The mapping between the bit failure rate and the V_{dd} is examined by Miller et al [124] for 32nm technology which can be seen in Figure 7.1. It is shown that as V_{dd} is lowered, the bit failure rate increases exponentially. We reference these previous results for our evaluations.

On the other side, open or short circuits cause irreversible physical changes in the semiconductor devices (see Chapter 2). These permanent failures tend to occur early in the processor lifetime due to manufacturing faults (called the infant mortality), or late in the lifetime due to thermal and process related stress.

The location of a persistent failure is random and independent of whether the neighbouring bit is faulty or not [116]. The locations of persistently defective bits can be detected by performing built-in self test (BIST) [18] in the postmanufacturing period for each supply voltage values.

	Segmented ECC	2D ECC	Disabling/ Bit-Fix	Flexicache	SEC-MAEC
Persistent Failures	**yes**	**yes**	**yes**	**yes**	**yes**
Non-Persistent Failures	**yes**	**yes**	no	**yes**	**yes**
Minimum V_{dd}	375 mV	–	400 mV	**320 mV**	375 mV
Latency in the Low-Power Mode	1 cycle	1 cycle	**0 cycle**	1 cycle	1 cycle
Other Latency	**no**	read-modify -write	**no**	**no**	**no**

Table 7.1: Comparison of SEC-MAEC and Flexicache with Architecture Based Error Correction Schemes for Scaling Vcc (Bold is better)

Non-Persistent Failures

Radiation events or power supply noise can cause a bit flip and corrupt a data stored in a device until a new data is written [129]. As transistor dimensions and operating voltages shrink, sensitivity to radiation events increases drastically. On the other side, process variation or in-progress wear-out, combined with voltage and temperature fluctuations might cause correlated faults of short duration. They are termed intermittent faults (or erratic failures), that last from several cycles to several seconds [19]. Diagnosing an intermittent fault by BIST is hard since it does not persist and conditions that cause the fault are hard to regenerate (for further explanation see Chapter 2).

As V_{dd} decreases, the bit failure rate increases rapidly for both intermittent faults and persistent failures [122, 124]. The impact of voltage scaling on soft errors is not drastic [122]. However, scaling down of transistor size increases the likelihood of having multibit upsets in adjacent bits [121].

7.1.2 Related Work:

Multiple architecture-based error correction schemes have been proposed in the past. We discuss below the details of the most relevant ones. Table 7.1 summarizes their main characteristics and compares with Flexicache and SEC-MAEC.

Error Correction Codes (ECCs) [130] extend each cache line with the collection of several parity-bits to detect and correct persistent and non-persistent faults. However, ECC requires very high overhead, requiring storage for correction codes as well as complex and slow encoders/decoders.

Orthogonal Latin Square Code (OLSC) [123] is a state of the art ECC scheme used for level-1 caches when the supply voltage is lower than the safe margin. We explain the details of OLSC in Section 7.1.3. Multi-Bit Segmented ECC (MS-ECC) [122] utilizes OLSC at a finer granularity in order to increase the error correction capability of OLSC to be used for ultra-low voltage level. Thus MS-ECC can reduce the supply voltage until 350 mV in 35nm technology by providing 6.5% useful cache capacity (We define useful cache capacity as the portion of the cache which is not disabled) [124].

Kim, et al. [131], propose two-dimensional (2D) ECC to correct multi-bit errors with a minimum area overhead in check bits. 2D-ECC calculates and saves the ECC values of the rows and columns by using a simple ECC scheme such as SECDED (i.e Single Error Correction Double Error Detection) in order to provide a strong error correction capability for L1 caches. However, the correction capability of this scheme is strongly dependent upon the location of defective bits. So that, it is not convenient to use in low-power mode when failures are random. Also, it requires a read-modify-write operation for all Stores and for every cache miss which increases the delay and power consumed by all write operations to the level-1 cache.

Miller et al. [124] proposed Parichute which utilizes Turbocodes for reducing V_{dd} of the second and higher level caches. Although this scheme provides a very high error correction rate and supports a significant voltage reduction, its error correction latency can be couple of cycles (i.e. more than 5 cycles [124]) in the near-threshold voltage level. Thus, Parichute is not convenient to be used in time-critical L1 caches.

Yoon et al. proposed Memory-Maped ECC in which error correction data is saved in the memory hierarchy to be accessed in case of an error detection [132]. This scheme is not applicable in the near-threshold voltage execution in which all cache lines present bit failures and require error correction.

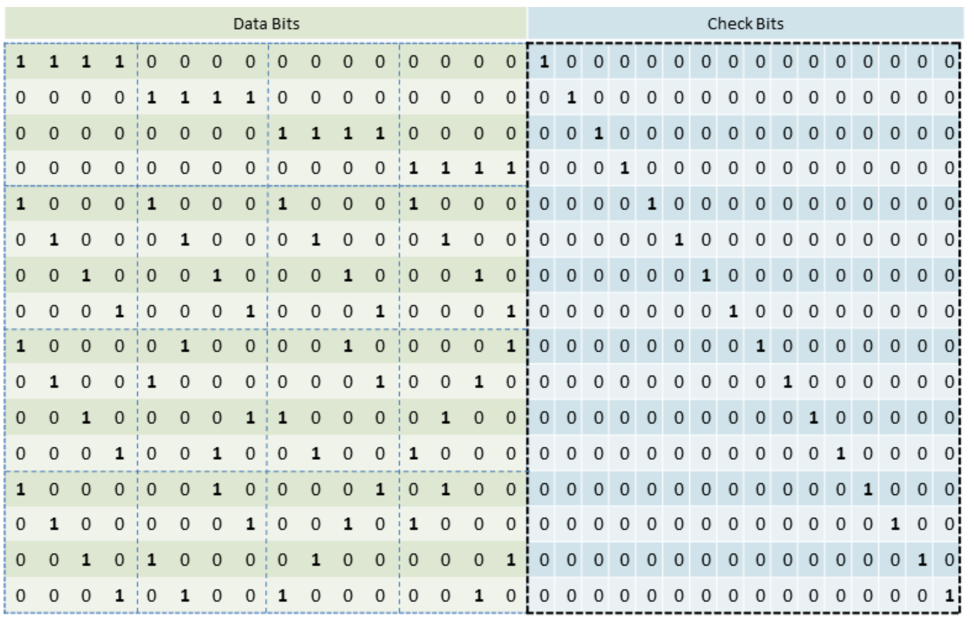
Several disabling schemes have been proposed for tolerating only persistent failures [133, 134, 135]. Wilkerson et al. [133] disables the faulty words in order to combine two consecutive cache lines to form a single cache line where only non-failing words are used. Although the area overhead of word-disable in high-power mode is only 8%, in the low power mode the available cache size shrinks to the half when the error rate is lower than 0.01%. Abella, et al. [134] disables sub-blocks instead of words in order to utilize more capacity in the low-power mode. Both disabling schemes need to access a fault map in parallel. Wilkerson et al. also proposed bit-fix in the same study as an alternative to disabling [133] in which the location of defective bits are stored in one of the cache ways. However, this scheme can correct up to 10 failing bits in a cache line among 4 ways (i.e. bit failure probability is around 0.005). ZerehCache [135] employs fine granularity re-mapping of faulty bits and relies on solving a graph coloring problem to find the best re-mapping of defects to spare lines. Bit-fix [133] stores the location of defective bits and their correct values to the quarter of cache ways. Koh et al. proposes buddy cache which pairs a faulty cache block with another faulty cache block in the same set [136]. Trivially, for a 4-way set associative cache, buddy cache in not applicable when the fault rate is higher than 10^{-3} since the probability of having a fault in the same bit position is high.

Besides architectural approaches, circuit-based hardening approaches have also been proposed. Commonly, 6T SRAM cells are used for cache structures. Alternative type of SRAMs such as 8T [137] or Schmitt-Trigger 10T [138] are also proposed to target low voltages. These cells are more stable against parameter variations than 6T cells and also they keep strong guaranties for reliability. Some existing processors such as Atom [115] uses 8T cells. However, theses cells present substantially high area (i.e. more than 30%) and energy overhead in the nominal voltage compared to 6T cells. Maric et al. also proposed combining different cell types together with ECC in order to balance the area overhead and energy reduction [139]. In this chapter, our goal is to reduce the energy consumption in te low voltage without increasing the area and energy overhead in the nominal voltage significantly. Thus, using different cell types are not convenient for our goal.

7.1.3 Orthogonal Latin Square Codes

OLSC are based on the concept of orthogonal latin squares [123] and can be decoded using majority voting. An OLSC encodes "orthogonal" groups of bits to form check bits. At decoding time, each data bit generates the final value through a voting process from a group of orthogonally coded data and check bits. Thus, an OLSC does not need to generate a syndrome but can "correct" errors directly from majority voting. To perform t error corrections in a data block consisting of m×m bits, an OLSC requires 2×t×m check bits. In Figure 7.2, we present the H-matrix of an OLS code for an 16-bit data correcting up to 2 bit errors (i.e. m=4, t=2).

The encoder computes each check bit C_i as the XOR over data bits corresponding to



Data Bits																Check Bits															
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	1	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 7.2: Parity check matrix for the Orthogonal Latin Square Code with k = 16 that can correct two errors

columns of the H-matrix that have a '1' , such that;

$$
\begin{aligned}
c_0 &= d_0 \oplus d_1 \oplus d_2 \oplus d_3 \\
c_1 &= d_4 \oplus d_5 \oplus d_6 \oplus d_7 \\
&\ldots \\
c_{15} &= d_3 \oplus d_5 \oplus d_8 \oplus d_{14}
\end{aligned}
$$

Each row in the H-matrix has exactly m bits of '1's. Thus the calculation for each check bit requires an m-input XOR operation with the critical path being ceil($log_2(m)$) levels of 2-input XOR gates.

The decoder decides the correct value of d_i via (2t+1)-input majority voter. One input of the voter is the received d_i' itself, the other 2t are derived from check bits that contain d_i as its

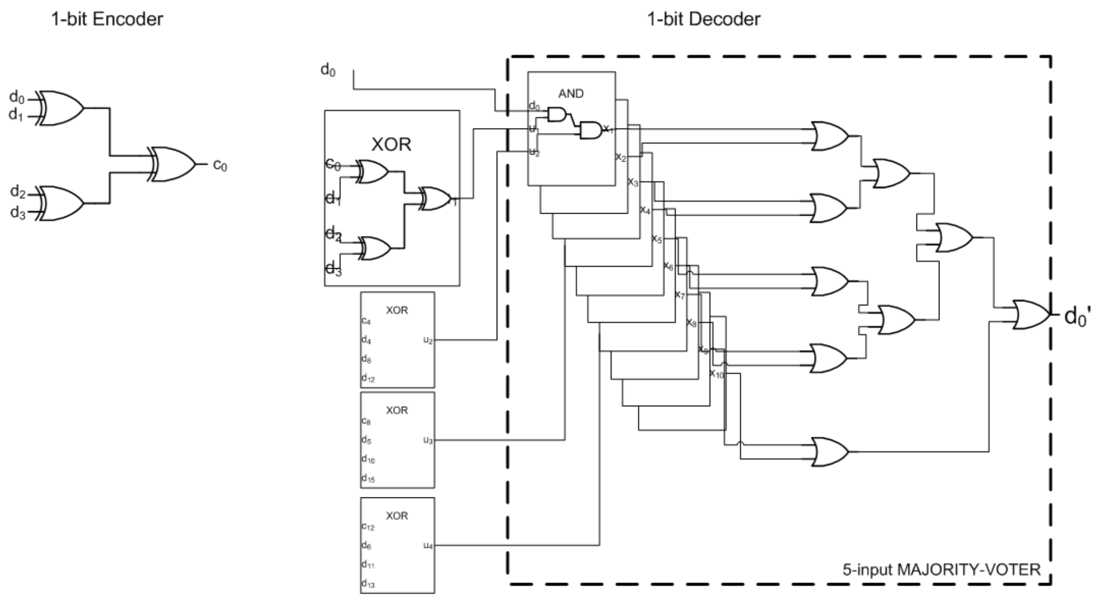


Figure 7.3: OLSC 1-bit Encoder and Decoder

encoding variable, such that;

$$\begin{aligned} d_0' = Majority \quad & (d_0, \\ & (c_0 \oplus d_1 \oplus d_2 \oplus d_3), \\ & (c_4 \oplus d_4 \oplus d_8 \oplus d_{12}), \\ & (c_8 \oplus d_5 \oplus d_{10} \oplus d_{15}), \\ & (c_{12} \oplus d_6 \oplus d_{11} \oplus d_{13})) \end{aligned}$$

The critical path for the decoder is ceil($log_2(m)$) levels of 2-input XOR plus (2t+1)-input majority function. Figure 7.3 presents the circuit logic for encoding and decoding single bit in an OLS code for an 16-bit data correcting up to 2 bit errors. We implemented the majority voter with 5 inputs required by the defined OLSC in a way that if any of the three inputs are 1, the result becomes 1, otherwise 0. In this design, while the critical path of the encoder consists of 2XORs, it consists of 2XORs+2ANDs+4ORs for the decoder.

7.2 Utilizin SEC-MEAC for Operating Below Safe V_{dd}

For Error Correcting Codes (ECCs) utilized in L1 caches, decoding latency is a key parameter since L1 is a time-critical component during the execution. Recently Reviriego et al. [9] have proposed Single Error Correction, Multiple Adjacent Error Correction (SEC-MAEC) codes whose encoders and decoders are very energy efficient and fast. These codes can correct single bit errors and also multiple bit errors as long as they affect adjacent bits. This is useful to correct multiple bit errors caused when a radiation particle hits the circuit. We argue that they are also potentially attractive to use in the high-fault rate below safe voltage operation, which also produces multiple bit errors. Additionally, the low decoding latency makes these codes attractive to protect L1 caches. Thus, in this chapter, we investigate using SEC-MEAC codes for scaling V_{dd}. In this section, we first explain the principles of SEC-MAEC. Then we present the architectural design of a L1 cache which utilizes SEC-MAEC under low V_{dd}.

7.2.1 Background of SEC-MAEC

In this subsection, we explain the principles of the SEC-MAEC code.

Figure 7.4 shows the encoder and decoder of proposed SEC-MAEC codes. The SEC-MAEC code with the ability of correcting s adjacent multiple errors in a data block consisting of k bits $\{d_0, d_1, d_2, ..., d_k\}$ generates k parity bits $\{p_0, p_1, p_2, ..., p_k\}$. In Figure 7.4, we illustrate the encoder and decoder of a SEC-MAEC code where s=2 and k=8. Depending on the size of the data block, the maximum value of s can be 2 (k = 8), 5 (k = 16), 10 (k = 32) and so on.

The encoder of the SEC-MAEC code computes parity check bits with the following equation:

$$p_i = d_i \oplus d_{mod(i-s,k)} \tag{7.1}$$

As it can be seen, the encode operation is accomplished for all bits in parallel, in one gate level. For instance, even single parity bit calculation in which all data bits are XORed, and only odd number of errors can be detected without correcting them, requires at least $log_2 k$ gate levels.

For a codeword (faulty or not) consisting of k data bits and k parity bits, SEC-MAEC decodes it in two stages: 1) SEC-MAEC generates k bit syndrome bits $\{s_0, s_1, s_2, ..., s_k\}$, 2) it

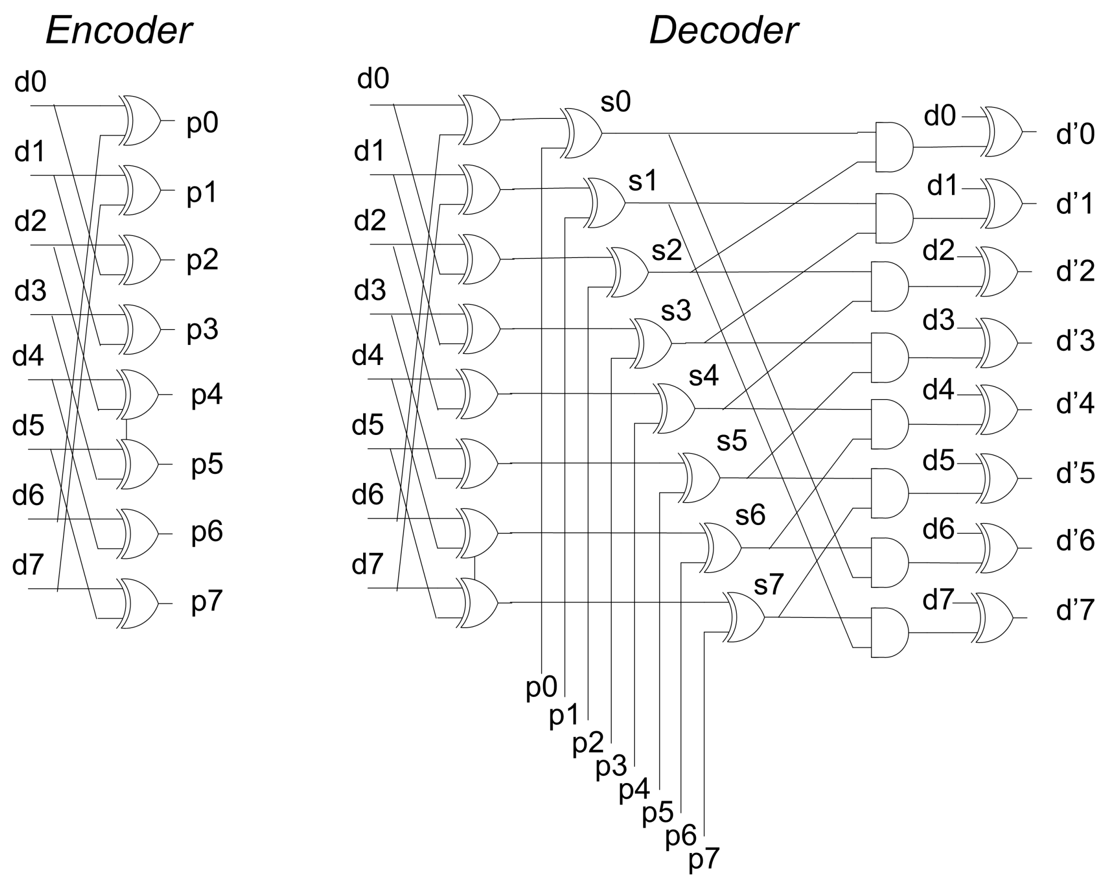


Figure 7.4: Encoder and Decoder of the SEC-MAEC code

calculates d', the corrected data after the decoding. Syndrome bits are calculated as:

$$s_i = p_i \oplus d_i \oplus d_{mod(i-s,k)} \tag{7.2}$$

Briefly, syndrome bits detect whether there is an error in the codeword or not. If all syndrome bits are zero, that means that the codeword is error free. After the syndrome bits, the correct data d' is also computed with the following equation:

$$d_i' = (s_i \cap s_{mod(i+s,k)}) \oplus d_i \tag{7.3}$$

The decoding logic of SEC-MAEC code is also quite simple and fast. More precisely, each correct d' bit is obtained with 4 gates (i.e. 3 XOR gates and 1 AND gate), and all bits can be calculated in parallel. Therefore, SEC-MAEC provides error correction with fast and

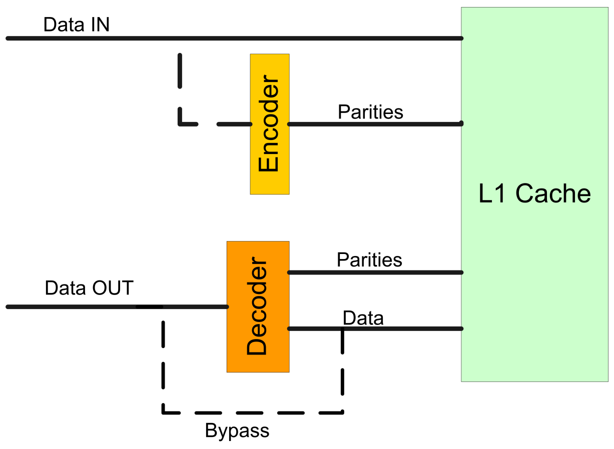


Figure 7.5: High-level overview of the architecture

area efficient encoders and decoders which makes it appealing to be used for low-power mode executions in the cache.

7.2.2 Architecture of Utilizing SEC-MAEC

In this subsection, we explain the architectural design of the first level caches which utilize SEC-MAEC for saving energy in the low-performance mode in which V_{dd} is reduced.

ECC schemes used for near-threshold voltages require a high number of parity bits (e.g 100% area overhead) in order to increase the error correction capability. However, this high area overhead does not present any benefit in the high-performance mode in which failure rate is very low. Thus, adaptive and variation-aware ECC-based schemes selectively enables protection for only the faulty cache lines in the low-power execution mode [122, 124]. In Figure 7.5, we present the high-level overview of the cache architecture protected by an ECC. All writes to the faulty cache lines go through encoder while reading from faulty cache lines go through the decoder. Also, for lines requiring no protection, encoding/decoding is bypassed, which reduces access latency and the area overhead.

In order to enable the protection for the faulty lines dynamically, the hardware should have the ability of determining the faulty lines. Failures due to voltage variation do not present a dynamic behaviour, and they occur persistently in the given supply-voltage level. Thus, faulty lines can be detected easily via simple built-in self test (BIST) in postmanufacturing or at boot time. BIST writes and reads two test patterns to each line: one containing all 0's and one with

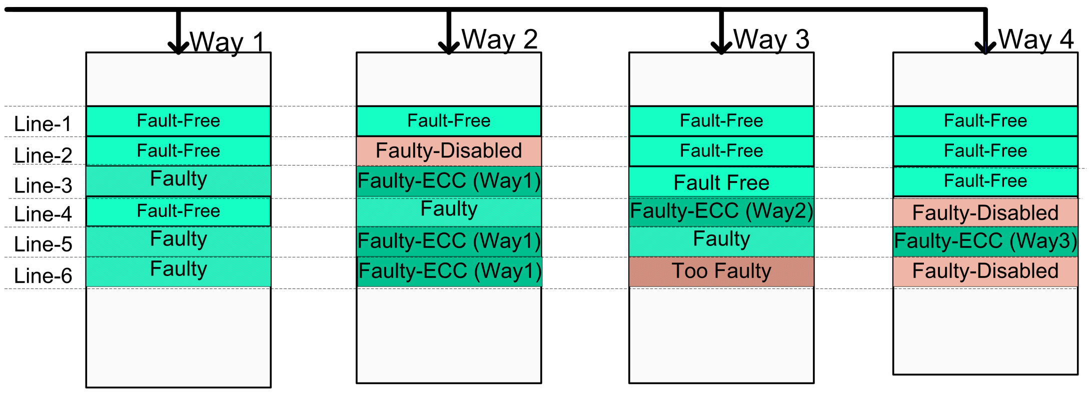


Figure 7.6: The organization of cache ways for faulty lines and the ECC values

all 1's, so that, it can determine the relative vulnerability of cache lines in each voltage interval. After the test, the cache configuration maximizing the capacity at each voltage interval is saved to some memory location (i.e. ROM or main memory) in the system.

It is not trivial to configure the cache when the supply voltage is lowered. In a naïve approach, the ecc-protection is disabled entirely in the high-performance mode, and it is enabled for all lines in the low-power mode after some V_{dd} level. This method has been utilized by MS-ECC in L1 cache [122]. However, due to the non-uniform distribution of errors, variation-aware protection algorithms that consider the relative vulnerability of cache lines are amenable in order to maximize the useful cache capacity. In this study, we present a variation-aware algorithm for the SEC-MAEC code when it is utilized in a 4-way L1 cache. In our algorithm, we inspired by the one utilized for 8-way L2 caches [124] in which higher number of ways in a cache line presents higher flexibility for the cache organization.

In Figure 7.6, we present an example for the organization of the cache ways for faulty cache lines and ECC values of the faulty lines. We do not extend fault-free cache ways with ECC so that in the high-performance mode execution, the entire cache capacity is available (e.g Line-1). When only one way is faulty in the cache line (Line-2), instead of allocating a fault-free cache line for ECC, we disable the faulty cache way. In this way, we present the same cache capacity (e.g. 3 out of 4 ways are utilized) without increasing the access time due to encoding/decoding logic. When there are 2 faulty cache ways in a line (Line-3), we combine these two ways. We save the data in one of them and the ECC to the other one. So that we can still provide 3 useful cache ways out of 4 ways. When there are 3 faulty-ways in a line (Line-4), we combine two of them and disable the third one. When all the ways are faulty (Line-5), we save data to

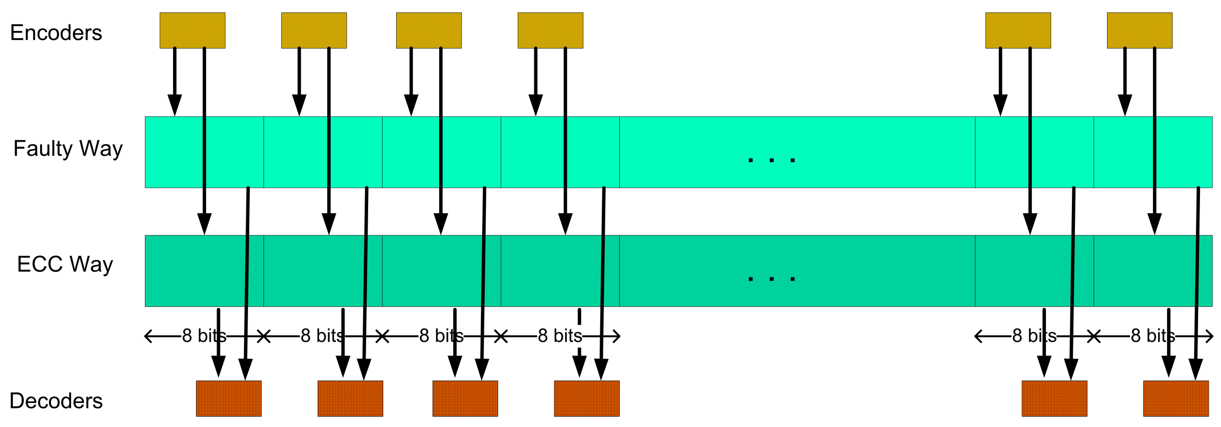
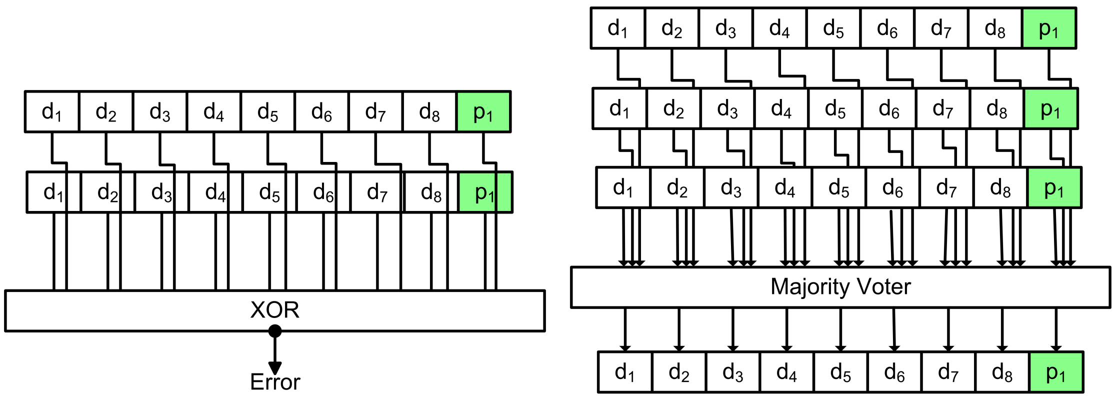


Figure 7.7: Direct mapping between the data and ECC partitions

two of them and ECC values to the other two. Finally, when the supply voltage is very low as in near-threshold execution, there can be a cache way which has more errors than it can be corrected by the utilized ECC scheme (Line-6). In this case, we disable this cache way as well. In order to accomplish this cache organization, we need to save two information for each cache way: 1) is it fault-free, faulty-data, faulty-ecc or disabled (2 bits) 2) if it is faulty, the address of the pair cache way for the ECC or data (2 bits). Thus, this cache organization presents 16-bit (4-bit in a 4-way cache) area overhead per a cache line which is a negligible overhead for 2Kb cache lines (512-bit per way). Also, it is possible to save this information in the tag area of the cache.

The second question for the SEC-MAEC usage in the L1 cache is the organization of the data-blocks. We use the SEC-MAEC with the configuration that k=8 meaning that the size of the data block is 8 bits. In order to apply it, we divide the cache lines into partitions consisting of 8 bits (i.e. 64 partitions in a 64B cache line). We also divide the ECC way into partitions with the same size (64 8-bit partitions) and combine the data partitions with the ECC partitions in the same order. We present this direct mapping in Figure 7.7.

For the 8-bit data sizes, when s=2, SEC-MAEC can correct two adjacent bit failures. Similarly, when s=3 it can correct 2 failures that are 1-bit far from each other. Thus, due to the random distribution of failures, some lines can be corrected with s=3 although they can not be corrected when s=2 or vice versa. In order to take advantage of both configurations for different lines, we decide s parameter dynamically during the BIST for each cache way. In order to accomplish this dynamic decision, we extend each cache-way with a single bit determining s



(a) DVM (Flexicache) (b) TVM (Flexicache)

Figure 7.8: The figure presents the basics Flexicache for DVM (Figure.7.8(a)) and TVM (Figure.7.8(b)) for 8-bit partitions.

parameter used (s=2 or s=3).

In a cache way, when the V_{dd} is in the middle-low level (i.e. lower than the safe margin but higher than the near-threshold level), since the bit failure rate is not drastically low, there could be several error-free partitions in a faulty cache way. One can think applying the optimization of extending only the faulty partitions with ECC values in order to reduce the ECC overhead. For instance if k of 64 partitions are faulty, parity bits for only those k partitions are calculated and saved. However, in this case, the ECC partitions need to include the address of the data partition that they are combined with. This addressing requires 8 additional bits (e.g. addressing 64 partitions in 4 ways) for each ECC partition which diminishes the benefit of ECC. Thus, in this study we avoid this optimization and we utilize direct mapping between the ECC and data values.

7.3 Flexicache: Circuit-Driven Replication

In this section, we present Flexicache. Flexicache is a novel, reliable cache design which configures itself for different supply voltages from the nominal to the near threshold voltage levels in order to duplicate or triplicate each data line when higher reliability is required. First, we show the architecture design of Flexicache. Then we present the circuit design of Flexicache by showing the details of sub-array and address decoder.

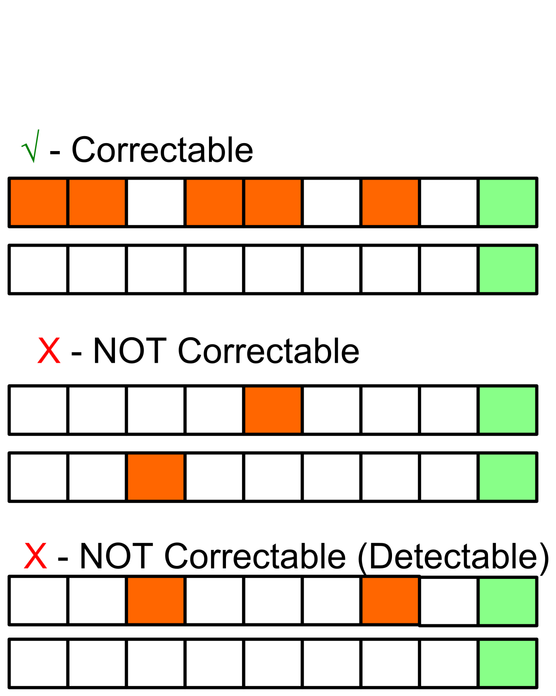
7.3.1 Architecture of Flexicache

Flexicache is designed for SRAM caches, which are resilient to several failures. Flexicache allows three modes of error protection according to the resilience level of the applied V_{dd}: Single Version Mode (SVM), Double Version Mode (DVM) and Triple Version Mode (TVM). Flexicache divides each cache line into parity-protected-partitions akin to many commercial L1 caches protected by single-bit parity in block, word or byte granularity [140].

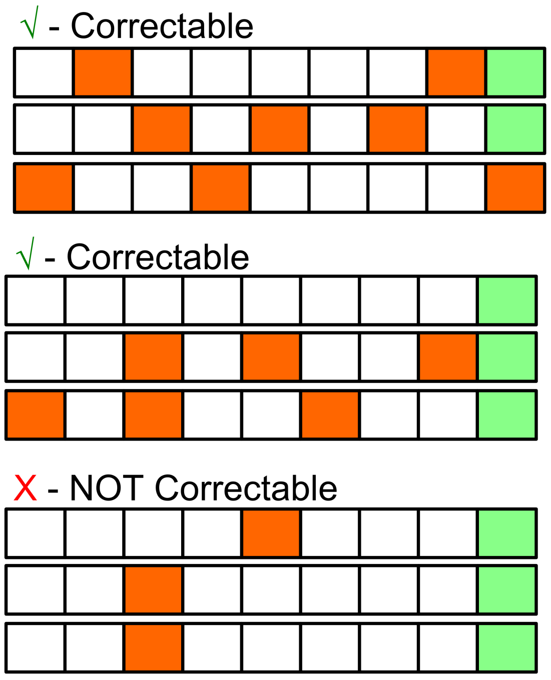
Figure 7.8 presents the design of DVM and TVM for a hypothetical 8-bit partition. SVM, which is not presented in the figure, provides reliability solely based on single bit interleaved parity. Note that in a bit interleaved cache, multiple data words are stored in an interleaved fashion along a single physical row of the cell array. In this way, physically-contiguous multi-bit errors affect logically different data word and parity protection can detect these multi-bit errors as if they are single bit errors. In this study, Flexicache runs in SVM in the nominal voltage when the failure rate is minimum in order to provide full cache capacity for the applications. Note that instead of parity, a stronger Single Error Correction Double Error Detection Code can also be utilized to provide a higher reliability for mission critical applications.

Flexicache runs in DVM when the V_{dd} is medium-low and writes the data to two cache lines. Note that the circuit design allows writing/reading multiple lines simultaneously (i.e. without increasing the access time) as we explain in the following section. In a read, DVM compares two duplicated, parity-protected partitions through the XORs to check if there is any fault. In case of the complete match, Flexicache dispatches one of the partitions to the output buffer. Otherwise, Flexicache calculates the parity of each partition and sends out the non-faulty partition which has the correct parity. DVM provides a backup copy for each partition. For instance, when a particle strike effects several adjacent bits in a line, the correct value is read from its replica without requiring any decode-and-correct time. In order to avoid the possibility of a strike affecting both coupled lines, Flexicache couples the lines with spatially distant locations. (e.g 0th and 63th lines.)

When the V_{dd} is near threshold, in order to tolerate the drastically increased error rate, Flexicache runs in TVM by writing the data to three cache lines simultaneously. On a read, Flexicache uses bitwise majority voting to obtain the correct data and calculates the parity of the data. Unless parity confirms that the result is correct, Flexicache calculates the parities of three partitions and sends out the correct partition. In TVM, the whole cache should be divided into three which is not trivial for a cache having 2^n lines. One solution can be manually connecting



(a) DVM



(b) TVM

Figure 7.9: Examples for correctable and non-correctable faults

lines by taking into account that the lines in the same group should be in distinct positions (e.g. 0th, 42th and 84th lines for a 128-line cache). However, this considerably increases the complexity of the address decoder. Instead, we add spare lines to make the cache dividable into three. For instance, for a 128-line cache, we add 16 spare lines and we connect every 48 lines.Note that using spare lines for tolerating yield loss is a common approach [135, 141] and, the area overhead due to extra lines is similar to this approach.

DVM can correct odd number of errors if they effect only one copy of the data (Figure 7.9(a)). However, if the faults are in different copies of the data, DVM can only detect the bit-positions of the faults without correcting them. Similarly, if there is even number of faults in one partition DVM cannot correct them, either. TVM (Figure 7.9(b)), on the other hand, can correct errors easily unless they are affecting the bits in the same position (it has a significant possibility in very high bit failure rate). Otherwise, after calculating the parity, TVM detects that the result of majority voter is not correct, and it can correct errors if one of the three copies is error-free. If all three copies are erroneous, and some errors are in the same

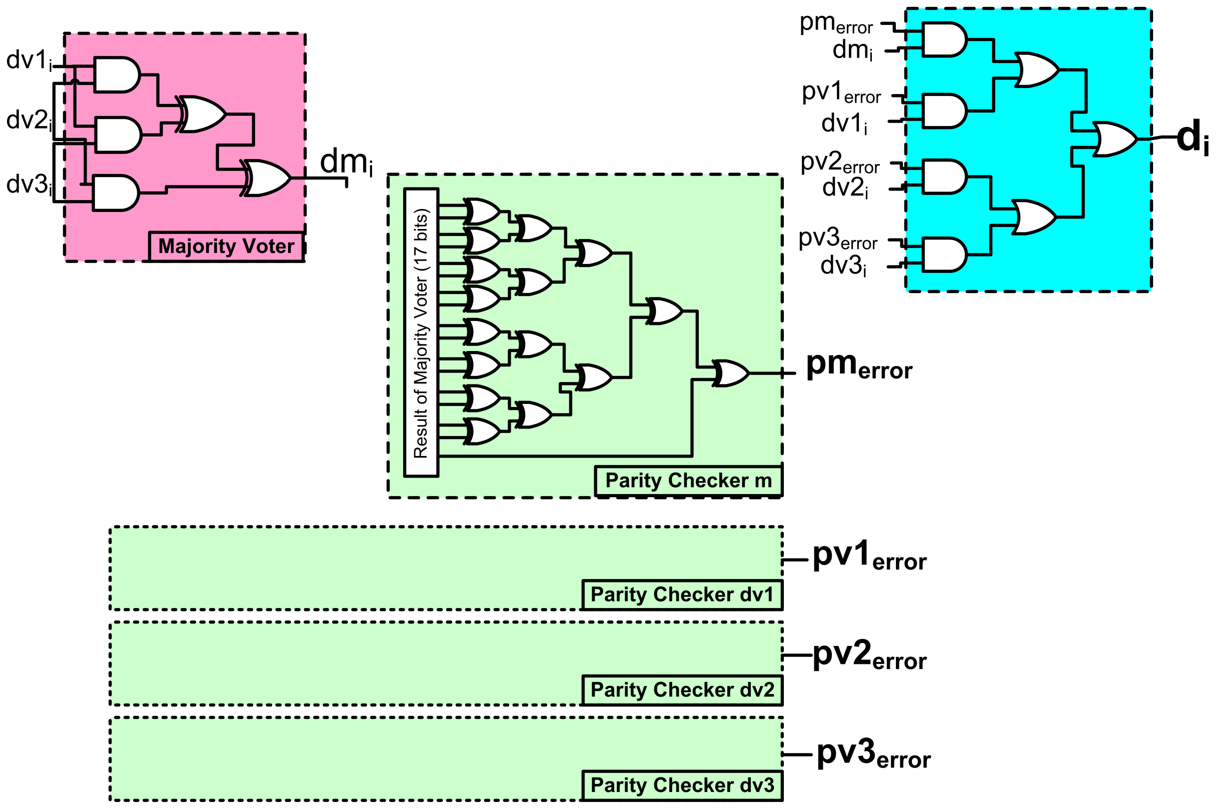


Figure 7.10: The figure shows 1-bit decoder when Flexicache works in TVM. Note that parity bit calculators are for per partition not per bit.

bit position, TVM can not correct the partition.

In Figure 7.10, we show the circuit design for decoding one bit when Flexicache is in TVM. In TVM each partition is protected by parity and these parities are majority voted as well. Thus, for each partition, there should be 4 parity calculation units. Also, one of them should wait until the result of the majority voter is produced while other three can be calculated earlier. The correct result is decided according to the parity check and the stored bits.

When there is an uncorrectable partition in a line, we utilize a partition-fix mechanism in DVM and TVM to avoid wasting the correct partitions. Partition-fix is similar to the bit-fix proposed by Wilkerson et al [133]. It uses a quarter of the cache ways to store locations and the correct values of defective partitions. This reduces both the cache size and associativity in the low-power mode. Thus, we utilize partition-fix mechanism only for the lines which have uncorrectable partitions. Note that, our partition-fix mechanism is different from the bit-fix for a non-persistent bit failure correction. In bit-fix, the cache lines are not protected by any other means, they only rely on memory tests and fixing the detected failures. In Flexicache, the fixed partitions are also protected by DVM or TVM which can still correct non-persistent failures. Previous triplication schemes [126, 127] write data to three cache lines and read the correct value from the majority voter. In Flexicache, partitioning and parity protection of each partition present higher error correction capability.

Persistent-fault tolerating proposals perform BIST [18] either postmanufacturing or at boot time to determine the uncorrectable cache lines at each voltage level [124, 133, 134]. These lines are stored in on-chip ROM or main memory and loaded before the processor transitions into near-threshold. For non-persistent failures, if the system can not correct a fault in L1 cache, either the correct value is re-fetched from L2 cache if the write-through cache is utilized or the system issues a machine check exception unless other means are utilized. Flexicache performs BIST test as in previous proposals to determine faulty partitions in order to fix them or disable the cache ways/lines including them. In runtime, Flexicache can detect and correct non-persistent failures, as well. For uncorrected non-persistent failures, Flexicache can utilize lightweight, global checkpointing such as SafetyNet [142].

7.3.2 Circuit Design

Conventional triplication schemes either write three lines sequentially [127] or increases the number of read/write ports [126]. While sequential writes harm application performance, increasing the number of ports increases the energy consumption.

Previously, Seyedi et al. designed dvSRAM which includes two values in each cell, primary value and secondary value [143]. These two values can be accessed, modified, moved back and forth between the main and secondary cells within the access time of the cache. Similarly, Flexicache needs to access replicated data within the cache access time with minimum energy. Armejach et al [144] present how a reconfigurable cache using dvSRAM circuits can be designed so that it can dynamically switch its configuration between a 64KB general purpose data cache and a 32KB special purpose, dual version using data cache. Flexicache also requires a reconfigurable cache design so that it can provide three different execution modes (i.e. SVM, DVM, TVM) not to sacrifice the cache capacity in the high-performance execution mode.

In this section, we elaborate how we can design the circuit of Flexicache for L1 data cache so that it can replicate cache lines without increasing access latency and with minimal energy

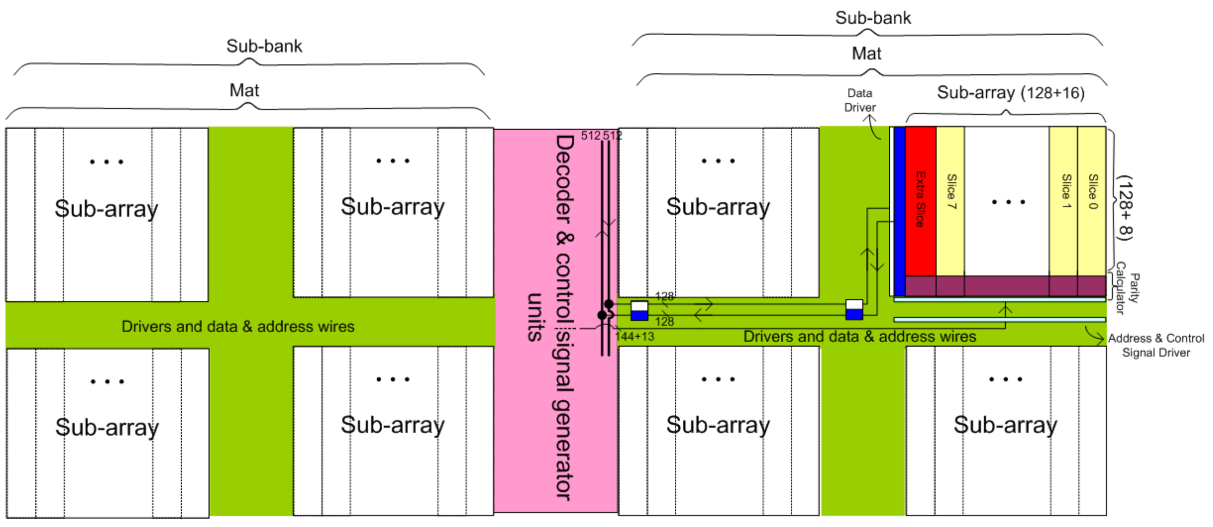


Figure 7.11: Block Diagram of a Bank in a 64KB, 4-way Flexicache.

overhead. Note that it is straightforward to extend the design for the instruction cache and the L2 cache. Flexicache can also be designed orthogonally to dvSRAM so that it can support both optimistic concurrency and near-threshold voltage execution that we leave it out of the scope of this study.

Block Diagram

In this section we present the design of Flexicache for 4-way, 64-KB data cache with 64-byte cache lines, and two clock cycle access time. Figure 7.11 presents the block diagram of one of 4 ways. We use the 45-nm Predictive Technology Model [145] with 1V supply voltage. We use Cacti [146] to determine the optimal number and size of Flexicache components (e.g. number of sub-banks) and the cache architecture with optimal access time and power consumption. For a one-bank array, Cacti suggests 2 identical sub-banks, 1 mat for each sub-bank and 4 sub-arrays in each mat (Figure 7.11). We utilize these high-level CACTI results as inputs to subsequent cache circuit design steps: we construct for one way Hspice transistor level netlist using 45-nm Predictive Technology Model [145]. During an access, only one of the two sub-banks (i.e. left sub-bank and right sub-bank) and four identical sub-arrays of the mat (i.e. each sub-array holds a part of the cache line) are activated. The address decoder and control signal generator units are placed in the middle part of the array. Necessary data and address wires and drivers are placed in the middle part of each sub-bank. Flexicache divides each sub-array to eight equal slices (i.e sub-array slice) each containing 16 lines with the individual precharged circuit, the write circuit, the sense amplifier circuit and input and output buffers. (64KB = 4 bank$\times$8 sub-bank/bank$\times$8 slice/sub-bank$\times$16 lines/slice$\times$128 bits/lines). Also, it extends each sub-array with an extra slice (i.e. to make it divisible by 3).

128+16 word-line addresses and 13 control signals are generated in the middle part of the array while 512 data-in/data-out bits are routed from the up side of the sub-bank. The necessary optimized drivers (chain of two series inverters) are in paths, and they can reach their related loads in the sub-arrays. We place necessary control signal unit circuits in the middle of Flexicache which generate suitable enable signals according to the V_{dd} level. The enablers and control signals activate the pertinent parts of sub-arrays while inactivating others. They also select the buffers which should be activated to transfer the appropriate data to the output.

Details of Sub-array

Figure 7.12 presents the abstract view of the block diagram of one sub-array in Flexicache. According to the decoded addresses and the V_{dd} level, one, two or three slice(s) are activated and the data coming from the bus is written to the enabled slice(s). Cosemans et al. [147] evaluated the energy consumption of the cache elements during read or write operations in a design based on 90nm technology. For instance, during the read operation, timing components (including delay elements and control wires) is the most energy consuming element (i.e. 30%). Similarly, address decoder consumes around 25% of the read/write energy. Since Flexicache still uses the most of the energy-hungry components (e.g. buses, data drivers and the address decoder) only once in DVM and TVM, it slightly increases the energy consumption of timing elements and the address decoder. On the other side, Flexicache only duplicates (triplicates) the energy consumption of cells and sense amplifiers which consumes less than 15% of the read/write energy. Thus, Flexicache presents modest additional energy consumption in DVM and TVM for reading and writing multiple cache lines. Note that in reading energy, we place XORs and majority voters close to the cells so that the corrected data can go through single bus without duplicating/triplicating the bus energy.

Figure 7.13 shows the block diagram of each sub-array structure in detail. The figure presents the necessary buffers, comparators, parity calculators and control and data lines in detail. For writing the selected cache line in SVM, signal IEU1 is high and activates input

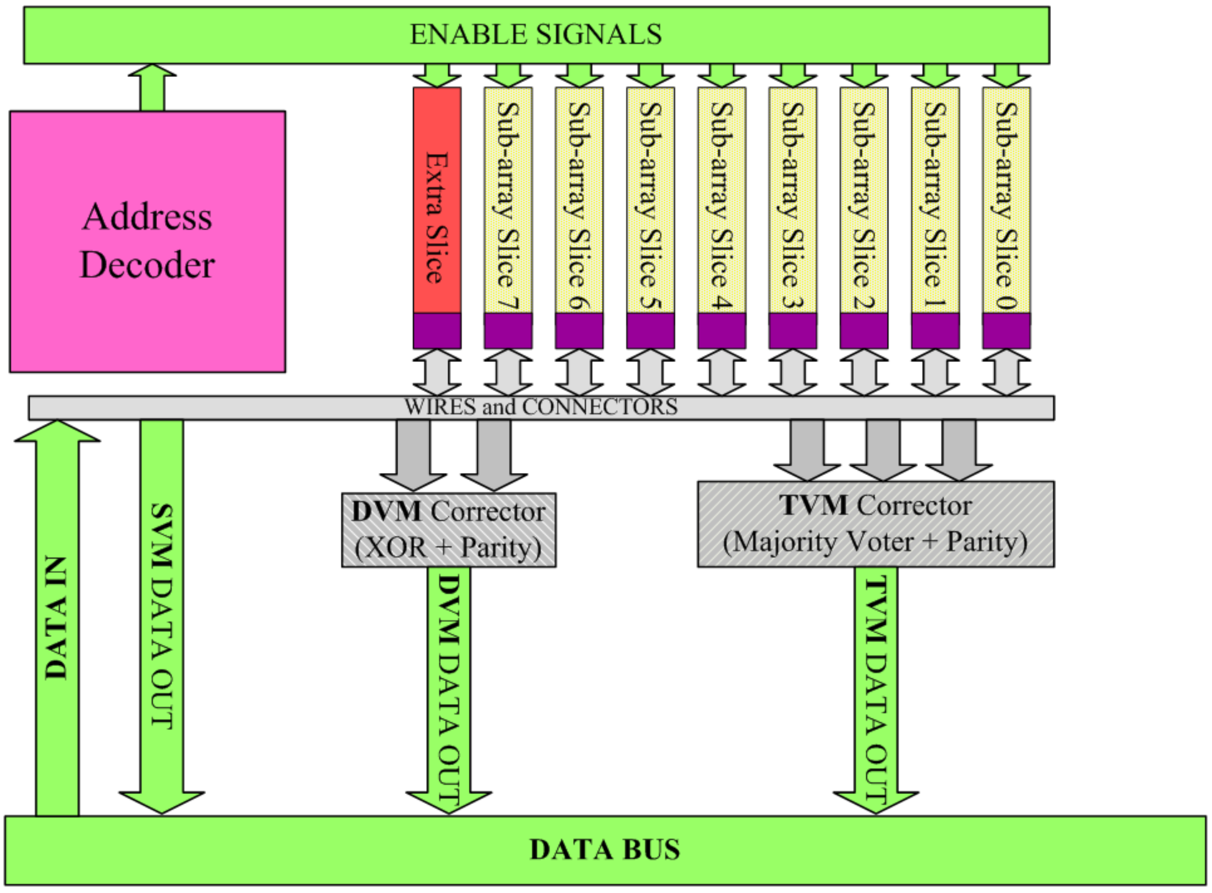


Figure 7.12: The figure presents the basic components of Flexicache such as buses and decoder

buffers IB1 and IB2 and data can transfer to the selected cache-line via Bus4 and Bus1; and similarly for reading the selected line, signal OEU1 is high and output buffers OB6 and OB13 are active and data is transferred from Bus1 to Bus3. Bus3 (Bus4) is connected to output data drivers (input data drivers) which are located close to each sub-array. At each access time, the enabler signals (CDE and CTE) are high and activate connector buffers, CD1, CD2, CD3, CD4, CT1 and CT2 and connect nodes B1, B2, B3, B4 and B5 to each other (Each connector buffer contains two series inverters with enablers). Similar to many typical L1 caches error protection is based on bit-parity calculation in order to achieve high performance. We divide each cache-line into 8 partitions each contains 16 bits where each interleaved parity protects one partition. At each reading time parity bit calculated and compared with the original parity bit.

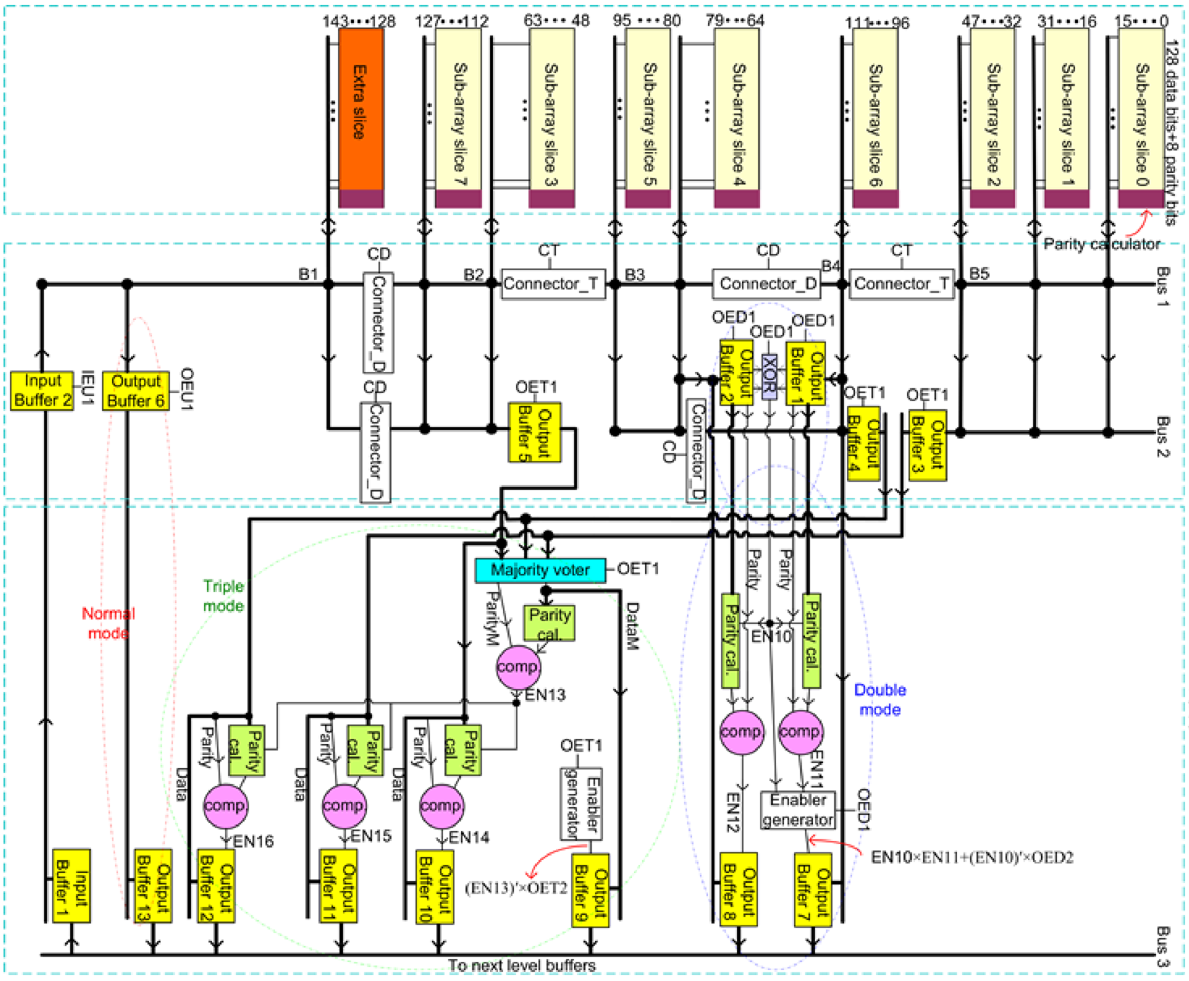


Figure 7.13: Address Decoder and Sub-Array in Flexicache

For writing in DVM, signal IEU1 is high and data is transferred from Bus4 to Bus1 via IB1 and IB2 and is written to two selected lines at the same time. Parity calculator circuits generate parity bits and write them in parity bit cells as well. For reading the two selected lines, signals CTE is high and CDE is low, connector buffers, CD1, CD2, CD3 and CD4 disconnect B1 with B2 and also B3 with B4 while connector buffers, CT1 and CT2 connect B2 with B3 and also B4 with B5. With this method, Bus1 is divided into two parts; sub-array slices 0,1,2,6 are connected to the first part and sub-array slices 3, 4, 5, 7 are connected to the second part. Signal OED1 is high; output buffers OB1 and OB2 transfer two selected data to the XOR circuit to check the cell contents are identical. Signal EN10 activates two parity calculator circuits to calculate parity bits of selected lines. Then the result of these parity calculator circuits are compared with the original parity bits of each selected-lines. These two comparators generate two enable signals EN11 and EN12. Whenever one of comparator shows equality (when EN11 or EN12 is high), the related output buffer transfers its data to Bus3. If two compared data are equal, signal EN10 is low and output buffer OB7 transfers data to BUS3.

In TVM, for writing the selected cache-lines signal IEU1 is high and data is transferred from Bus4 to Bus1 via IB1 and IB2 and written in three selected lines simultaneously. Similar to SVM and DVM, signals CDE and CTE are high to connect separated Bus1 nodes with each other. At the reading time, CTE is low and CDE is high so Bus1 is divided into three parts; sub-array slices 0, 1, 2 are connected to the first part and sub-array slices 6, 4, 5 are connected to the second part and sub-array slices 3, 7 and extra slice are connected to the third part. CT1 and CT2 circuits disconnect B2 with B3 and B4 with B5. For reading the three selected lines, each from separate sub-array slice group, signal OET1 is high, OB3, OB4 and OB5 are active and data including parities are transferred to a majority voter (the correct value is decided by bit-wise majority voter). The majority voter output for cache-lines is DataM and for their parity-bits is ParityM. Then, the parity calculator circuit calculates the parity bits of DataM. Later one comparator circuit compares these results (the parity bits of DataM) with ParityM. If there are any differences, signal En13 will be high and the parity bits of selected lines are calculated and compared with their original parity bits. Whenever one of parity comparators shows equality, En14, En15 or En16 is high, the related output buffer (OB10, OB11 or OB12) is active and transfers data to Bus3. If signal En13 is low, DataM is transferred to Bus3 via OB9.

Address Decoder

Figure 7.14 presents an abstract view of the address decoder. In the figure, A_0 to A_7 represents the addresses bits. The decoder uses the 4 least significant bits (i.e A_0 to A_3) in order to address the line number within a slice. Also, it uses A_7 to activate either the left sub-bank or the right sub-bank. *Voltage Level Detector* activates either SVM, DVM or TVM. These three signals together with A_4 to A_6 generates enable signals (EN0 to ENex) which activate slice(s). At each time, depending on the mode, one, two or three Enable Signals are high and data is written to (and read from) one two or three cache lines simultaneously.

The details of address decoder are present in Figure 7.15. Voltage level detector circuit

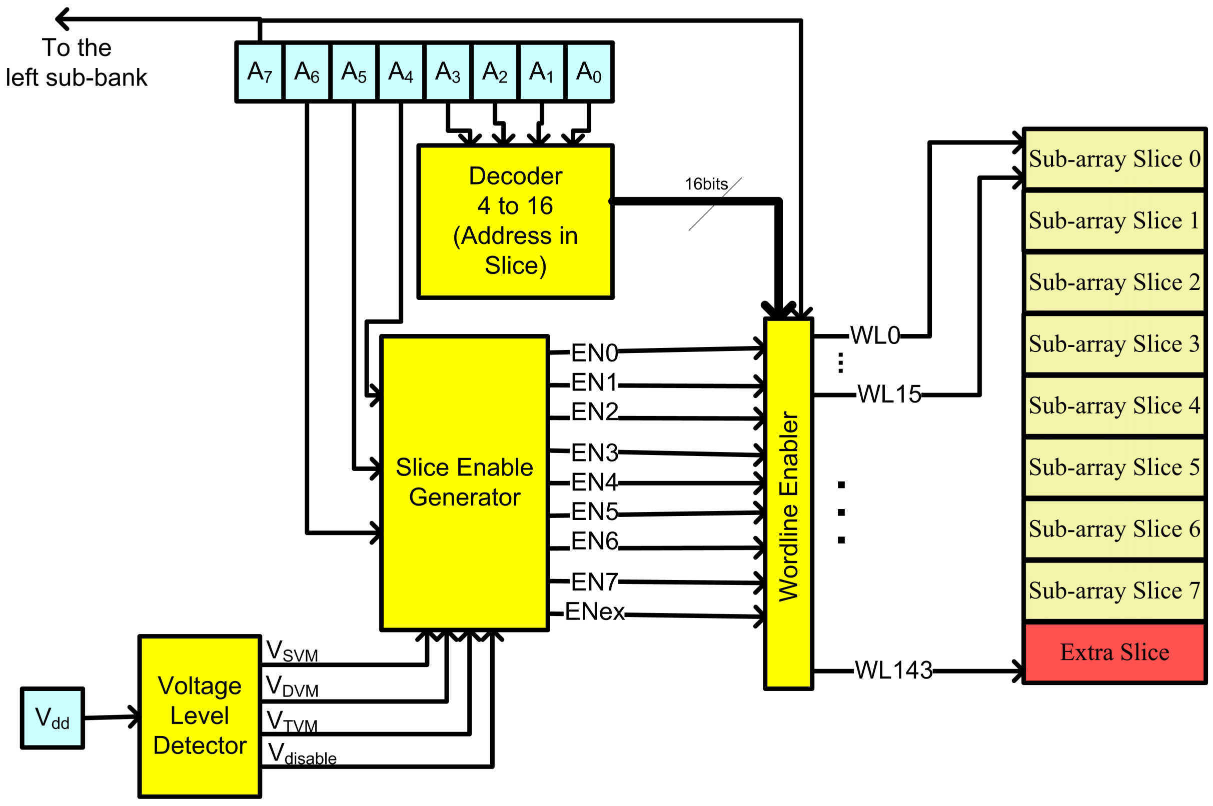


Figure 7.14: Abstract view of the address decoder of Flexicache

generates four output signals V1, V2, V3 and V4 according to the supply voltage, V_{dd}; if V1 is high, the cache is in SVM and only one word-line address is activated at each access time; if V2 is high, the cache will be in DVM and two word-line addresses will be activated at each access time; if V3 is high, the cache will be in TVM and three word-line addresses will be activated at each access time; if V4 is high, the supply voltage level is lower than the threshold voltages and the memory cells operate in sub-threshold mode which is beyond our work in this paper and we leave it for future; so the cache-lines will be deactivated in this state. Pre-decoder 2, control signal generator unit 2 and control signal generator unit 3 and control signal generator unit 4 generate enabler signals, En1, En2 . . . and En9 to activate 144 word-line addresses. There are two groups of buffers located in the right and left side of the pre-decoder 1. Each buffers group contains 9 sub-groups, and each sub-group has 16 buffers. The outputs of pre-decoder 1 are connected to the buffers of each sub-group and generated 144 word-line addresses. All buffers of each sub-group are activated with one enabler signal. For example all buffers of sub-group 1

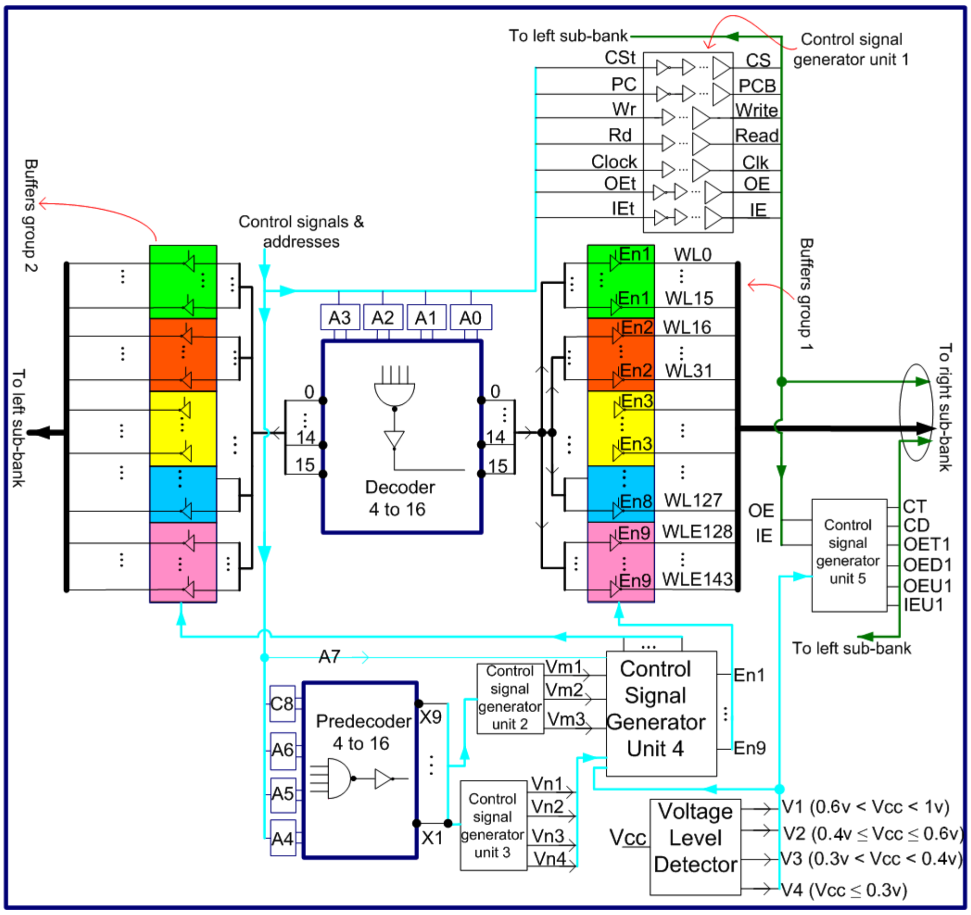


Figure 7.15: Necessary decoders and control signal generators

are enabled by signal En1. When partial address 0 and En1 are high, WL0 will be generated. In this way, all word-line addresses from 0 to 143 are generated. If A7 is high, the left part of each cache way is activated; similarly, If A7 is low, the right part of each cache way is activated. At each access time, depending on the mode, one, two or three Enable signals are high and data is written to (read from) one, two or three cache-lines simultaneously. For example in DVM, WL0 and WL48 are activated simultaneously; whenever one of the addresses 0 or 48 are activated, X1 or X4 are high and Vn1 is high so En1 and En4 are high. En1 and En4 are enablers for buffers and let partial address 0 pass and generates WL0 and WL48.

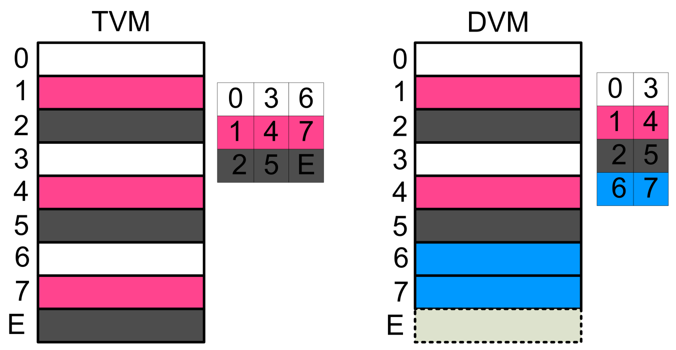


Figure 7.16: Slices activated at a time in DVM and TVM

Switching Between Modes

The V_{dd} can be increased or decreased in the runtime, thus, Flexicache needs to switch between modes. In a naïve approach, before mode switching, the whole cache is flushed which presents a cache warm-up performance overhead immediately after switch. In this section, we present a more efficient approach. We organized the activated slices in each mode in order to ease the switching. In Figure 7.16, we present the activated slices at a time during the read/write operation of DVM and TVM.

In order to switch TVM→DVM→SVM, it is adequate to flush the slices in the last column of the old mode in the tables shown in Figure 7.16. In another word, when Flexicache switches from DVM to SVM, slices 3, 4, 5 and 7 are flushed from the cache. Similarly, when Flexicache switches from TVM to DVM, slices 6, 7 and Extra slice are flushed. Also, if Flexicache switches from TVM to SVM (although many systems do not allow this fast voltage increase), combination of both columns (i.e. slices 3, 4, 5, 6, 7, Ex) are flushed. Obviously, before this flushing operation, the slices which are not flushed (i.e staying slices) should be corrected with the old mode. One option can be stopping the execution of the application right after the voltage increase, using the to-be-flushed lines for correcting the staying lines by utilizing the old mode and continuing the application execution after all staying lines are corrected. In the second option, in order to avoid this stopping overhead, all staying lines are traced after changing the mode. When a line is read for the first time after the mode change (or a dirty line is evicted from the cache), this line is corrected by using the old mode. The second or the third replica of the line can be flushed after this correction. If a line is written without reading after changing the mode, the flushing can be done without requiring any correction.

However switching SVM→DVM→TVM is not that trivial since the correct data should be updated in the second or third replica before reducing the supply voltage. Thus, for instance, when Flexicache switches from DVM to TVM, before reducing the supply voltage, lines in the slices 6, 7 and Ex are first evicted from the cache. Then, these lines are updated as the third copy. As an example, lines in slice 6 should be updated by reading the lines in slices 0 and 3 and obtaining the correct data via DVM circuit. It is only safe to reduce the supply voltage after that. Although switching SVM→DVM→TVM present the performance overhead of a runtime barrier for updating the second or third copies, it is not a show-stopper since this switch operation is required when going towards low-power mode from the high-performance mode meaning that the application can trade off the performance for power.

7.4 Evaluation

7.4.1 Evaluation of Utilizing SEC-MAEC

In this subsection, we evaluate how much energy reduction in L1 caches can be provided by SEC-MAEC code. We compare SEC-MAEC with Orthogonal Latin Square Code (OLSC) which is the state of the art, multi-bit correcting ECC used for the voltage scaling in L1 caches [123]. The complexity of OLSC scales well with the number of error corrections, thus, Chishti et al. [122] use OLSC for L1 caches. We utilized the OLSC code with a block size of 16 with the error correction capability of up to 2 failures in a block. Note that OLSC block size is optimized for the block size of m^2, and its encoder/decoder complexity increases when the larger blocks are used. Thus, we choose the best combination for OLSC.

We analyse 1) useful cache capacity, 2) error correction latency, 3) area overhead and 3) energy minimization. We calculate the useful cache capacity as the ratio of the portion of the cache which can be used reliably (i.e. the lines which are not disabled after the memory test at the boot time). We use fault injection methodology to measure the cache capacity under the bit failure rates (i.e. probability of a bit fails). We repeat each fault injection experiment 100 times for each failure rate.

In Figure 7.17, we compare the useful cache capacity provided by SEC-MAEC and OLS codes versus the bit failure rate. In order to present the benefits provided by cache-way-organization and dynamic-s-decision separately, we show two configurations of SEC-MAEC in the figure. SEC-MAEC (s=2) presents the benefit of cache-way organization on useful cache capacity. SEC-MAEC (s=2/s=3) presents the benefit of dynamic decision of s-parameter on

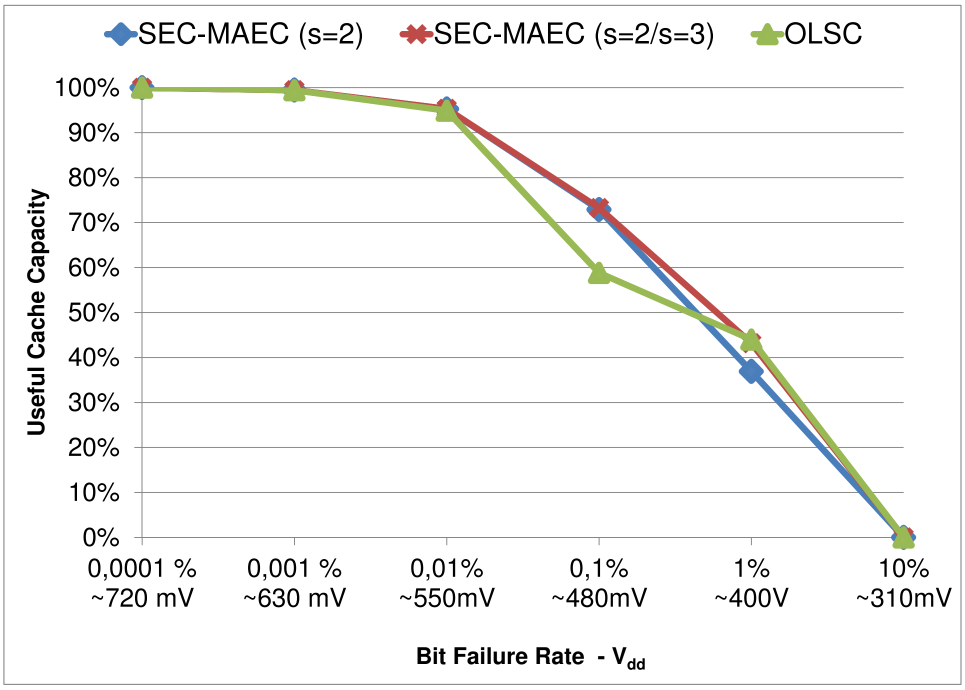


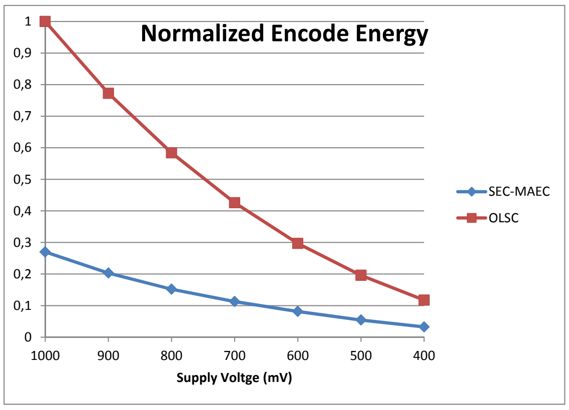
Figure 7.17: Useful Cache Capacity

top of cache-way organization. Both SEC-MAEC and OLSC provide similar cache capacity when the V_{dd} is close to the safe operating margin (i.e higher than 550mV). When the V_{dd} is middle low (i.e. between 550mV-430mV), SEC-MAEC extends some of the cache ways with ECC protection. On the other hand, OLSC does not activate the ECC protection for any cache way when the V_{dd} is higher than a threshold value. Note that, this threshold value should be determined as the V_{dd} level in which provided useful cache capacity without ECC protection is around 50%. Thus, when V_{dd} is middle low, SEC-MAEC presents higher cache capacity due to its cache organization in the architecture. In the near-threshold voltage execution (lower than 450 mV), while the cache capacity provided by SEC-MAEC (s=2) is lower than OLSC, SEC-MAEC (s=2/s=3) presents a high cache capacity similar to OLSC.

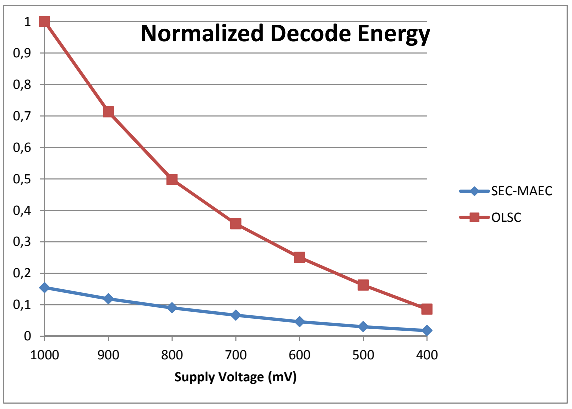
In Table 7.3, we compare the area overhead and the latency presented by encoders and decoders in SEC-MAEC and OLSC. We first present the number of gates in the critical paths. SEC-MAEC has nearly half number of gates in the critical path compared to OLSC both for the encoder and decoder. This also affects the time spent in the encoder and the decoder, thus, the latencies of SEC-MAEC are much less than OLSC. Note that in this study, we only target correcting V_{dd} dependent, persistent failures, and soft errors are an orthogonal issue. We

	SEC-MAEC		OLSC	
	Encoder	Decoder	Encoder	Decoder
Number of Gates in the Critical Path	1 XOR	3 XORs + 1 AND	2 XORS	2 XORs + 2 ANDs + 4 ORs
Total Number of Gates	512 XORs	1,5K XORs + 512 ANDs	1,5K XORs	6K XORs + 10K ANDs + 4,5K ORs
Latency	39 ps	182 ps	78 ps	324 ps

Table 7.2: Number of Gates in Encoder and Decoder for a 64B cache lines



(a) Encode Energy



(b) Decode Energy

Figure 7.18: Encode/Decode Energy Normalized to the energy of the encoder/decoder of OLSC at 1V

assume that the memory structure is protected for soft errors by any other additional means such as parity or Single Error Correction Double Error Detection (SECDED) Code. The area overhead of parity/SECDED bits is relatively low, and they can be saved to the tag area of the cache. Obviously, when the V_{dd} is below the safe margin, the decoder of parity/SECDED can be activated after persistent failures are corrected. Nevertheless, the low latency of the decoder of SEC-MAEC codes leaves additional time for the decoder of soft error protection.

Similarly, the area overhead of the encoder and the decoder in SEC-MAEC is significantly smaller than the one in OLSC. While the encoder of SEC-MAEC is only one-third of the encoder of OLSC, the decoder is in the size of 10% of the one in OLSC. This saving in the area overhead is proportional with the static energy consumed by the encoders and the decoder. We also compare the dynamic energies of the encoders and decoders in each supply voltage level in Figure 7.18. In the figure, we normalize energies to OLSC. Obviously, SEC-MEAC presents substantially less energy consumption for both encoder and decoder.

7.4.2 Evaluation of Flexicache

In this section, we compare Flexicache against a conventional triplication scheme (TMR) and MS-ECC [122]. Note that MS-ECC utilizes OLSC that we explain the details of it in Section.7.1.3. We use 4-way set associative, 64KB L1 cache with 2-cycle access time, 64B line size. We divide each line into 32 partitions for both OLSC and Flexicache with the partition size of 16 bits. For the design of Flexicache, we use the 45-nm Predictive Technology Model [145] with 1V supply voltage. We use Cacti [146] to determine the optimal number and size of Flexicache components (e.g. number of sub-banks) and the cache architecture with optimal access time and power consumption. For a one-bank array, Cacti suggests 2 identical sub-banks, 1 mat for each sub-bank and 4 sub-arrays in each mat. We utilize these high-level CACTI results as inputs to subsequent cache circuit design steps: we construct for one way Hspice transistor level netlist using 45-nm Predictive Technology Model [145].

Flexicache targets to tolerate ultra high bit failure rates occurring in the near-threshold voltage level without harming the performance of the cache in the low error rate. For the calculation of the V_{dd} that Flexicache operate reliably, we inject persistent faults into random locations according to bit failure rate (i.e. probability that a single bit fails) given in [124]. We calculate the useful cache capacity as the portion of the cache which is not disabled. For non-persistent failure such as soft errors, we inject multi-bit failures varying between 1 to 10 bits. We present the experimental results for the aspects of 1) useful cache capacity, 2) error correction latency, 3) energy reduction of cache operations, and 4) reliability against non-persistent faults (mean time to failure) and 5) uniform view of the cache. 6) area overhead,

Useful Cache Capacity

Figure 7.19 compares the cache capacities. We extend Flexicache with extra slices in order to make it divisible to three, and we normalized the useful cache capacity to the non-extended capacity for fair comparison. First, when the V_{dd} is high, Flexicache do not sacrifice the useful cache capacity due to its flexible circuit design which dynamically switch its configuration to 64KB general purpose data cache (i.e. SVM) in the high-performance mode. Second, due to

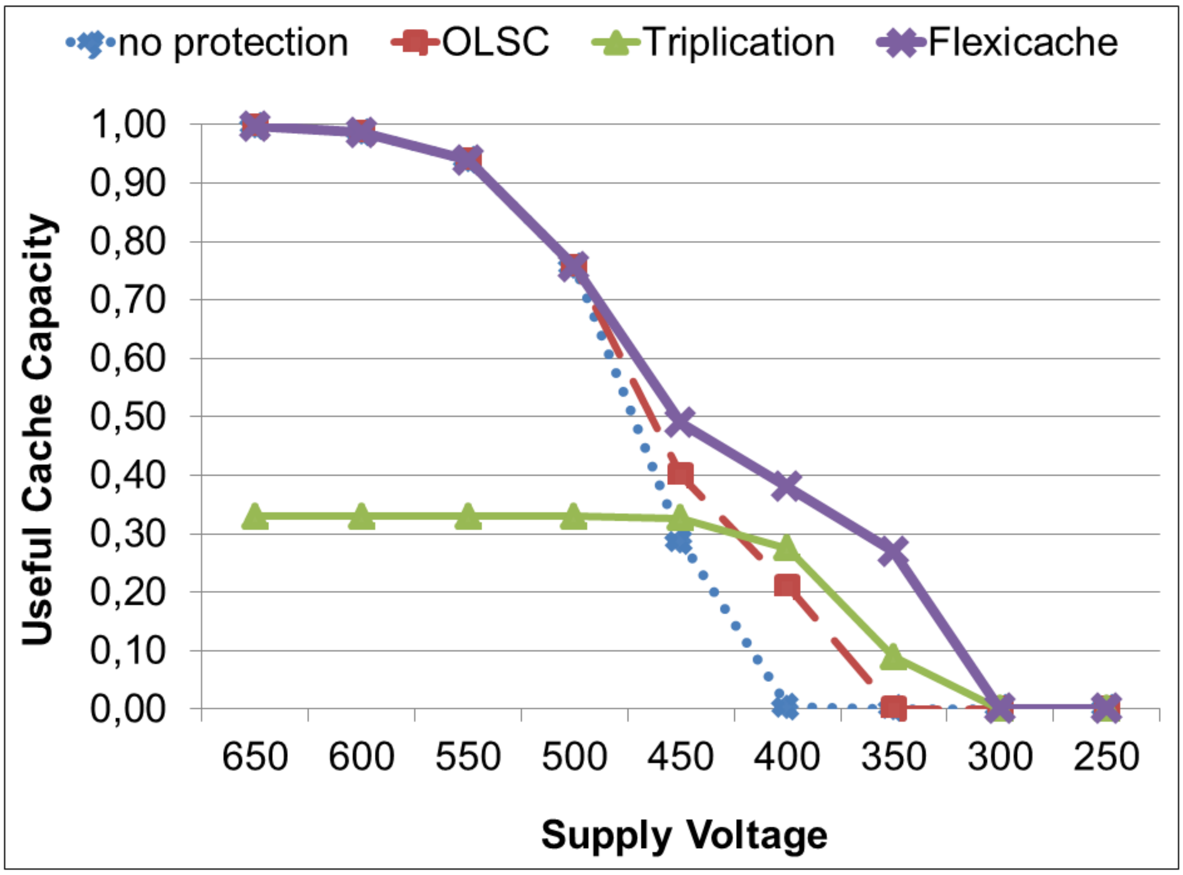


Figure 7.19: Useful capacity after disabling uncorrectable lines.

the partitioning and partition-fix mechanism of Flexicache, it provides higher cache capacity than the conventional triplication schemes even in the low-power mode. Third, Flexicache can operate until the persistent bit failure rate is 12% while TMR can operate until 6% bit failure rate and MS-ECC can operate until 2% bit failure rate (Bit failure rates are not shown in the graph). Therefore, TMR and MS-ECC can provide more than 20% of the cache capacity when the supply voltage is as low as 400mV while Flexicache can provide the similar amount of useful cache capacity when the supply voltage is 320mV.

	Flexicache		MS-ECC	
	Encoder	Decoder	Encoder	Decoder
Number of Gates in the Critical Path	4 XORs	7 XORs + 2 ANDs + 2 ORs	2 XORS	2 XORs + 2 ANDs + 4 ORs
Total Number of Gates	480 XORs	3K XORs + 1,5K ORs + 3,5K ANDs	1,5K XORs	6K XORs + 4,5K ORs + 10K ANDs
Latency	1 cycle	1 cycle	1 cycle	1 cycle
Energy Overhead (In the nominal voltage)	2,5%	20%	5,5%	50%
Area Overhead (Encoder+Decoder)	0.06%		0.12%	

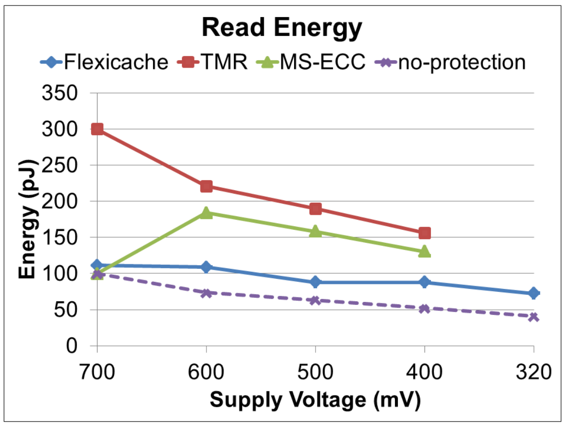
Table 7.3: The table analysis the area overhead and latency.

Latency

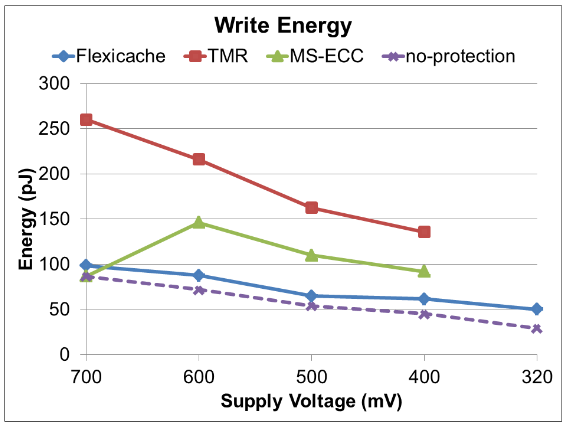
In Table 7.3, we compare the area overhead and the latency presented by encoders and decoders in Flexicache and OLSC. We first present the number of gates in the critical paths. Although, in Flexicache, the number of gates in the critical path are higher than the one in MS-ECC, both encoding and decoding in each scheme can be accomplished in 1 cycle. Note that the decoding latency can be tolerated since decoding is done simultaneously with writing. On the other hand, total number of gates in the encoder and decoder of MS-ECC is much higher than the one in Flexicache which presents higher overhead in both read/write energies (4th line in the table) and area (5th line in the table). Both Flexicache and MS-ECC require changes in the address decoder of the cache to be able to write more than one line simultaneously. The overhead of these address decoders are similar in both schemes.

Energy Reduction

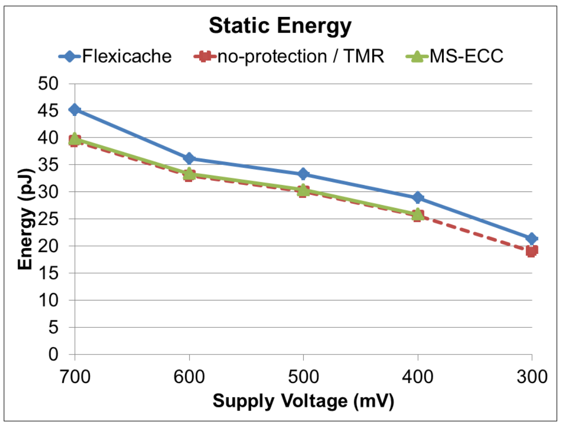
Figure 7.20 presents the energy consumption of cache operations (i.e. read/write energy and static energy). For read and write energies, TMR allocates three cache ways in a non-modified cache which triplicates the energy consumption. Similarly MS-ECC allocates two cache ways (1 for data and the other for parity bits) when the supply voltage is lower than 700 mV, thus at this point MS-ECC also roughly duplicates reading and writing energies. This is mainly



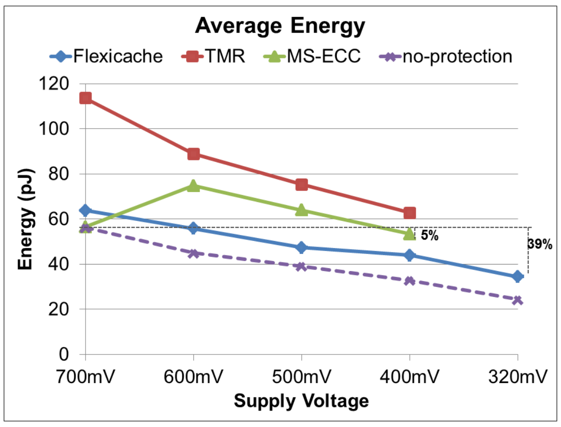
(a) Read Energy



(b) Write Energy



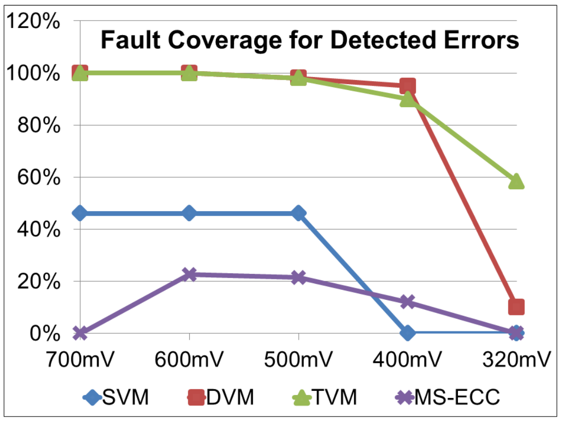
(c) Static Energy



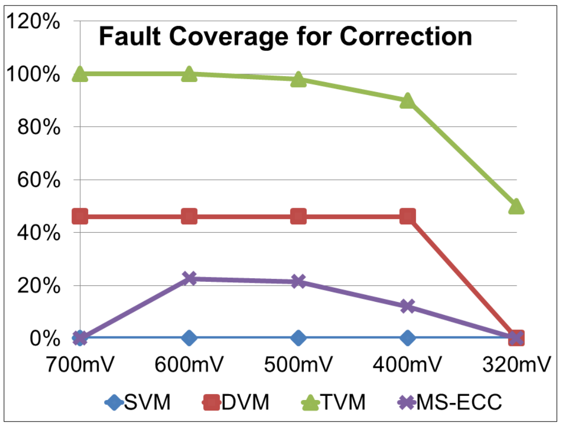
(d) Average Energy

Figure 7.20: Energy reduction in cache operations

because the size of the in/out data is duplicated (or triplicated). Also, the energy consumption of the OLSC decoder is very high (i.e. 50%). Thus, which diminish the energy saving of scaling voltage for read energy as it can be seen at 600mV when OLSC is activated in MS-ECC (Figure. 7.20(a) and Figure. 7.20(b)). On the other hand, Flexicache accomplishes replication and fault recovery within a way without increasing the size of the data in/out bus coming to the way. Thus, reading and writing energies of Flexicache is much lower than MS-ECC and triplication. For the static energies (i.e. energy spent in one cycle when the cache is idle), Flexicache presents slightly higher energy consumption than an unmodified cache mainly due to the additional extra slices (Figure. 7.20(c)). Note that these additional slices also increase the cache capacity that we excluded this increased capacity in our previous results. The static



(a) The percentage of **detected** errors.



(b) The percentage of **corrected** errors.

Figure 7.21: Non-persistent fault injection

energy consumption of MS-ECC is negligibly higher than a non-modified cache due to OLSC encoder/decoder. It has been showed that dynamic energies are only the 30% of cache energy consumptions and among them they are mostly (two out of three) read operations. By considering that, in Figure 7.20(d), we present the average energy consumption of a cache at a time. The figure shows that only Flexicache can operate when V_{dd} is 320 mV by presenting 39% reduction in the energy consumption compared to non-modified cache when it executes in the high-performance mode with the minimum safe V_{dd} (i.e. 700 mV). MS-ECC can reduce the energy consumption by only 5% compared to the same minimum safe voltage level.

Reliability against Particle Strike

In Figure 7.21(a) and in Figure 7.21(b), we inject non-persistent, multi-bit faults (i.e. size of the faults are between n=1-10 bits which means n adjacent bits become faulty due to a particle strike) to the non-disabled cache portion and, we present the fault coverage for error detection (Figure 7.21(a)) and error correction (Figure 7.21(b)). We define the fault coverage as the percentage of the injected faults which are detected or corrected. In the high-performance mode, MS-ECC can not detect or correct non-persistent faults since it does not extend the cache lines with ECC codes. On the other hand, each cache line is extended with ECC protection in the low-power mode when the persistent fault rate is very high. At this point, additional multi-bit non-persistent faults lead the total number of faults in the cache line to be higher than OLSC can correct. Thus, non-persistent fault correction capability of MS-ECC is around 20% or less.

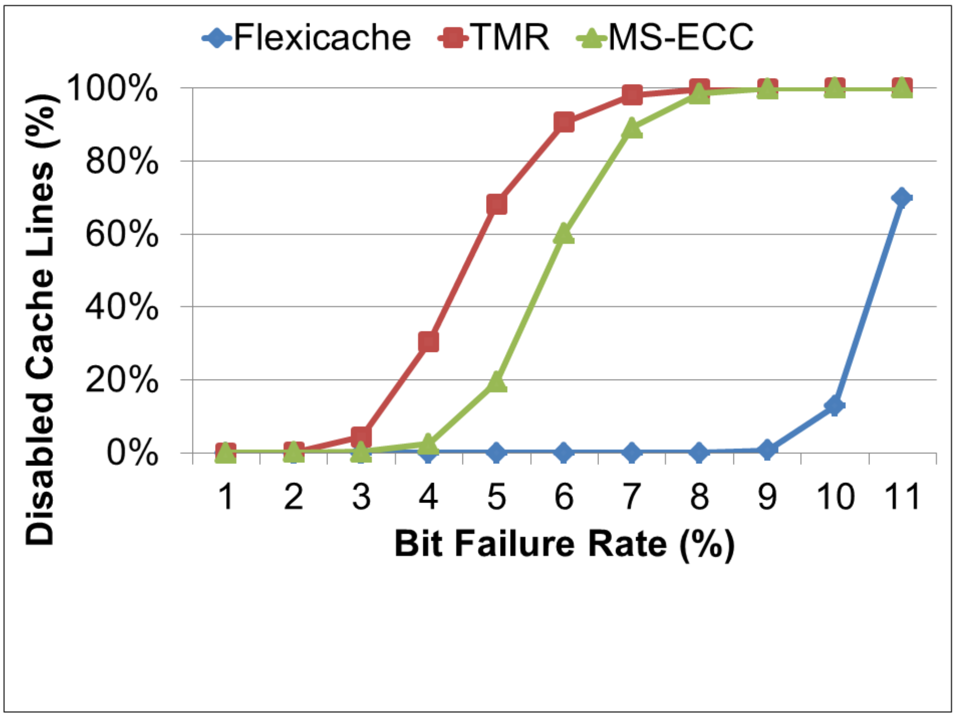


Figure 7.22: The figure presents the percent of disabled lines in very-high error rate. Flexicache presents a uniform cache view until 9% bit failure rate.

Note that error detection capability and error correction capability of MS-ECC are identical since OLSC intends to produce the correct data without trying to detect if there was a fault or not. In SVM, Flexicache can not correct faults, but it can detect half of the injected faults (i.e. when the size of the fault is odd). In DVM, it can correct half of the injected faults since it uses parity for the error correction while it can detect more than 90% of the injected faults. TVM can provide more than 90% error correction capability until V_{dd} is 400mV. When V_{dd} is 320mV, only TVM can provide useful cache capacity. At this point, it can detect 58% of the injected non-persistent faults and can correct half of the injected faults. In this study, we switch from SVM to DVM when the V_{dd} is 600mV. One can decide to utilize DVM for higher V_{dd}s for reliability critical applications or systems in faulty environments in order to provide higher reliability with the cost of useful cache capacity.

Uniform View of the Cache

Both Flexicahe and MS-ECC disable a whole cache line when none of the 4 ways can be correctable in the line. However, when a cache line is disabled, the uniformity of the cache is lost and the system should be notified about it. In Figure 7.22, we present the percentage of the disabled lines in each scheme according to bit failure rate in the low-power mode. According to the figure, Flexicache can present a uniform view to the system until the error rate is 9% since it couples lines instead of ways (i.e. couples horizontally instead of vertically) and thus it has comparatively more cache ways available for a given line.

Area Overhead

Figure 7.23 shows the layouts of one sub-bank [148] for both Flexicache and typical cache arrays; the second symmetric sub-bank is omitted. After adding parity bits, parity calculators, extra slices, XORs, majority voters, buffers and peripheral circuits, Flexicache presents 12% area overhead compared to the typical cache without any protection. Note that the biggest portion of this overhead belongs to the extra slices which we add to make the cache dividable by there, therefore, actually increasing the size of the cache. This layout allows Flexicache dynamically switch between SVM, DVM and TVM which provides maximum 100%, 50% and 33% useful cache capacity as we presented in Figure 7.19.

7.5 Summary

Although downscaling the supply voltage provides a substantial energy saving in computer systems, when the V_{dd} is reduced lower than the safe operation margin, it causes drastic increase in the number of persistent failures especially in memory structures. Utilizing Error Correcting Codes (ECC) in the memory structures is the most appealing approach to reduce the supply voltage below the safe operating margin. However, ECC schemes presenting high encoding/decoding overhead diminishes the performance of the system. In this chapter, we propose two approaches to support that level1 cache can operate at the ultra-low voltage level. In the first approach, we utilize a fast and low-complexity SEC-MAEC code in L1 caches under scaling supply voltage. We also demonstrate how to organize 4-way cache in the architecture level in order to maximize the cache capacity. In the second approach, we present Flexicache, a novel, reliable cache design which configures itself for different supply voltages from the nominal

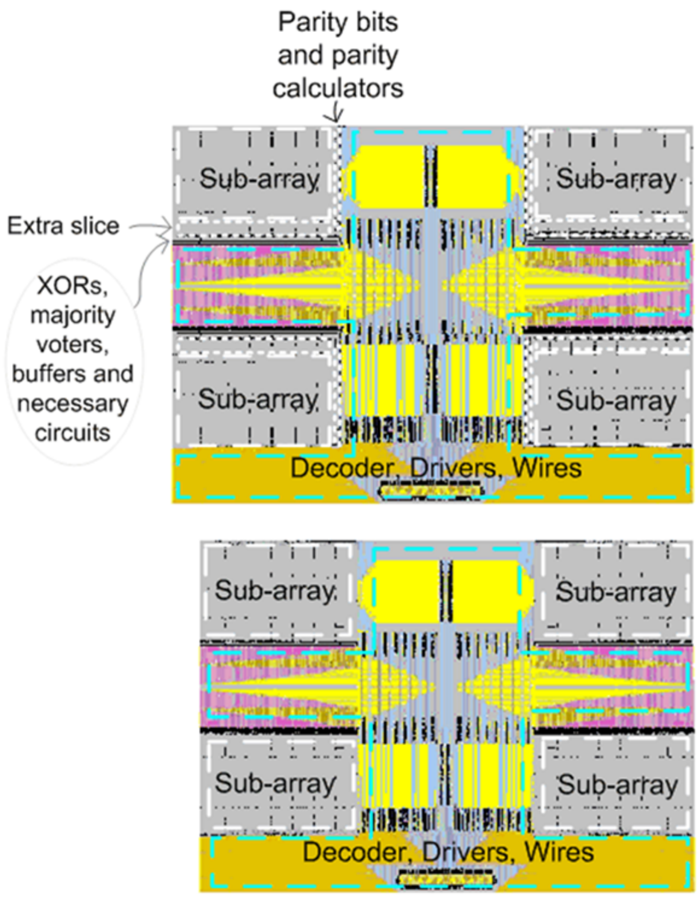


Figure 7.23: The figure shows the layout of a sub-bank and address decoder of Flexicache (top) and a typical cache (down).

to the near threshold voltage levels in order to duplicate or triplicate each data line if higher reliability is required. Flexicache can continue to operate reliably up to 10% bit failure rate. Therefore, it alters the possibility to operate in 320 mV. Compared to OLSC [122] and conventional triplication, Flexicache provides a cache with a higher capacity in low power mode with significantly lower error correction latency and with less energy consumption.

8 Increasing the Lifetime of NAND Flash Memories

NAND flash memory has been widely used as a storage medium for many systems such as laptops, PDAs, and mobile phones because of its high performance, large storage density, non-volatility and low power consumption. The per-bit cost of NAND flash memory continues to fall dramatically every year due to aggressive technology scaling and the introduction of multi-level flash cells. This allows NAND flash to be applicable for even more applications, such as solid-state disks (SSDs) for personal computers and enterprise servers.

However, the widespread adoption of flash-based storage in performance-intensive applications has led to concerns regarding the reliability and endurance of the underlying flash memories. A flash memory cell has limited endurance, i.e. data cannot be reprogrammed into the cell more than a limited number of times. A single-level flash cell (SLC) can tolerate $\sim$10k program/erase (P/E) cycles while a 2-bit multi-level cell (MLC) can only survive for $\sim$3k P/E cycles for 30- 40nm (i.e., 3x-nm) technology generations. The available P/E cycles are expected to decrease even more in the near future as flash cells continue to scale down in size and more than 2 bits are programmed per cell. Generally, storage systems have strict requirements on reliability. For example, the uncorrectable bit error rate during usage should be less than 10-15 and stored data should be available for 5-10 years [10]. Enterprise-class SSDs are expected to support at least 10 full disk writes per day for at least five years under fully random data patterns. Assuming typical write amplification of 2 times (due to additional writes caused by garbage collection and wear leveling [149]]) and ideal wear-leveling, current MLC flash based storage will use up all its reliable P/E cycles (e.g., 3000) within 5 months. It is therefore clear that flash memories cannot satisfy the lifetime requirements for enterprise SSDs, which require much longer than 5 months of lifetime.

One way to improve flash lifetime is to use stronger error correction codes (ECC) [7]. Stronger ECC detects and corrects raw bit errors that happen over the lifetime of a flash cell, thereby increasing the number of P/E cycles each cell can tolerate without exposing the raw bit errors to the user. Unfortunately, stronger ECC has two major shortcomings: (1) high implementation overhead and (2) diminishing returns on flash lifetime improvement. The latter is because the raw bit error rate increases exponentially with P/E cycles while ECC error correction capability increases less than linearly, as detailed in later sections. As such, techniques that tolerate raw bit errors in flash cells without relying on stronger ECC are desirable. In this paper, we present new techniques that achieve this and thereby allow us to utilize unreliable flash media in a reliable way at high numbers of P/E cycles.

In this section we propose 2 main schemes to increase the lifetime of flash memories:

- The first scheme is Flash Correct-and-Refresh (FCR) that periodically reads, corrects, and reprograms or remaps the stored data before it accumulates more retention errors than that can be handled by the ECC on SSD controller.
- The second scheme is Neighbor-cell Assisted Correction (NAC) in order to mitigate programming errors. During the guaranteed lifetime of flash memory, the data is read by using the *global optimum reference voltage* and the data is corrected via ECC. When ECC fails to correct data, NAC is triggered and the data is re-read by using local read reference voltages which is defined by the data values stored in the neighbor cells.

Flash memories can be equipped with the "bad block management" scheme which maps bad blocks to additional good blocks in order to increase the lifetime of the flash memory [150]. Both schemes presented in this chapter are orthogonal to the idea of extending flash disks with additional blocks, so that they increase the lifetime of the disk even further.

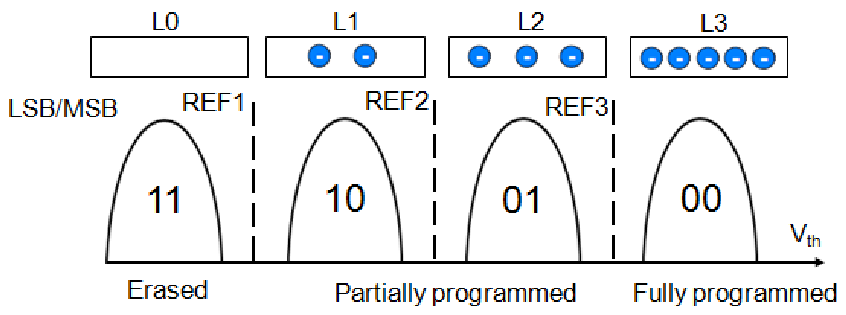


Figure 8.1: Threshold Voltage Distribution of 2-bit MLC Flash

In the next section, we present the background of flash memory operations and faults occurring in flash memories. Than we present Flash Correct-and-Refresh (FCR) and Neighbor-cell Assisted Correction (NAC) schemes in Section 8.2 and Section 8.3 respectively. We evaluate FCR and NAC in Section 8.4.

8.1 Flash Memory Background

Floating gate transistor is the atomic storage unit of NAND flash memory. Its threshold voltage can be modulated by the amount of electrons programmed on the floating gates. NAND flash memory can be of two types: single level cell (SLC) flash and multi-level cell (MLC) flash. Only one bit of information can be stored in a SLC flash cell, while multiple bits (e.g. 2-4 bits) can be stored in a MLC flash cell. For n-bit MLC NAND flash memory, the threshold voltage range of its transistors is divided into 2^n separate regions and each region represents a unique n-bit value. Figure 8.1 shows 2-bit MLC which is separated 4 regions (i.e. P_0, P_1, P_2, P_3) representing the bit values of 11, 10, 01, 00 respectively. The threshold voltage of a given cell is mainly affected by the number of electrons trapped on the floating gate. Figure 8.1 shows the bit mapping to Vth and the relative proportion of electrons on the floating gates of a 2-bit MLC flash. The bits stored in a 2-bit MLC NAND flash memory cell can be classified into most significant bit (MSB) and least significant bit (LSB), depending on the location of the bit inside the bit-string.

A NAND flash memory chip is composed of thousands of blocks. A block consists of a 2-D array of flash cells. Each row of the array forms one wordline and each column forms one bitline. The address of the wordline increases one by one from bottom to the top. Thus,

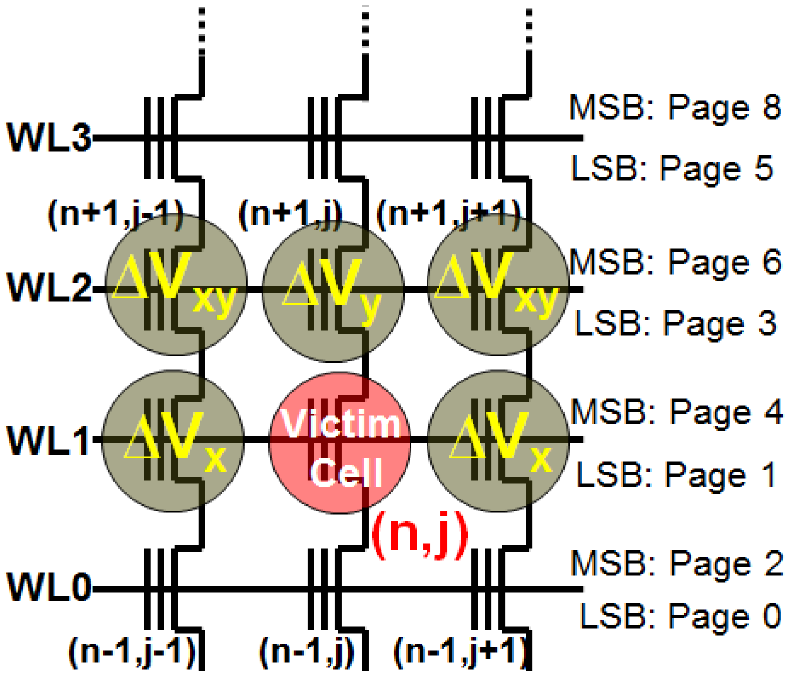


Figure 8.2: All-bit-line NAND flash block architecture

a cell location can be uniquely determined by its wordline and bitline address inside a block. For all-bit-line NAND flash memory, the MSBs of all the cells on the same wordline can be programmed and read simultaneously, which forms an MSB page. Similarly, all the LSBs of the cells on a wordline form one LSB page. Each page has its unique physical address inside a block and an example of the page address mapping inside a flash block is shown in Figure 8.2. We can see that the LSB page number on wordline n is 2n-1 while the corresponding MSB page number is 2n+2 for all-bit-line flash memory. The exceptions are the bottom wordline (i.e. wordline 0) and top wordline (i.e. wordline N) of a block. The numbers of LSB /MSB page on bottom and top wordline are 0/2 and (2N-3)/(2N-1) respectively.

The size of each page is generally between 2kB and 8kB (i.e. 16k and 64k bitlines). The stack of flash cells in the bitline direction forms one string. The string is connected to a bit line through SGD (the select gate at the drain end) and connect to the common source diffusion through SGS (the select gate at the source end).

In this section we first explain flash memory operations. Than we present how faults occur in flash memories.

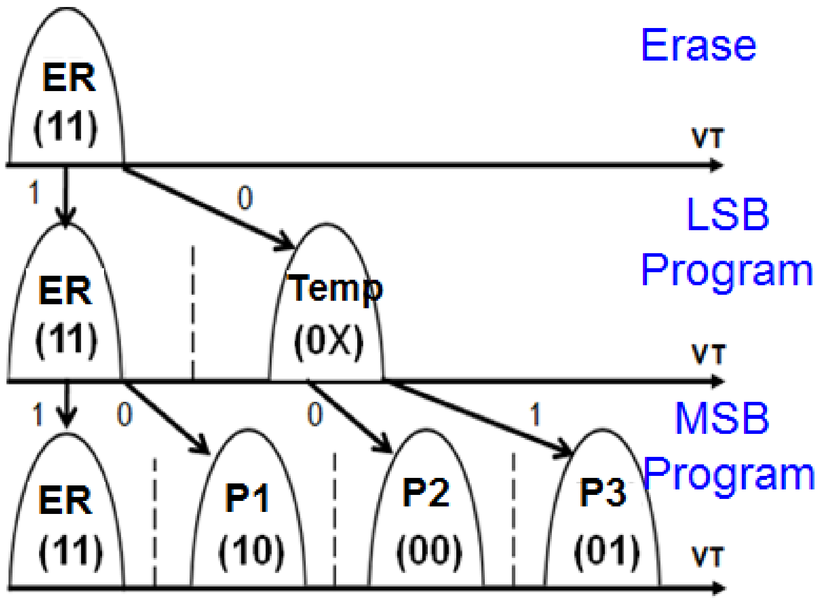


Figure 8.3: 2-bit MLC flash programming scheme. Cell states are encoded in format (LSB, MSB)

8.1.1 Flash Memory Operations

Flash memories generally support three fundamental operations as follows:

Erase Operation

During erase operation, a high positive erase voltage (e.g. 20V) is applied to the substrate of all the cells of the selected block and the electrons stored on the floating gate are tunnelled out through Fowler-Nordheim (FN) mechanisms [151]. After a successful erase operation, all charge on the floating gates is removed and all the cells are configured to P0 (11) state. Erase operation is at the granularity of one block.

Program Operation

During program operation, a high positive voltage is applied to the wordline, where the page to be programmed is located. The other pages sharing the same wordline are inhibited (from being programmed) by applying 2V to their corresponding bitlines to close SGD and boost the potential of corresponding string channel. The programming process is typically realized by incremental step pulse program (ISPP) algorithm [152]. We first provide background on ISPP.

ISPP. Before a flash cell can be programmed, the cell must be erased (i.e., all charge is removed from the floating gate, setting the threshold voltage to the lowest value). When a NAND flash memory cell is programmed, a high positive voltage applied to the control gate causes electrons to be injected into the floating gate. The threshold voltage of a NAND flash cell is programmed by injecting a precise amount of charge onto the floating gate through ISPP [152]. During ISPP, floating gates are programmed iteratively using a step-by-step program-and-verify approach. After each programming step, the flash cell threshold voltage is boosted up. Then, the threshold voltage of the programmed cells are sensed and compared to the target values. If the cell's threshold voltage level is higher than the target value, the program-and-verify iteration will stop. Otherwise the flash cells are programmed once again and more electrons are added to the floating gates to boost the threshold voltage. This program-and-verify cycle continues iteratively until all the cells' threshold voltages reach the target values. Using ISPP programming, flash memory cells can only be programmed from a state with fewer electrons to a state with more electrons and cannot be programmed in the opposite direction.

Flash memory is programmed page by page: the MSB page of a wordline is programmed at a different time from the LSB page of the same wordline. A contemporary two-bit-per-cell MLC flash cell is programmed to a desired value in two stages, as shown in Figure 8.3. After an erase operation (applied to the page the cell resides in), the cell starts out at the erased state (ER). If the LSB of the cell is programmed as 0 (during the programming of the corresponding LSB page), the flash cell moves into a temporary program state (Temp). Otherwise it remains in the ER state. During the programming of the corresponding MSB page, if bit value 1 is programmed into the cell's MSB, the flash cell either remains in the ER state or moves from the Temp state into the P2 state. If 0 is programmed into the cell's MSB, the flash cell moves either from the ER state to the P1 state or from the Temp state to the P3 state.

In general, flash memory manufacturers recommend that the pages inside a flash block be programmed sequentially in page number order (0, 1, 2, ...). With this programming policy, pages in Figure 8.2 would be programmed in the following order: page 0 (LSB of WL 0), page 1 (LSB of WL 1), page 2 (MSB of WL 0), page 3 (LSB of WL 2), page 4 (MSB of WL 1), and so on. This is called in-page-order programming. On the other hand, a flash block can be programmed without following this recommendation, a method called out-of-pageorder programming. This increases the flexibility in programming flash. This method can have many variants. For example, pages in a block can be programmed in a completely random order, or the pages in a block can be programmed in their wordline number order.

Read Operation

The read operation is also at the page granularity. The SGD, SGS and all deselected wordlines are turned on. The wordline of selected read page is biased to a series of predefined reference voltages and the cell's threshold voltage can be determined to be between the most recent two read reference voltages when the cell conducts current.

Reading a flash cell is mainly to determine the voltage region that its threshold voltage falls in. For LSB reading of 2-bit MLC flash memory, a reference voltage (e.g. REFb in Figure 8.1) is selected to compare with the threshold voltage of the flash cell. If the threshold voltage of the flash cell is larger than REFb, it will be read as 0 otherwise it will be read as 1. For MSB reading, the threshold voltage of a cell will compare to two reference voltages (e.g. REFa and REFc in Figure 8.1). If the cell's threshold voltage is within the range of [REFa, REFc], it will be read as 0 otherwise it will be read as 1. The above selected reference voltage is generally called read reference voltage. Recent flash memory [153] allows the read reference voltage to be configurable so that different reference voltages can be tried to so that a better one can be found to achieve lower error rate, which is called read-retry. Previous works [154, 155] leverage read-retry to identify the exact threshold voltage of each cell and study the threshold voltage distributions of flash memories.

8.1.2 Errors in NAND Flash Memories

Cai et al [156] classify the observed errors into four different types from the controller's point of view: Erase error, Program interference error, retention error and Read error. All types of errors are highly correlated with P/E cycles. At the beginning of the flash's lifetime, the error rate is relatively low and the raw bit error rate is below 10^{-4}, within the specified lifetime (3k cycles). As the P/E cycles increase, the error rate increases exponentially. The P/E cycle-dependence of errors can be explained by the deterioration of the tunnel oxide under cycling stress. During erase and program operations, the electric field strength across the tunnel oxide is very high (e.g., several million volts per centimeter). Such high electric field strength can lead to structural defects that trap electrons in the oxide layer. Over time, more and more defects accumulate and the insulation strength of the tunnel oxide degrades. As a result, charge can leak through the tunnel oxide and the threshold voltage of the cells can change more easily. This leads to more errors for all types of flash operations. Now, we explain these errors.

Retention error happens when the data stored in a cell changes over time. The main reason is that the charge programmed in the floating gate may dissipate gradually through the leakage current. The long-term retention errors are the most dominant; their rate is highest compared to other errors.

The retention errors are value dependent. The most common retention errors (00→01, 01→10, 01→11 and 10→11) are all cases in which Vth shifts towards the left. This is because, the electrons stored on the floating gate gradually leak away under stress induced leakage current (SILC). When the floating gate loses electrons, its Vth shifts left from the state with more electrons to the state with fewer programmed electrons. A cell in state 11 cannot shift to another state by losing electrons because there is no other state to the left of it

Retention error rates are highly dependent on retention test time. If the time before we test for retention errors is longer, the floating gate of flash memory is more likely to lose more electrons through leakage current. This eventually leads to Vth shift across Vth windows and causes errors. For example, the 3-year retention error rate is almost three orders of magnitude higher than one-day retention.

Program interference error happens when the data stored in a page changes (unintentionally) while a neighbouring page is being programmed due to parasitic capacitance-coupling [154]. Due to coupling capacitance between neighboring floating gates, the programmed threshold voltage of a cell may change when neighbor cells are programmed later. Program interference is the phenomenon in which the threshold voltage of a flash cell, called the victim cell, unintentionally changes (i.e., gets disturbed) while another cell, called the aggressor cell, is being programmed [154, 156, 157]. If the change in threshold voltage due to program interference causes the victim cell's voltage to shift to a different threshold voltage range, then the victim cell's value becomes incorrect, leading to an error when the cell is read. The program interference error rate ranks the second (after retention errors) and is usually between error rates of 1-day and 3-day retention errors.

The most dominant programming interference errors are 11→10 and 10→01. Their relative percentages are 70% and 24% respectively. Less common errors are 10→00, 11→01, and 01→00. Their relative percentages are 2.2%, 1.5% and 0.4% respectively. Similar to retention errors, program interference errors also show strong asymmetry with respect to the cell value, but they occur in the opposite direction with regard to the Vth shift. The cell states mainly shift from the states with fewer programmed electrons to the states with more electrons (i.e., from left to right in Figure 8.3). When a page is being programmed, a high positive programming voltage is applied to all the control gates on the selected wordline, including those of the cells of the other pages that share the same word-line but that are not supposed to be programmed. This high positive voltage could attract additional electrons into the floating gates of these other pages through tunneling even though such pages may have already been programmed. If there are too many electrons attracted to these gates, the Vth of disturbed cells will shift towards the right into the higher threshold window. Note that it is not observed any program interference errors in cells that are in state 00. Even if additional electrons are injected into the cells in state 00 and the Vth may shift right under programming voltage interference, the cell will hold its value because its voltage threshold window will still stay the same.

Erase error happens when an erase operation fails to reset the cells to the erased state. This is mainly due to manufacturing process variations or defects caused by trapped electrons in the tunnel oxide after stress due to repeated P/E cycles. Erase errors are the least significant among NAND flash errors and they occur often only after millions of P/E cycles, which is more than 100x times the specified lifetime of the flash memory tested in [156].

Read error happens when the data stored in a cell changes as a neighboring cell on the same string is read over and over. Read errors happen mainly due to threshold voltage shifting to the adjacent threshold voltage window. The read error rate is slightly less than 1-day retention error rate

8.2 Flash Correct-and-Refresh (FCR): Mitigating Retention Errors

As discussed previously, retention errors are the most dominant errors and are highly correlated with retention time. In order to preserve the information in memories, it is straightforward to use a memory refresh mechanism which periodically reads information from an area of computer memory and immediately rewrites the read information to the same area. Refreshing is required in semiconductor dynamic random access memory (DRAM), the most widely used type of computer memory. Thus, we propose a set of new techniques called Flash Correct-and-Refresh (FCR) that exploit the dominance and characteristics of retention errors to significantly increase NAND flash lifetime while incurring minimal overheads. FCR schemes periodically read, correct, and reprogram or remap the stored data before it accumulates more retention errors than that can be handled by the ECC on SSD controller. Thus, we can achieve a low Uncorrected Bit Error Rate (UBER) while still using a simple, low-overhead ECC. Two key questions related to designing a system that uses FCR techniques are: (1) how to refresh the data (remapping or reprogramming); (2) when to refresh the data (periodically or adaptively).

8.2.1 Remapping-based FCR

Unlike DRAM cells, which can be refreshed in-place, flash cells generally must first be erased before they can be programmed. To remove the slow erase operation from the critical path of write operations, current wear leveling algorithms remap the data to another physical location rather than erasing the data and then programming in-place. The flash controller maintains a list of free blocks that have been erased in background through garbage collection and are ready for programming. Whenever a write operation is requested, the controller's wear leveling algorithm selects a free block and programs it directly, remapping the logical block address to the new physical block. The key idea of FCR is to leverage the existing wearleveling mechanisms to read, correct, and remap to a different physical location each valid flash block in order to prevent it from accumulating too many retention errors. Operational flow of FCR consists of four main steps:

(1) During each refresh interval, a block with valid data that needs to be refreshed is selected;

(2) The valid data in the selected block is read out page by page and moved to the SSD controller;

(3) The ECC engine in the SSD controller corrects all the errors in the read data, including retention errors that have accumulated since the last refresh. After ECC, the data are error free;

(4) A new free block is selected and the error free data are programmed to the new location, and the logical address is remapped.

Note that the proposed address remapping techniques can leverage existing hardware and software of contemporary wear leveling and garbage collection algorithms of the flash based SSD controller.

The error rate can almost be linearly decreased if we keep decreasing the period and increase the number of times for data regeneration over data storage time. As the remapping based FCR can greatly decrease the raw BER from the flash media and it has two benefits: 1) Given fixed lifetime increase requirement of flash-based SSD, the ECC engine on SSD controller can be greatly simplified. 2) Given fixed ECC on flash SSD controller, FCR can greatly increase the maximum number of P/E cycles that flash memory can tolerate while keeping the qualified storage reliability requirements for host applications.

However, periodic FCR scheme will introduce additional erase operation overhead. This is because after the flash data are corrected and remapped to the new destination block, the old data in the original block will be marked as outdated. Thus, the block will eventually be erased and reclaimed by garbage collection. The more frequent FCR, the more additional erase operation will be introduced. As we know that flash can only tolerate limited number of P/E cycles, each erase operation will make the flash block worn out and come closer to the end of life. There might be a cross point that over frequent refresh will use up more P/E cycle than the P/E cycle gained by FCR techniques. In the next section, we propose a new technique that can reduce erase operations by leveraging the intrinsic properties of retention errors and the physics of ISPP program operation of contemporary NAND flash.

8.2.2 Hybrid FCR

To reduce the overheads associated with remapping, we propose a hybrid data correct and reprogramming/remapping-based flash refresh scheme that can perform in-place refresh most of the time, without a preceding erase operation and can greatly reduce the overhead. This in-place refresh takes advantage of the fact that retention errors arise from the loss of electrons on the floating gate over time and the flash cell with retention errors can be reprogrammed to its original correct value without erase operation under contemporary ISPP programming scheme.

When a NAND flash memory cell is programmed, a high positive voltage applied to the control gate causes electrons to be injected into the floating gate through Fowler-Nordheim mechanisms [151]. Thus, NAND flash memory cells can only be programmed from a state with fewer electrons to a state with more electrons step by step with ISPP programming and cannot be programmed in the opposite direction. However, retention errors are caused by the loss of electrons from the floating gate overtime and cells with retention errors change from a state with more electrons to fewer electrons. Thus, cells with retention errors can be reprogrammed to the value before retention by re-charging additional electrons step by step through ISPP without additional erase operations.

The in-place reprogramming mechanism works as follows. First, we select a block to be refreshed and read the data out of the memory page by page into the flash controller. By selecting suitable refresh interval, the total error number is below the correction capability of the ECC and the controller can fully correct all the errors in the read data. Then we can re-program the flash cells in the same location using the read data after error correction (hence in-place reprogram) without erasing the whole block. For in-place reprogramming, there are three possible cases: 1) If the new corrected value corresponds to the state with less charge than that of the old erroneous value, we will fail the programming and the cell will remain on its error state. 2) If the new corrected value corresponds to the state with more charge than the old erroneous value, then the final programmed value on the cell will be the new corrected value and the error will be corrected. Recall that this is because the flash cells can only be programmed from the state of fewer electrons to the state with more electrons. Note that, the location of the cell can be uniquely determined by its wordline and bitline address inside a block and the cell can be re-programmed with additional electrons. 3) If the corrected value is exactly the same as the original value, the in-place reprogramming will not change the stored data value, as ISPP will stop programming the cell as soon as it detects that the target value has already reached. Note that most of the cells are reprogrammed with the exactly the same data as error rates are generally below 1%. So the re-program interference to the neighbor pages will be low as only few cells are really programmed with additional electrons into the floating gates during in-place reprogramming.

A potential issue with in-place reprogramming schemes is that the accumulation of programming errors across the multiple continuous reprogramming of the same flash block. This is because in-place reprogram is suitable for correcting retention errors but not program errors due to the fact that the in-place program can only add more electrons into the floating gate instead of removing. However, the retention error rates are two orders of magnitude higher than the program error rate. So, only in cases of high reprogramming frequency, will the programming error rate become comparable to that of the retention errors. If the program error counts reach the error correction capability of ECC, it is highly probably that the data read in the next refresh interval can no longer be recovered as the sum of current accumulated program errors and newly produced retention errors may exceed the error correction capability of ECC on the SSD controller.

To mitigate the accumulated reprogramming errors issue, we propose a hybrid reprogramming/remapping refresh technique to control the number of consecutive in-place programming to the same location. The main idea of this proposed technique is to remap the corrected data to a new free block to eliminate all errors when the accumulated right shift program error count is too high. The workflow of hybrid refresh is as follows:

1) Select the flash block that have data need to be refreshed at each reprogram period and read the data from that block page by page.

2) SSD controller buffers the LSB page and MSB page of the same word-line and combines the bits of the location of LSB and MSB pages to identify the threshold voltage state of the cells on the selected word-line.

3) The ECC engine corrects the two page data respectively and combines the bits of corrected page as a pair to form the cell states to be corrected to.

4) The corrected state pair is then compared with the programmed states before correction to identify the percentage of right shift errors and left shift errors. If the right shift error count is less than a certain threshold level (i.e. 30% of the maximum number of errors that could be corrected by ECC, which is conservative), then we still trigger in-place reprogramming as the sum of the remaining program errors plus the maximum retention errors before the next refresh interval would still be within the error correction capability of ECC. Otherwise, remap-based data refresh needs to be triggered. A new free block should be allocated and be programmed to save the corrected data page by page. Note that after a remapping-based refresh, the existing retention errors and the accumulated right shift program errors are eliminated. Then, the data in the newly allocated block can be refreshed using in-place refresh for the next few refresh periods until accumulated program errors are too high once again and another re-mapping based refresh need to be triggered. With such hybrid refresh technique, the additional erase operation can be greatly reduced compared to remap based refresh, as only one erase operation is needed after every N-times in-place reprogramming.

8.2.3 Adaptive-Rate FCR

It is observed that the rate of retention errors is very low during the beginning of flash lifetime (i.e. nominal lifetime) [156]. Until more than 1000 P/E cycles, the retention error rate is lower than the acceptable raw Bit Error Rate (BER) that can be corrected by the simplest BCH code. Hence, at the beginning of its lifetime, flash memory does not need to be refreshed. Retention error rate increases as the number of P/E cycles increases. We leverage this key observation to reduce the number of unnecessary refresh operations. We further propose dynamic adaptive-

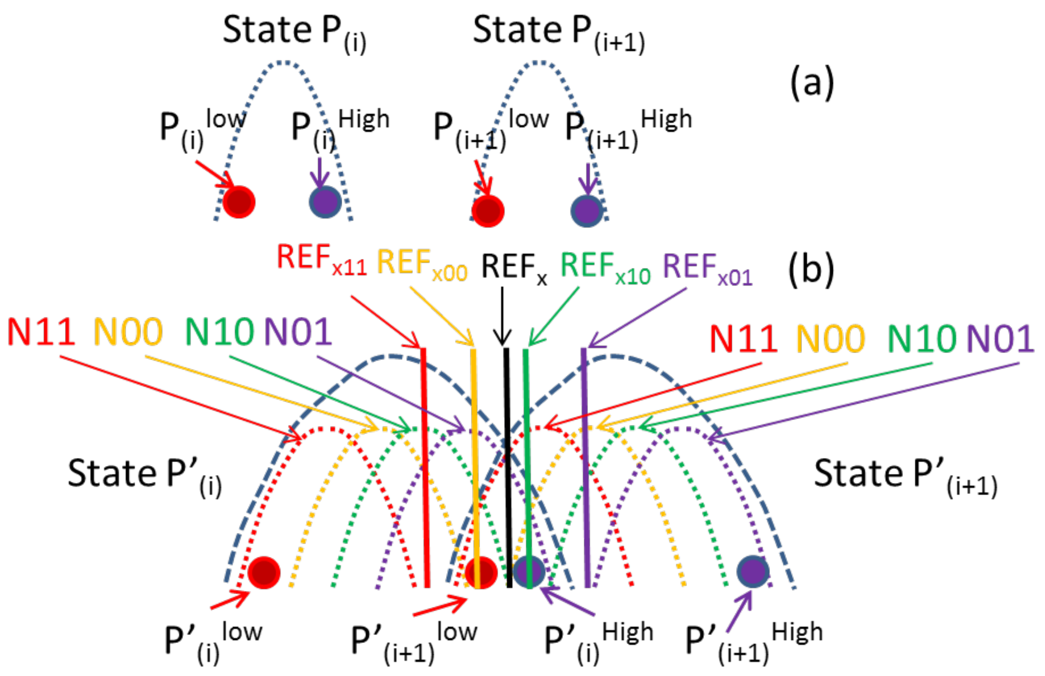


Figure 8.4: Global and Local Read Reference Voltages

rate refresh techniques to improve flash memory lifetime while eliminating unnecessary refresh operations. The main idea of adaptive-rate FCR is to adapt the refresh rate to the number of P/E cycles a block has incurred. Initially, refresh rate for a block starts out at zero (no refresh). Once ECC becomes incapable of correcting retention errors, the block's refresh rate increases to tolerate the increased retention error rate. Hence, refresh rate is gradually increased over each flash block's lifetime to adapt to the increased P/E cycles. The whole lifetime of a flash block can be divided into intervals with different refresh rates ranging, for example, from no refresh (initially), yearly refresh, monthly refresh, weekly refresh, to daily refresh. The frequency of refresh operations at a given P/E cycle count is determined by the acceptable raw BER provided by the used ECC and the BER that corresponds to the P/E cycle count. Note that this mechanism requires keeping track of P/E cycles incurred for each block, but this information is already maintained to implement current wear-leveling algorithms.

8.3 Neighbor-cell Assisted Correction: Mitigating Programming Errors

In this section, we present Neighbor-Cell Assisted Correction (NAC) that exploits the characteristics of programming errors to extend the lifetime of NAND flashes. As the P/E cycles increase, the rate of the programming errors increases exponentially. Binary BCH codes [7] which is widely deployed in Flash memories is mostly adequate to correct the programming errors in the nominal lifetime. We utilize NAC scheme in the extended lifetime of the Flash memory when ECC fails to correct programming errors.

In order to explain principles of NAC, we define two reference voltages to read flash memory: (1) global optimum read reference voltage, (2) local optimum read reference voltage. We define the global read reference voltage between neighboring state P_i and P_{i+1} as the mean value of $REF_x = (\mu^{P(i)} + \mu^{P(i+1)})/2$ (See Figure 8.4 for μ of each P_n in 2-bit MLC) . Similarly, we define the local read reference voltage between neighboring state P_i and P_{i+1} when the data read from the direct aggressor cell is ab as $REFxab = (\mu_{Xab}^{P(i)} + \mu_{Xab}^{P(i+1)})/2$ the mean values of conditional distribution REFx11 REFx00, REFx01 and REFx10 to read conditional distribution x11, x00, x01 and x10 respectively. In Figure 8.4 we show the local and global reference voltages for 2-bit MLC. In this way, we define four local read reference voltages as REFx11, In order to measure the minimum Bit Error Rate, we program random data into 2Y-nm flash devices. Then, we read out the data by using the global read reference voltages and local read references. We compare the read out data from the cells with the originally programmed data to count the number of errors. according to our experimental results, reading with local optimum read reference voltage for each conditional distribution can reduce raw BER by 86% on average compared to reading with global optimum read reference voltage.

This result directly motivates us to classify the cells in each wordline into N types depending on the values of their direct aggressor cell. For the cells of each type, we can use its local optimum read reference voltage to read and we can get approximately 1/N-th of the whole page each time. We can repeat the above reading by N times for all types of cells. Eventually, we can combine the read values of the N steps together as a complete page. Thus, we can achieve lower raw BER. This needs to classify the cells according to their direct neighbor aggressor cell and we call this technique as neighbor-cell assisted reading (NAR). NAR may significantly degrade the system read performance. To read the data out, reading with global optimum read

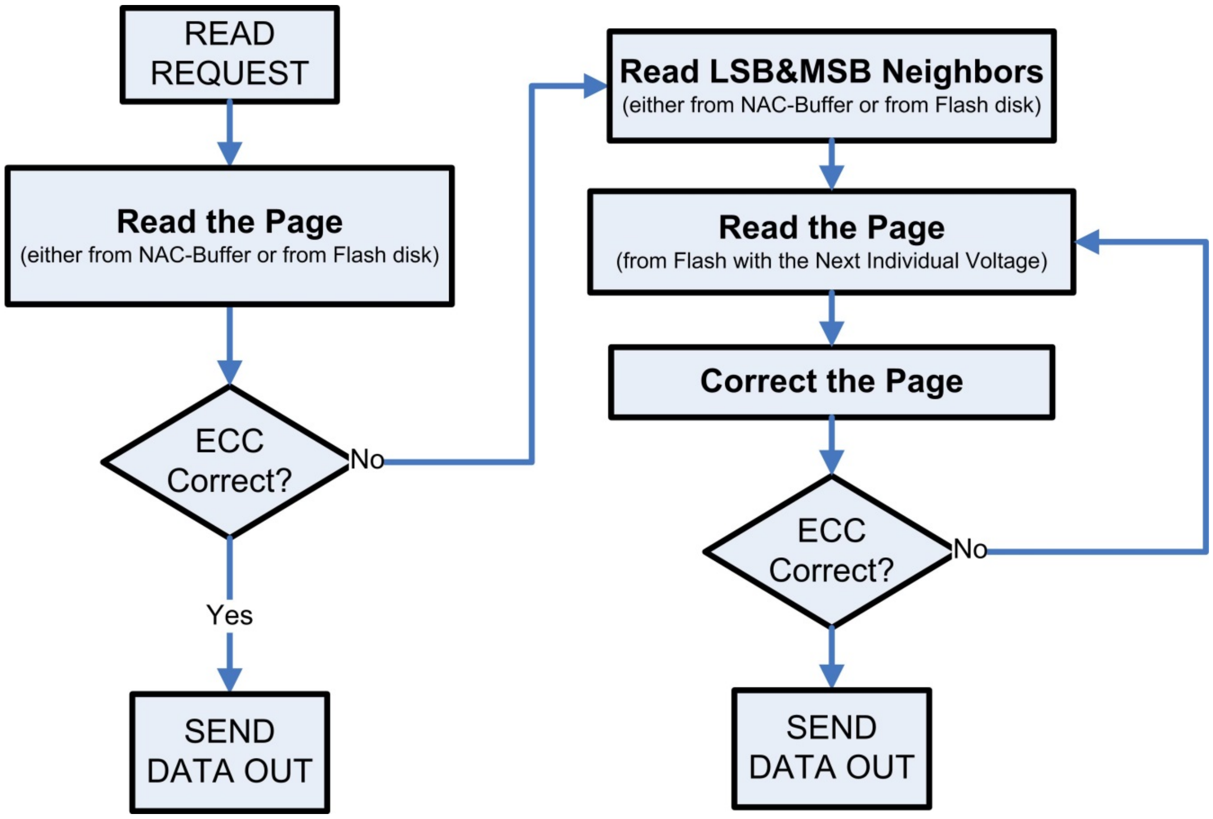


Figure 8.5: Flow of the Neighbor Assisted Error Correction

reference voltage only needs one read operation. However, NAR first needs to read both the LSB page and MSB page of the direct neighbor aggressor cells. After that, NAR needs to read the selected page for N times using different local optimum read reference voltage for N types flash memory cells. The read latency degradation is 1/(N+2). For 2-bit MLC all-bit-line flash memory (N=4), the performance will be degraded by up to 83.3%, which can significantly slow down the system reading. To mitigate such performance degradation with NAR, we propose to first read with global optimum read reference voltage, which only needs one read operation. Only when ECC fails to correct such data, NAR will be triggered. Thus, the average read latency is degraded to 1/(1+Pfail (N+2)). Here Pfail is ECC failure rate to correct the data reading with global optimum reference voltage. If ECC failure rate is low, the average read latency is still low. We call this optimized technique neighbor-cell assisted correction (NAC).

8.3.1 Flow to Trigger NAC

The detailed flow to trigger NAC is shown in Figure 8.5. We allocated a small portion of the buffer in the flash controller as the NAC-buffer for saving current data and its neighbors. When flash controller starts a read operation, it first checks whether the required data is in the NAC-buffer or not before requesting the page from the flash. If not, NAND flash interface will fetch the requested data and send it to the ECC engine. If ECC successfully corrects all the errors, flash controller sends the corrected data to the target destination, such as a host machine, and also puts it into the NAC buffer. If the ECC fails, neighbor data is required for further correction with NAC. If the requested page is in the NAC-buffer and it is error free, the data can be forwarded from the buffer to the host machine without reading the flash memory. Such data forwarding is much faster than real flash memory reading, which has a longer latency including reading data and ECC error correction. If the data is in the buffer but its ECC correction fails, which is indicated by a flag of failure for the last ECC decoding, normal ECC processing cannot recover the data and neighbor data is needed for further correction with NAC. In all-bit-line NAND flash memory, NAC requires only the LSB and MSB pages in the neighbor wordline directly above current requested wordline. Thus, when ECC fails to correct the page, unless neighbor LSB and MSB pages are in the NAC-buffer, flash controller will read the neighbor pages out from flash memory. In the meantime, a status tag will indicate whether ECC successfully corrected the neighbor data or not. After saving the neighbor pages to the NAC-buffer either error-free or erroneously, each stored page in the NAC-buffer is assigned one tag to indicate ECC decoding status. At this point, NAC is ready to correct failed data. The optimum size of the NAC-buffer is a key question here. In order to determine the optimum size, we assume the worst case scenario in which all the pages require NAC correction including their neighbor pages. It is common that pages are read from the flash memory in the page order due to large data access or the sequential reading applications. For instance, as in Figure 8.2, when Page 3 is failed and corrected by NAC, Page 5 and Page 8 are saved to NAC-buffer. Page 5 and 8 are also the neighbors of Page 6, hence, we prefer buffering those pages until Page 6 is read. However, when Page 5 is read between Page 3 and Page 6, 2 more neighbor pages are added to the NAC-buffer (i.e. Page 7 and Page 10). Therefore, we set the optimum size of the NAC-buffer as 5 pages to be able to buffer Page 5, 6, 7, 8, 10 at the same time. Trivially, when we add a new page to the NAC-buffer, if there is no free space, we evict the page with the lowest page number. Also, if there is a page which has a different block id than the added page, we rather eliminate that page instead of eliminating a page with the same block id. In this case, pages with the highest page numbers would not stack in the NAC-buffer.

In Figure 8.6, a series of examples are shown on how to use NAC buffer. Note that we only present the examples in which one more local optimum read reference is enough to correct a page with NAC.

Case 1: If the required page happens to be in the NAC-buffer with no ECC failure, it is no longer necessary to read the flash memory. The buffered data is forwarded and zero reads are required as in Figure 8.6 (a).

Case-2: If the required page is read from the flash memory correctly, it is saved to NAC-buffer without reading its neighbors. Only one read is needed as in Figure 8.6 (b).

Case-3: If the ECC correction of a page fails and this page shares the same wordline with a page which has been corrected by NAC previously, the neighbor pages of the failing page had already been fetched to the NAC-buffer. It is no longer necessary to read the neighbor pages again. Only two reads are needed as in Figure 8.6 (c); one for reading the page with the local optimum reference voltage and one for reading with the global read reference voltage.

Case-4: If the required page happens to be in the NAC-buffer but ECC fails to correct the page (e.g. in the case that the page was the neighbor of a page which previously corrected by NAC), it is no longer necessary to read the page with global read reference voltage again. The page is read from the flash disk with local read reference voltage, and its neighbor LSB and MSB pages are also loaded in NAC-buffer as Figure 8.6 (d). Three reads are required unless any neighbor was buffered.

Case-5: If the ECC correction fails for a page and if neither the page nor its neighbors are not in the NAC-buffer, four reads are needed as in Figure 8.6 (e) for reading the page with local read reference voltage in addition to reading the page and its neighbors with global read reference voltage.

8.3.2 Prioritized NAC

The errors can be classified into eight types when read with global optimum read reference voltage REFx (see in Figure 8.4). The errors for cells written in lower state (e.g. Pi state) with direct aggressor cell in value 11, 10, 00 and 01 but misread as higher state (Pi+1 state) are denoted as Pi(11)→Pi+1, Pi(10)→Pi+1, Pi(00)→Pi+1, Pi(01)→Pi+1. Similarly, the errors for cells written in higher state (Pi+1 state) with direct aggressor cell in value 11, 10,

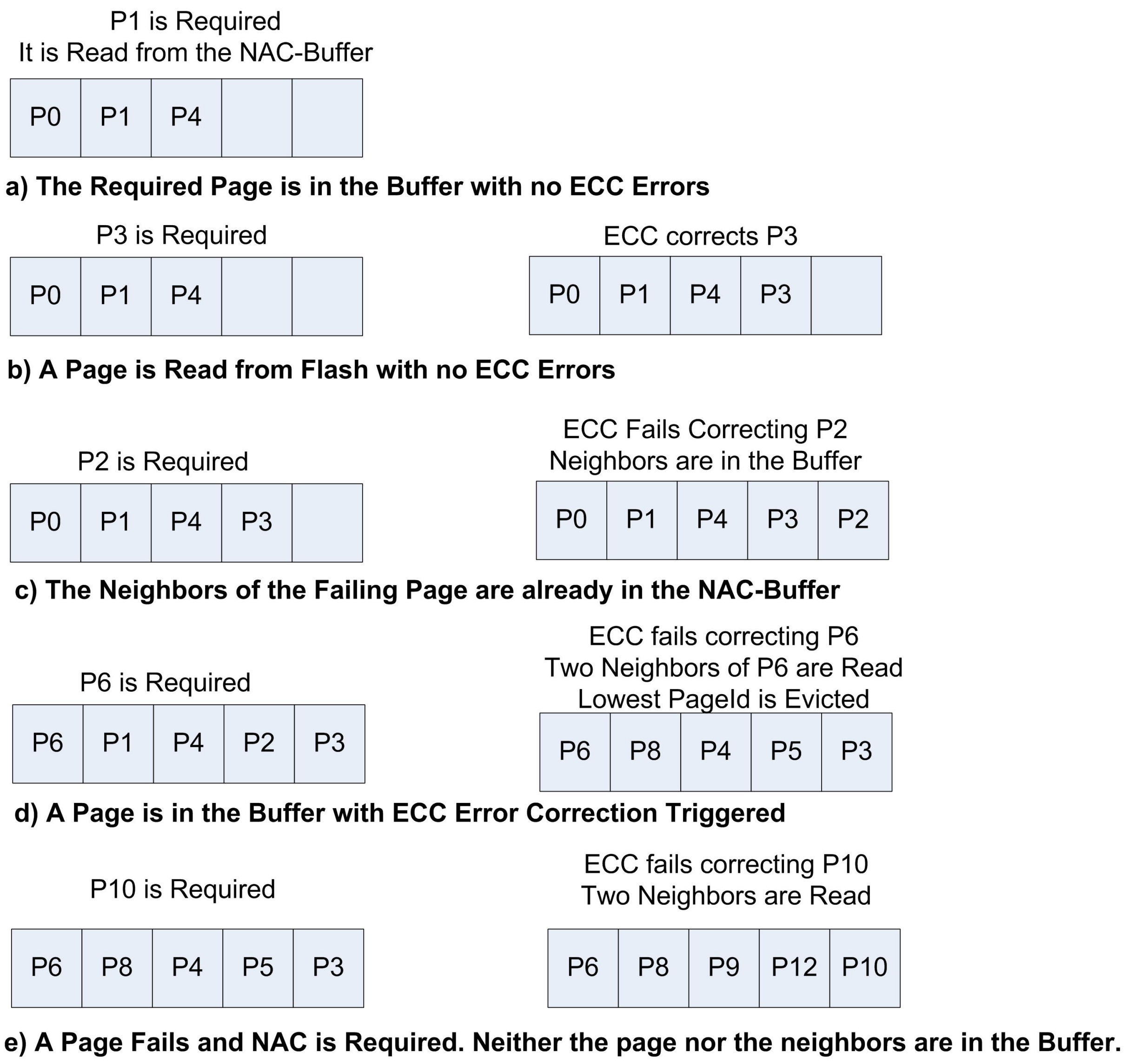

P1 is Required
It is Read from the NAC-Buffer
P0
P1
P4
a) The Required Page is in the Buffer with no ECC Errors
P3 is Required
P0
P1
P4
ECC corrects P3
P0
P1
P4
P3
b) A Page is Read from Flash with no ECC Errors
P2 is Required
P0
P1
P4
P3
ECC Fails Correcting P2
Neighbors are in the Buffer
P0
P1
P4
P3
P2
c) The Neighbors of the Failing Page are already in the NAC-Buffer
P6 is Required
P6
P1
P4
P2
P3
ECC fails correcting P6
Two Neighbors of P6 are Read
Lowest PageId is Evicted
P6
P8
P4
P5
P3
d) A Page is in the Buffer with ECC Error Correction Triggered
P10 is Required
P6
P8
P4
P5
P3
ECC fails correcting P10
Two Neighbors are Read
P6
P8
P9
P12
P10
e) A Page Fails and NAC is Required. Neither the page nor the neighbors are in the Buffer.

Figure 8.6: Examples of the NAC-buffer management when Neighbor Assisted Correction is in use.

00 and 01 but misread as lower state (Pi state) are Pi+1(11)→Pi, Pi+1(10)→Pi, Pi+1(00)→Pi and Pi+1(01)→Pi. Pi+1(11)→Pi, Pi(10)→Pi+1 and Pi(01)→Pi+1 are the three dominant errors. The reason of such dominance is that cells with threshold voltage near the borderline of neighboring distribution (read reference voltage) tend to have errors. The threshold voltage distributions of victim cells before and after neighbor cells are programmed are illustrated in Figure 8.4(a) and 8.4(b) respectively. Victim cell with aggressor cell programmed in ER state (11) has less positive additive interference. In Figure 8.4, the cells at the bottom region of higher state P(i+1) (denoted as P(i+1)low) before neighbors are programmed tend to stay at the bottom region of type N11((denoted as P'(i+1)low)) after neighbor aggressor cells are programmed to 11, which introduces the least additive program interference. On the other hand, cells at the top region of lower state P(i) (denoted as P(i)high) before interference tend to stay at the top region of type N01((denoted as P'(i)high)) if neighbor aggressor cells are programmed to 01, which introduces the largest additive program interference. It is possible that the threshold voltage of cells originally programmed in lower state but receive higher interference become larger than the cells originally programmed in higher state but receive less interference after different neighbor program interference as illustrated in Figure 8.4(b). Thus, cells with threshold voltage at the bottom and top region of a state may distort into the lower neighbor state and higher neighbor state respectively and cause failures when optimum read reference voltage REFx is applied. Another finding is that Pi+1(11)→Pi error is the largest one. This is because the program interference of aggressor cells in 10 and 01 are close (N01 and N10 overlap in Figure 8.4). Then more cells locate at the top region of each state of overall distribution. The global optimum read reference voltage deviate toward higher state than lower state and thus more cells of N11 tend to be misread as lower state.

The above discoveries can help reduce NAC performance overhead by starting the correction step from the faults that dominate. After the data read with global optimum reference voltage fail to be corrected by ECC, we prioritize to select REFx11 as the local optimum reference voltage to read the failed page to fix the dominant Pi+1(11)→Pi error. The read data of cells with direct aggressor cell in ER state (11) during global optimum reading will be replaced by the data value obtained with the new local optimum reading REFx11. For cells of type-N11 with threshold voltage larger than REFx or smaller than REFx11, the value read with REFx and REFx11 are exactly the same. For cells of type-N11 with threshold voltage within the range of [REFx11, REFx], the value read with global and local optimum read reference voltage are different. The cells of type-N11 in Pi+1 state with threshold voltage in [REFx11, REFx] was originally misread with REFx can now be fixed correctly. The cells of type-N11 in Pi state with threshold voltage in the same region was originally correctly read with REFx but will be now be fixed in a wrong way using REFx11. Since REFx11 is the optimum read reference voltage for cells in type-N11, the number of corrected cells is larger than that of mis-corrected cells. Since the read data of cells with direct aggressor cell in other state are unchanged, the total number of errors of the data after fixing with local optimum reading REFx11 will be significantly reduced even with just one time fixing. Since a large percentage of raw errors have been corrected, the number of remaining errors may be within the error correction capability of ECC. Thus, ECC may be able to correct the remaining errors directly without further fixing the other type of errors by reading with other local optimum read reference voltages (e.g. REFx01 and REFx10). In such case, system performance can be improved. If ECC still fails, we can apply the similar methodology to read with REFx10 and REFx01 respectively to further fix the errors read with global optimum reference voltage REFx.

8.4 Evaluation of FCR and NAC

We use Disksim [158] with SSD extensions [159] to quantitatively evaluate FCR and NAC. Each scheme is simulated using various real workload traces: iozone [160], cello99 [161], oltp, postmark [162], MSR-Cambridge [163] and a web search engine [164]. We present the details of evaluation traces in Table 8.1.

We configure the simulated flash-based SSD with four channels. Each channel has 8 flash chips. Each flash chip has 8192 blocks containing 128 pages. The page size is 8KB. The total storage capacity is 256GB. The energy of flash read, program, and erase operations are collected from an experimental flash memory platform [165], and are used in the simulation infrastructure to obtain the overall energy consumption.

8.4.1 Evaluation of FCR

In this section, we evaluate the lifetime improvement of our FCR mechanisms. Note that, we presented the lifetime in days by calculating the days required in each benchmark to consume the limited number of P/E cycles. We first will compare and evaluate the remapping based FCR over no refresh techniques, which only relies the ECC engine on SSD controller to recover all the possible errors in the flash media, including the retention errors generated overtime. Then,

Cello99: This trace of typical researcher activity (i.e. simulations, compilation etc.) from the HP Storage Research Lab is collected from January 14 to December 31, 1999.The trace from the first month is used in our experiments. It features 62% writes while the read request sizes are relatively small (mostly 8KB).
Postmark: In this trace, randomly generated files between the sizes 1 ⟍ 2KB to 10KB are read/written randomly. When a file is to be read, a randomly selected file is opened, and the entire file is read into memory. Appending data to a file opens a random file, seeks to its current end, and writes a random amount of data. In this trace, most of the read operations are 64KB or 128KB
MS-Cambridge: I/O traces of enterprise servers at Microsoft Research Cambridge. The request sizes diverge between 1 KB - 512KB
Financial OLTP: An I/O trace from OLTP applications running at two large financial institutions. Has around 50% read request. Generally features small read requests (i.e less than 8KB) with maximum request size of 64KB.
Web Search Engine: Three I/O traces from three popular search engines. It is a read intensive trace (i.e. 99%) with request sizes lower than 32KB

Table 8.1: Details of Traces

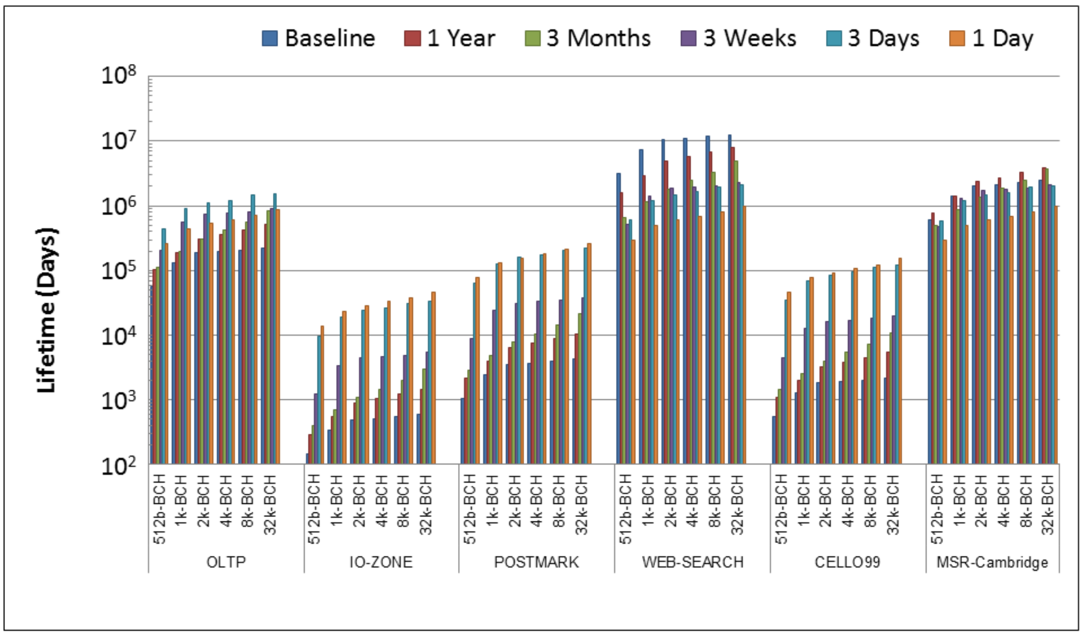


Figure 8.7: Flash lifetime (in days) provided by remapping-based FCR. The baseline represents the lifetime without any refresh (Y-axis is in log scale).

we use the remapping based FCR as baseline, and compare hybrid FCR and adaptive FCR with it.

Remapping-Based FCR: Figure 8.7 compares the lifetime provided by FCR to the baseline with no refresh. Flash lifetime is evaluated under various ECC configurations (ranging from weak 512b to strong 32k-bit BCH codes) with five refresh periods for all workloads. All P/E cycle overheads introduced by remapping are taken into account in these evaluations. First, given the same workload, stronger ECC always provides a longer lifetime than weaker ECC. For example, for IO-zone trace without refresh, lifetime can be increased by 3.5 times if 32k-bit BCH codes are used instead of 512b codes. This is because strong ECC can tolerate high raw BER at high P/E cycles and thus improves lifetime.

Second, FCR improves lifetime over both periodic FCR mechanisms for all workloads. On average, FCR provides 46.7x higher flash lifetime compared to no-refresh. We conclude that FCR is a promising mechanism for significant and consistent lifetime enhancement of flash memory.

FCR techniques can introduce additional energy consumed by refresh operations. We evaluate the energy overhead of FCR and find that it is only 1.5%. Recall that FCR starts out with no refresh and gradually increases the refresh rate up to daily refresh as the P/E cycles accumulate. Its energy overhead is significantly lower than always daily refresh.

Hybrid FCR: As we have already discussed before that given the same workload and same refresh technique, the strong ECC always outweighs weak ECC in terms of lifetime improvement. To make a fair comparison between hybrid FCR and re-mapping based FCR, we assume that the same ECC codes are applied in SSD controller for both these two technologies. In Figure 8.8, we compare the lifetime improvement of hybrid FCR versus remapping-based FCR under 512b-BCH error correction codes. From the simulation results, we get the following observations. First, hybrid FCR always has better lifetime improvement than remapping-based FCR for all workloads. This is because hybrid refresh greatly reduced the additional full disk P/E operations overhead by in-place refreshing while flash storage's endurance is hard limited by the maximum number of P/E cycles. Generally, hybrid FCR improves 3× times lifetime over remapping based FCR. Second, the benefits of hybrid FCR become more obvious for read intensive traces, i.e. web search workload. However, it is not that obvious for write intensive applications. This is because, although hybrid FCR can greatly reduce additional P/E operation cycles, the relative overhead is different for write intensive workloads and read intensive workloads. For example, OLTP have 0.14 full disk P/E operations per day on average. The iozone

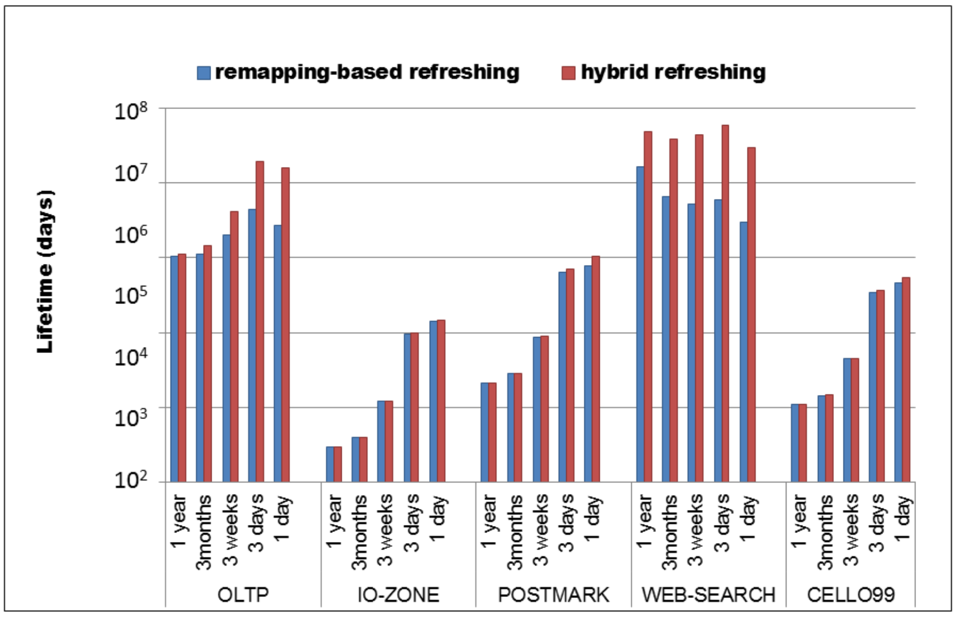


Figure 8.8: Flash lifetime (in days) provided by remapping-based FCR versus hybrid FCR (Y-axis is in log scale).

has about 20 full disk P/E operations per day. Daily re-mapping based refresh introduce one additional full disk P/E operation. If we trigger out-place replace after every 10 in-place refresh, the hybrid refresh can decrease the P/E overhead of daily refresh to 0.1 per day. It is obvious that the relative reduction of P/E cycles decreased only 5% for iozone, while up to 80% overhead decrease for oltp workload. Thus, the lifetime improvement benefits more for balanced and read intensive workload than write intensive applications. Third, the hybrid FCR starts to show positive impact on the lifetime of Web Search applications for longer period refresh (i.e. 55% longer lifetime than no refresh). For low write workload, such as MSR-cambridge, hybrid FCR increase lifetime even for short refresh period, such as daily refresh. Note that remapping based FCR decreases lifetime for both web search and MSR-cambridge workload especially for short periodic refresh.

Adaptive-Rate FCR: Figure 8.9 compares the lifetime improvement of adaptive-rate FCR with baseline no refresh technique and our proposed FCR. As adaptive does not trigger refresh periodically, we select the longest lifetime that remapping based FCR and hybrid FCR can provide among various intervals for comparison. Adaptive-rate FCR is better than both no refreshing and hybrid FCR for all the traces. This is because adaptive-rate FCR avoids the

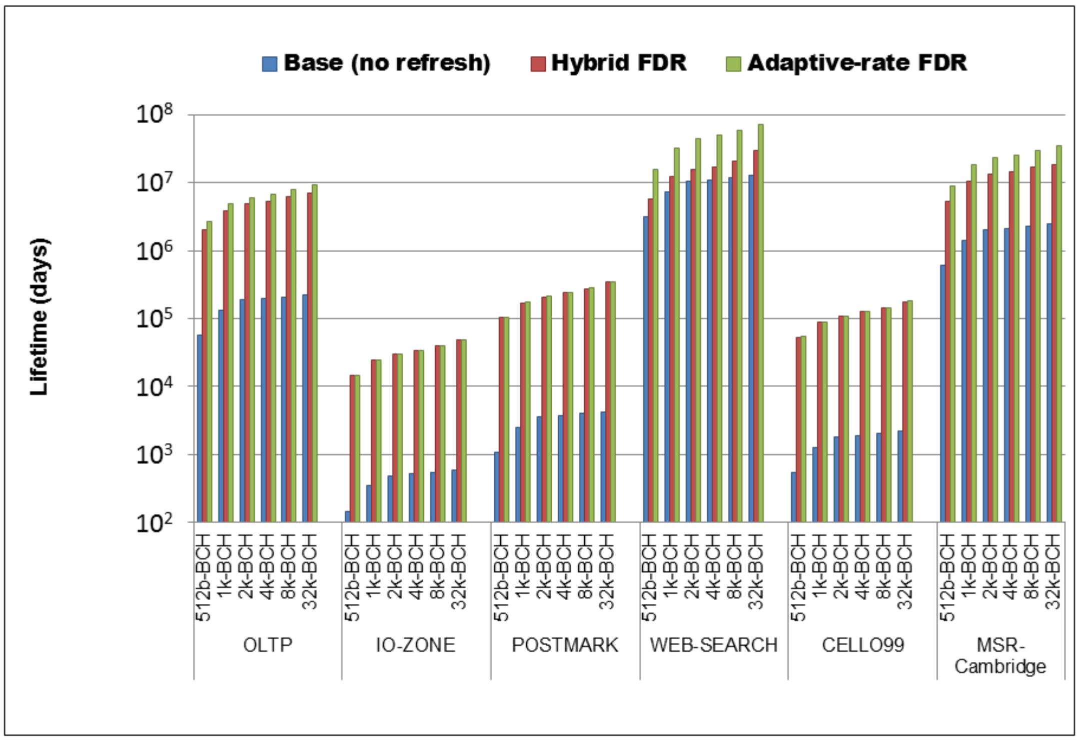


Figure 8.9: Lifetime comparison of no refresh, remapping-based FCR, hybrid FCR and adaptive-rate FCR (Y-axis is in log scale).

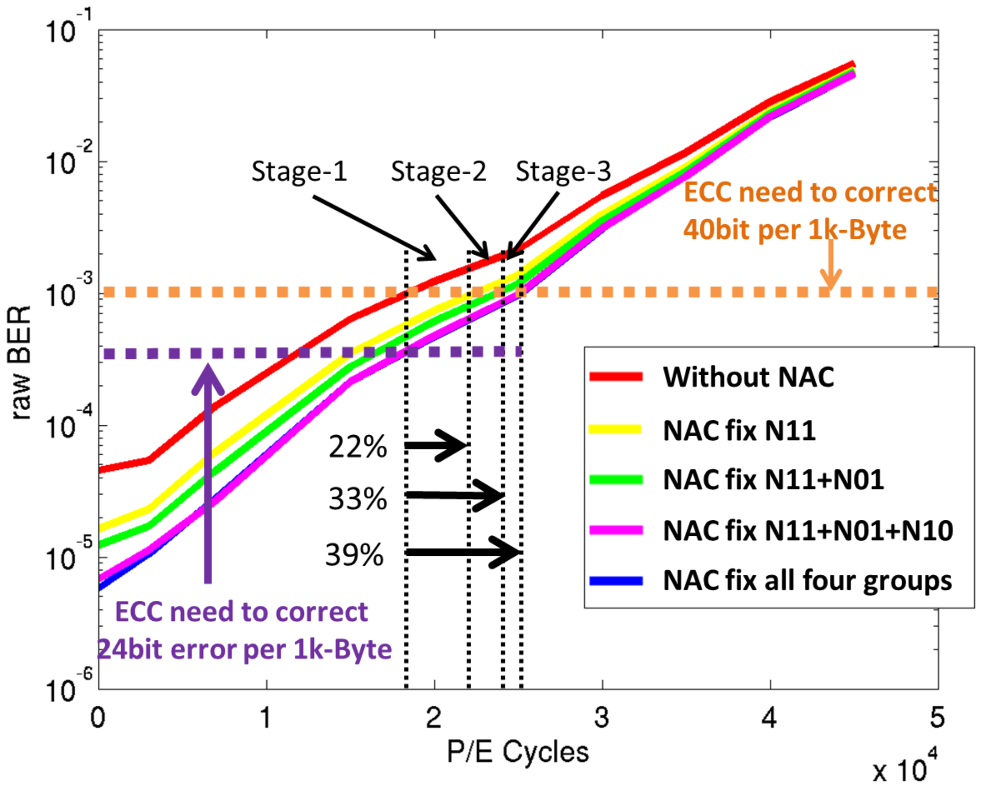


Figure 8.10: Raw bit error rate of experimental 2Y-nm NAND flash memory without NAC and with NAC using different number of bit fixings

unnecessary refreshes. Also, even read intensive applications benefit from adaptive-rate FCR since adaptive-rate FCR avoids unnecessary refreshing when the flash is in the low P/E cycle range. On average, adaptive-rate FCR provide 46.7x lifetime improvement compared to no-refresh case, 1.5x and 4.8x times lifetime improvement over hybrid refresh and remapping-based FDR.

8.4.2 Evaluation of NAC

In this section, we evaluate the lifetime improvement of our NAC mechanism.

P/E Cycle Lifetime Evaluation: We program random data into 2Y-nm 2-bit MLC flash memory up to 50k P/E cycles. Assuming that errors due to long retention can be fixed by flash refresh techniques, the programmed data are read out after one week retention at room temperature. The raw BER reading with global optimum read reference voltage without NAC and reading with NAC with one, two, three and four bit-fixing over P/E cycles are shown in Figure 8.10. To guarantee system reliability, the raw BER must be less than the acceptable raw BER (e.g. 10^{-3}) of baseline ECC. Thus, the maximum P/E cycle lifetime of the flash memory without NAC is only 18k P/E cycles as shown in Figure 8.10. Compared to without-NAC technique, the P/E cycle lifetime increases by 22% (22k P/E cycles), 33% (24k P/E cycles) and 39% (25k P/E cycles) for one bit-fixing, two bit-fixing and three bit-fixing respectively using NAC. Since the fourth bit-fixing mainly corrects errors for the conditional distribution that is in the middle of overall distribution and there are less errors in the raw BER curve over P/E cycles with three and four bit-fixing almost overlaps and there is no observable P/E cycle lifetime improvement from three bit-fixing to four bit-fixing.

NAC frequency Analysis: As can be seen in Figure 8.10, the extended lifetime due to NAC can be divided into three regions: one bit-fixing (stage 1), two bit-fixing (stage 2) and three/four bit fixing (stage 3). In the beginning of extended lifetime, the raw BER is about 10^{-3} and the ECC failure rate before NAC is 10^{-14}. NAC will seldom be triggered. As P/E cycles increase, the raw BER after reading with global optimum read reference voltage increases. Such raw BERs before NAC at the end of stage-1, stage-2 and stage-3 are 1.6×10^{-3}, 2×10^{-3} and 2.2×10^{-3} respectively. These cause the ECC failure rates before NAC to be 10^{-5}, 10^{-2} and 33% respectively. This means that ECC does not always fail during the extended lifetime region and NAC are not always triggered. We can use the ECC failure rate before NAC at the end of each stage to estimate the NAC frequency in that stage in the worst case. Note that the ECC failure rate is generally lower than that as they increase over P/E cycles. We can see that in stage-1, NAC frequency is still very low ($< 10^{-5}$). However, the ECC failure rate would not satisfy storage requirement (10^{-15}) without NAC protection. In stage-2, NAC is triggered at the rate <1%. Only in the third region, the NAC is triggered often (e.g. <33%).

Workload Analysis To be able to explain the NAC performance, we first need to analyze the behavior of read operations in the workloads. NAC presents time overhead for reading neighbor LSB and MSB pages of victim pages where ECC error-correction fails. However, we preserve these pages in the NAC buffer – in fact keeping the most recently read 5 pages - so that if these neighbor pages are requested soon after, they can be served from the NAC-buffer. In Figure 8.11, we present the ratios of the pages read in each workload whose neighbors are also requested temporally (i.e. Neighbor LSB/MSB hit ratio in the NAC-buffer). In the same figure, we also analyze if some page requests in the workloads repeat soon after, so that, these pages can be read from the NAC-buffer instead of the flash memory (i.e. victim page hit ratio). First,

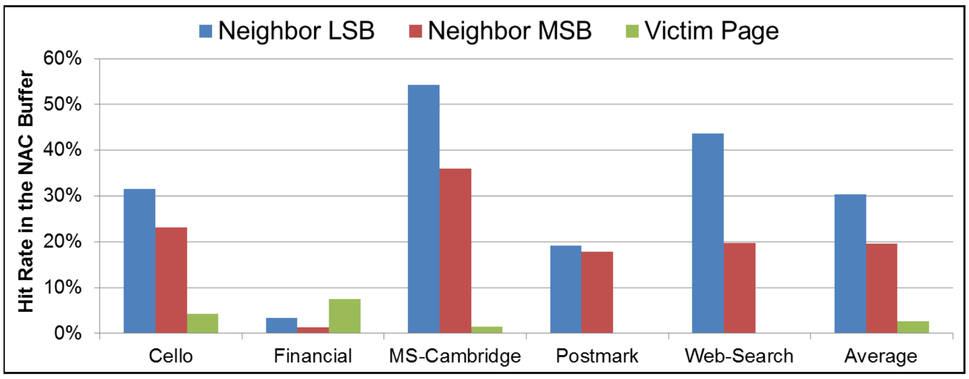


Figure 8.11: The Hit Ratio of the NAC-Buffer in Error-Free Execution (i.e., when ECC fail rate is zero

for cello, Ms-Cambridge and web-Search workloads, more than 30% of the read operations are sequential requests. For these workloads, the size of each request can be much higher than the page size therefore the read requests are mostly sequential. For instance, in MS-Cambridge, up to 512 consecutive pages are read in a single read request. Financial and postmark workloads have relatively smaller request sizes; therefore, they feature less sequential read requests when compared to other workloads. Note that the hit ratio of LSB neighbors is higher than the hit ratio of MSB neighbors. This is because the number of pages between the victim page and its MSB neighbor is higher than the number of pages between the victim page and its LSB neighbor (e.g. Page 3 and Page 6 are the LSB and MSB neighbors of Page1. While Page 3 is only 2 pages ahead in the sequential read, Page6 is 5 pages ahead.) In Figure 8.11, we also present the NAC-buffer hit ratio for the victim pages. The hit ratio of NAC-buffer is quite low (i.e. less than 5%) which indicates that only few pages are accessed repeatedly from the NAC buffer. It is because, in order to reduce the disk accesses, Operating System keeps the mostly used data in main memory instead of accessing the flash disk. Financial and Cello workloads hit the NAC-buffer a couple of times while postmark and web-search workloads have poor page locality (i.e. zero hit rates to NAC-buffer).

NAC Overhead Analysis: In Figure 8.12, we evaluate the overall performance impact of NAC for read latency at distinct P/E cycles. We activate the NAC-buffer when the first ECC failure is experienced in the flash memory. After that, each requested page is first searched in the NAC-buffer before accessing the flash disk. Note that the time spent for this search operation is around 1% of the latency of flash disk access latency (i.e. hit latency/miss penalty of NAC-buffer). We make three major observations. First, NAC does not present any perfor-

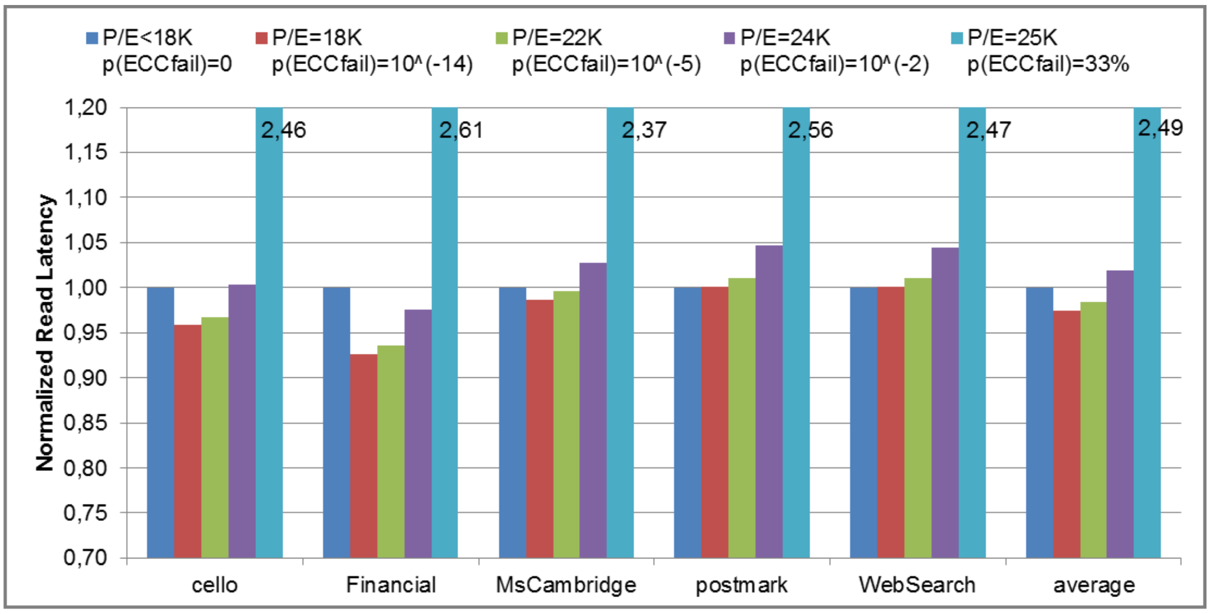


Figure 8.12: The overhead of NAC at different P/E cycles with different ECC failure rates in each P/E cycle.

mance impact when the flash is within the lifetime allowed by the base flash (i.e. less than 18K P/E cycles). Second, for the low P/E cycles in the extended lifetime, NAC presents either only negligible performance degradation or slight performance improvement. This is because, for cello, financial and Ms-Cambridge workloads, the 5-entry NAC-buffer behaves akin to a small cache in front of the flash memory. Therefore, the pages, which are requested repeatedly in a short time, are read from the NAC-buffer without accessing the flash memory (hit ratio of victim pages in Figure 8.11). Another reason is that, in real applications, the pages are commonly read in order. In other words, when a page is read, sometimes, its neighbors are also read either in the same read request or in the consecutive read requests. Thus, when NAC fetches the neighbor pages to the NAC-buffer with additional read overhead, this overhead is amortized in the following reads. The only overhead which is not amortized is incurred when the erroneous page is read again with a different read reference voltage. The third observation from the figure is that, in order to provide a 33% lifetime improvement in flash memory (i.e. from 18k to 24k P/E cycles), less than 5% performance degradation is incurred. Moreover, this performance degradation is only introduced for the higher P/E cycles in which a normal flash memory without NAC protection is considered to be non-functional. In Figure 8.12, there is a

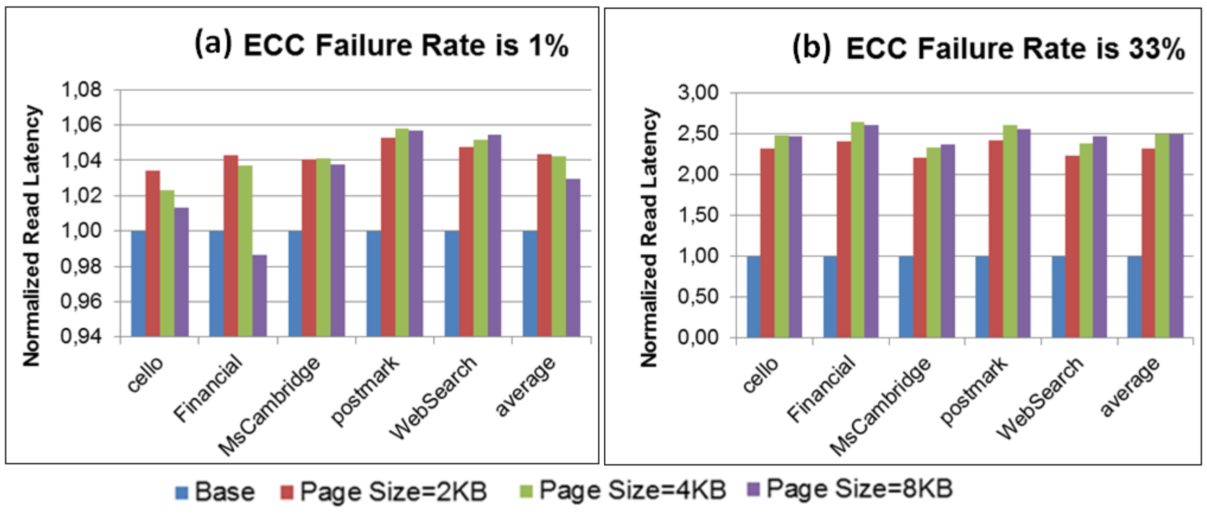


Figure 8.13: The performance overhead of NAC according to page sizes when ECC failure rate is (a) 1% at 24k P/E cycle and (b) 33% at 25k P/E cycle (Each bar is normalized to its base system).

sharp increase between 24k P/E cycles and 25P/E cycles. The main reason of this increase is the drastic increment of ECC failure rate from 10^{-2} to 33%. Hence, at 25k P/E cycles, one out of every 3 read operations requires NAC correction with the overhead of reading the erroneous page repeatedly by using 3 different read reference voltages. At this level, cello, Ms-Cambridge and websearch workloads present less degradation than financial and postmark since they have more sequential read requests (as in Figure 8.11), and they can amortize the penalty of reading neighbor pages.

How Page Size Affects Performance: Page size is one of the key parameters in designing flash memory. Thus, the page sizes of different flash memories could be different in a range from 2kByte to 8kByte among manufactures and generations. In this section, we will evaluate how the system performance is affected by flash memory page size. In Figure 8.13(a) and (b), we present the performance degradation of NAC according to page sizes when the ECC failure rate is 1% and 33% respectively. These two ECC failure rate corresponds to ECC failure at the end of stage-2 and stage-3 of extended P/E cycles. Reducing page sizes from 8 KB to 2 KB affects two main aspects essential for the performance degradation of NAC: Hit ratio of victim pages and neighbor pages in the NAC-buffer. However, when the page size is reduced, these two aspects are affected in the reverse order. For the hit ratio of victim pages, assume that two consecutive requests read 16KB from the same address. When we use 8KB pages in the flash memory, the first request fetches two pages to the NAC buffer, and the second request hits these two pages in the NAC buffer. However, when we reduce the page size to 2KB, at the end of the first request, only the last 10 KB of the read data stays in the NAC-buffer, and second request does not hit the buffer. Thus, using smaller pages reduces the hit rate of victim pages in the NAC-buffer. On the other side, for the hit ratio of neighbor pages, we request 8KB from the flash disk. When we read the page from the flash disk in finer granularity (i.e. 2KB), the neighbors of the first page is read within the same request. However, for larger page sizes, the request reads only one page without neighbors. Thus, reducing page sizes increases the hit ratio of neighbor pages in the NAC-buffer. There are two main observations that we can make from Figure 8.13(a) and (b). First, when the ECC failure rate is 1%, for the benchmarks presenting high hit rates for the victim pages (i.e. cello, financial and Ms-Cambridge), increasing the page size reduces the performance degradation of NAC. For the rest of benchmarks, increasing the page size slightly increases the performance degradation of NAC in the same ECC failure rate. Second, when the ECC failure rate is high at 33%, the hit ratio of neighbor pages becomes more important than the hit ratio of victim pages. Thus, increasing the page size mostly increases the performance degradation of NAC.

8.5 Summary

NAND flash memory has been widely used as a storage medium for many systems. Although flash memories present high performance, large storage density, non-volatility and low power consumption, a flash memory cell has a limited endurance. In this chapter, we present two main flash data correct techniques: 1) Flash Correct and Refresh (FCR) and 2) Neighbor-cell Assisted Correction (NAC) .

FCR greatly reduces the raw BER of flash media by controlling the dominant retention errors below a certain level that simple ECC can handle through periodic data correct and refresh operations. Our hybrid FCR and adaptive FCR can greatly reduce the overhead introduced by additional P/E operations due to refresh. Our experimental evaluation results show that FCR is effective in improving flash storage system lifetime with only minor overhead. FCR's flexible configurability makes it potentially applicable to a variety of systems. Its benefits would increase as the endurance of flash memory further decreases in the near future as flash cell size continues to shrink and more than 2 bits are programmed per cell.

NAC is a low-overhead and high-accuracy error correction method which corrects programming errors by leveraging information from neighbor cells to correct errors in cells that are being read from flash memory. Our experimental evaluations show that NAC can significantly reduce the raw bit error rate and improve flash memory lifetime at zero or very modest performance overheads. As flash memory scales down to smaller technology nodes and cell-to-cell interference therefore becomes an even more dominant cause of errors, we expect that the error correction techniques proposed in this chapter will become even more important for reliable operation.

9 Conclusion

It is foreseen that technology trend will increase the transient and permanent fault rates in future processors. Thus, providing reliability for both the applications running on personal computers as well as running on mission-critical systems is becoming an absolute necessity. On the other side, due to the lower power budget of the computer systems such that mobile devices, it is attractive to scale down the voltage. However, when the voltage level scales to below the safe margin especially to the ultra-low level, the error rate increases drastically. Moreover, new memory technologies can provide only limited amount of nominal lifetime and when they excite this lifetime they also present high number of faults. In this dissertation, we present reliability designs for three main purposes: 1) Increasing the reliability performance of computer systems, 2) Reducing the energy consumption of the computer system by allowing it to operate reliably at ultra-low voltage level, 3) Increasing the lifetime of new memory technologies by presenting reliability schemes to be utilized in the extended lifetime.

In Chapter 3, we introduced FIMSIM, our fault injection infrastructure for microarchitectural simulators. We inject transient, permanent and intermittent faults both as a single bit fault or as multi-bit faults to in-order microarchitecture using Alpha [53] instruction set. We utilized

FIMSIM to evaluate our reliability schemes such as SymptomTM and FaulTM.

In Chapter 4, we introduce SymptomTM, an availability approach that leverages lazy-lazy hardware transactional memory (HTM) system for error recovery and detects errors by monitoring error symptoms. SymptomTM avoids error propagation to the whole system by utilizing the isolation property of transactions.

In Chapter 5, we present FaulTM, a redundancy-based error detection and recovery proposal based on Hardware Transactional Memory (HTM) providing high reliability for mission-critical systems. FaulTM reduces the comparison overhead of replication significantly by comparing the redundant execution streams at the end of the transactions instead of after every store instruction. SymptomTM and FaulTM are the two first research proposals that leverage transactional memory hardware for error detection and recovery. During the time frame of this thesis, there have been several other proposals that we explain them at the end of the Chapter 5.

In Chapter 6, we discussed multiple error detection alternatives that utilize Transactional Memory for error recovery in order to minimize the energy consumption of CPUs by executing at ultra-low voltage level. Based on our evaluation, we conclude that the replication can provide energy reduction up to 28% while still providing high reliability.

In Chapter 7, we propose two approaches to support that L1 caches can operate at the ultra-low voltage level. In the first approach, we utilize a fast and low-complexity SEC-MAEC code in order to reduce the error correction latency and energy consumption. In the second approach, we present a novel circuit design which configures itself for different supply voltages in order to duplicate or triplicate each data line if higher reliability is required. Flexicache can continue to operate reliably up to 10% bit failure rate. Therefore, it alters the possibility to operate in 320 mV.

In Chapter 8, we present two main flash data correct techniques which greatly increase the lifetime of flash memories: 1) Flash Correct and Refresh (FCR) and 2) Neighbor-cell Assisted Correction (NAC). FCR controls the dominant retention errors below a certain level that simple ECC can handle through periodic data correct and refresh operations. We present three versions of FCR; remapping-based, hybrid and adaptive-rate. Our hybrid FCR and adaptive FCR can greatly reduce the overhead introduced by additional P/E operations due to refresh. Our experimental evaluation results show that FCR can provide on average 46× lifetime improvement. On the other side, NAC is a low-overhead and high-accuracy error correction method which corrects programming errors by leveraging information from neighbor cells to correct errors in cells that are being read from flash memory. Our experimental evaluations show that NAC can significantly reduce the raw bit error rate and improve flash memory lifetime at zero or very modest performance overheads.

9.1 Future Work:

Some of the contributions described in this thesis may be further extended, more specifically, we believe that fault injection (Chapter 3), scaling voltage (Chapter 7), and reliability for emerging technologies (Chapter 8) offer clear further research directions that might be worth exploring.

Fault injection in the micro-architectural simulator, such as FIMSIM, provides the correlation between the criticality of circuit level faults and their impact on the application level. In this thesis, we only inject faults to in-order microarchitecture using Alpha instruction set. However, different instruction sets and different architectures (i.e., in-order or out-of-order) may affect the reliability of the applications differently. By extending FIMSIM for different architectures, those effects can be further investigated.

In Chapter 7, we present flexicache which configures itself entirely according to the level of the applied voltage. However, configuring the cache in the fine granularity may provide higher benefits for applications such that reducing the voltage when applications can tolerate high memory latencies for some part of its variables. Thus, providing a hybrid configuration of Flexicache which includes replicated and non-replicated parts at a time is a promising future research direction.

Technologies such as Flash exhibit new, previously unencountered, fault types such as endurance. Endurance refers to the inability of a memory cell to function correctly after a number of writes. In Chapter 8, we propose several sophisticated techniques to mitigate the Flash endurance problem. In order to mitigate any fault types associated with future memory technologies, similar unconventional ideas should be utilized.

10 Publication List

The content of this thesis led to following publications:

Journals:

1. Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Adrian Cristal, Osman S. Unsal, Ken Mai, Error Analysis and Retention-Aware Error Management for NAND Flash Memory, Intel Technology Journal, Volume 17, Issue 1 – June 2013

Conferences:

2. Gulay Yalcin, Azam Seyedi, Osman Unsal, Adrian Cristal, Flexicache: Highly Reliable and Low Power Cache under Supply Voltage Scaling, CARLA Latin American High Performance Computing Conference, October 2014

3. Gulay Yalcin, Emrah Islek, Oyku Tozlu, Pedro Reviriego, Adrian Cristal, Osman S. Unsal, Oguz Ergin, Exploiting a Fast and Simple ECC for Scaling Supply Voltage in Level-1 Caches, 20th IEEE International On-Line Testing Symposium (IOLTS), July 2014

4. Yu Cai, Gulay Yalcin, Onur Mutlu, Eric Haratsch, Osman Unsal, Adrián Cristaland, Ken Mai, Neighbor-Cell Assisted Error Correction for MLC NAND Flash Memories, In Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), June 2014

5. Gulay Yalcin, Anita Sobe, Alexey Voronin, Jons-Tobias Wamhoff, Derin Harmanci, Adrián Cristal, Osman Unsal, Pascal Felber, Christof Fetzer, Combining Error Detection and Transactional Memory for Energy-efficient Computing below Safe Operation Margins, In 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Feb 2014

6. Gulay Yalcin, Osman S Unsal, Adrián Cristal, Fault Tolerance for Multi-Threaded Applications by Leveraging Hardware Transactional Memory, In Proceedings of 10th Conference on ACM Computing Frontiers (CF), May 2013

7. Azam Seyedi, Gulay Yalcin, Osman S. Unsal, Adrián Cristal, Circuit Design of a Novel Adaptable and Reliable L1 Data Cache, In 23rd Great Lakes Symposium on Very Large Scale Integration (GLSVLSI), May 2013

8. Gulay Yalcin, Osman Unsal, Adrián Cristal, FaulTM: Error Detection and Recovery Using Hardware Transactional Memory, In International Conference of Design, Automation, and Test in Europe (DATE), Mar 2013

9. Yu Cai, Gulay Yalcin, Onur Mutlu, Eric Haratsch, Adrián Cristal, Osman Unsal, Ken Mai, Flash Correct-and-Refresh: Retention-Aware Error Management for Increased Flash Memory Lifetime, 30th IEEE International Conference on Computer Design (ICCD), Oct 2012

10. Gulay Yalcin, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, Mateo Valero, SymptomTM: Symptom Based Error Detection and Recovery Using Hardware Transactional Memory, 20th ACM International Conference on Parallel Architectures and Compilation Techniques (PACT), Oct 2011

11. Gulay Yalcin, Osman S. Unsal, Adrián Cristal, Mateo Valero, FIMSIM: A Fault Injection Tool for Microarchitectural Simulator, 29th IEEE International Conference on Computer Design (ICCD), Oct 2011

Workshops:

12. Gulay Yalcin, Santhosh Kumar, Rethinagiri , Oscar Palomar, Adrian Cristal, Osman S. Unsal. ParaDIME: Parallel Distributed Infrastructure for Minimization of Energy at Architecture Level. ICT Energy Newsletters, Jul 2014
13. Oscar Palomar, Gulay Yalcin, Santhosh Kumar Rethinagiri, Osman Unsal, Adrian Cristal Kestelman, Gina Alioto. ParaDIME: Parallel Distributed Infrastructure for Minimization of Energy. Joint Euro-TM/MEDIAN Workshop on Dependable Multicore and Transactional Memory Systems (DMTM), Jan 2013.
14. Adrián Cristal, Osman Unsal, Gulay Yalcin, Christof Fetzer, Jons-Tobias Wamhoff, Pascal Felber, Derin Harmanci, Anita Sobe. Leveraging Transactional Memory for Energy-efficient Computing below Safe Operation Margins. In 8th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2013), Mar 2013.
15. Gulay Yalcin, Osman S. Unsal, Adrian Cristal, Mateo Valero. FaulTM-multi: Fault Tolerance for Multithreaded Applications Running on Transactional Memory Hardware. Workshop on Wild and Sane Ideas in Speculation and Transactions, Oct 2011.
16. Gulay Yalcin, Osman Unsal, Ibrahim Hur, Adrian Cristal, Mateo Valero. FaulTM: Fault-Tolerance Using Hardware Transactional Memory. The 3rd Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA), Jun 2010.

Following studies, which are not included in the thesis, have also been published during the PhD studies:

1. Gulay Yalcin, Oguz Ergin, Emrah Islek, Osman Unsal, Adrian Cristal. Exploiting Existing Comparators for Fine-Grained Low-cost Error Detection. ACM Transactions on Architecture and Code Optimization, 2014
2. Serdar Zafer Can, Gulay Yalcin, Oguz Ergin, Osman S. Unsal, Adrian Cristal. Bit Impact Factor: Towards Making Fair Vulnerability Comparison. Elsevier Microprocessors and Microsystems (MICPRO) - May 2014

3. Gulay Yalcin, Oguz Ergin. Using Tag-Match Comparators for Detecting Soft Errors. IEEE Computer Architecture Letters (CAL) 6(2): 53-56, 2007

4. Gulay Yalcin, Osman Unsal and Mateo Valero. Architectural Mechanisms Leveraging the Floating Point Subsystem for Soft Error Mitigation. In Second Workshop on Compiler and Architectural Techniques for Application Reliability and Security (CATARS-2), Jun 2009.

5. Oguz Ergin, Gulay Yalcin, Osman Unsal, Mateo Valero. Exploiting the Dependency Checking Logic of the Rename Stage for Soft Error Detection. In First Workshop on Design for Reliability (DFR), Jan 2009

Bibliography

[1] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. *SIGARCH Computer Architecture News*, 32:102, 2004. xv, 36, 40, 54, 81

[2] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pages 7–18, 2003. xvi, 86, 88, 96

[3] J. Chang, G.A. Reis, and D.I. August. Automatic instruction-level software-only recovery. In *International Conference on Dependable Systems and Networks*, pages 83–92, 2006. xvi, 97, 98

[4] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. Software-Implemented Hardware Error Detection: Costs and Gains. In *The Third International Conference on Dependability*, Los Alamitos, CA, USA, 2010. IEEE Computer Society. xvi, 94, 95, 97, 98

[5] http://status.aws.amazon.com/s3-20080720.html. 1, 9

[6] Richard W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 29:147–160, 1950. 2, 13

[7] Hyojin Choi, Wei Liu, and Wonyong Sung. VLSI Implementation of BCH Error Correc-

tion for Multilevel Cell NAND Flash Memory. *IEEE Transactions on Very Large Scale Integration Systems*, 18(5):843–847, May 2010. 2, 146, 159

[8] R. G. Dreslinski et al. Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits. *Proceedings of the IEEE*, 98(2):253–266, 2010. 4, 86

[9] Pedro Reviriego, Salvatore Pontarelli, J. Maestro, and Marco Ottavi. Low-cost Single Error Correction Multiple Adjacent Error Correction Codes. *IEEE Electronic Letters*, 48:1470–1472, 2012. 5, 107, 115

[10] Swapna Yasarapu. Architectural Requirements for MLC based SSDs. In *Flash Memory Summit*, 2011. 5, 146

[11] Christopher Weaver, Joel Emer, Shubhendu S. Mukherjee, and Steven K. Reinhardt. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In *In Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 264–275, 2004. 7

[12] Robert Baumann. Soft Errors in Advanced Computer Systems. *IEEE Design and Test of Computers*, 22(3):258–266, 2005. 8

[13] Nematollah Bidokhti. SEU Concept to Reality (Allocation, Prediction, Mitigation). In *RAMS*, 2010. 8

[14] Timothy C. May and Murray H. Moods. Alpha-Particle-Induced Soft Errors in Dynamic Memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, January 1979. 9

[15] James F. Ziegler and Helmut Puchner, editors. *SER - History, Trends and Challenges: A Guide for Designing with Memory ICs* . Cypress Semiconductor Corporation, 2010. 9

[16] Eugene Norm. Single Event Upset at Ground Level. *IEEE Transactions on Nuclear Science*, 43(6):2742–2750, December 1996. 9

[17] Sarah E. Michalak, Kevin W. Harris, Nicolas W. Hengartner, Bruce E. Takala, and Stephen A. Wender. Predicting the Number of Fatal Soft Errors in Los Alamos National Labratory's ASC Q Computer. *IEEE Transactions on Device and Materials Reliability*, 5:329–335, 2005. 9

[18] Manoj Franklin et al. Built-in Self-Testing of Random-Access Memories. *IEEE Computer*, 23(10), October 1990. 11, 109, 124

[19] Cristian Constantinescu. Trends and challenges in vlsi circuit reliability. *IEEE Micro*, 23:14–19, July 2003. 11, 110

[20] Philip M. Wells et al. Adapting to Intermittent Faults in Multicore Systems. In *Proceedings of the 13th ASPLOS*, pages 255–264, 2008. 11

[21] Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. Towards understanding the effects of intermittent hardware faults on programs. *Dependable Systems and Networks Workshops*, 0:101–106, 2010. 11, 31

[22] Cristian Constantinescu. Impact of deep submicron technology on dependability of vlsi circuits. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 205–209, 2002. 11

[23] Cristian Constantinescu. Intermittent faults in vlsi circuits. In *Proceedings of the IEEE Workshop on Silicon Errors in Logic System Effects*, 2006. 11

[24] Riaz Naseer and Jeff Draper. Parallel double error correcting code design to mitigate multi-bit upsets in srams. In *34th European Solid-State Circuits Conference*, pages 222–225, 2008. 12

[25] Nishant J. George, Carl R. Elks, Barry W. Johnson, and John Lach. Transient fault models and avf estimation revisited. *Dependable Systems and Networks*, 0:477–486, 2010. 12, 31

[26] Shubhendu S. Mukherjee, Joel Emer, Tryggve Fossum, and Steven K. Reinhardt. Cache Scrubbing in Microprocessors: Myth or Necessity? In *In 10th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 37–42, 2004. 12

[27] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. *SIGARCH Computer Architecture News*, 28(2):25–36, 2000. 13, 53, 66, 68

[28] Eric Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, page 84, 1999. 13

[29] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the International Symposium on Computer Architecture*, pages 87–98, 2002. 13, 53, 66

[30] Rui Gong, Kui Dai, and Zhiying Wang. Transient Fault Recovery on Chip Multiprocessor based on Dual Core Redundancy and Context Saving. *International Conference for Young Computer Scientists*, pages 148–153, 2008. 13, 14, 15, 42, 47, 52, 53, 55, 56, 65, 66, 68

[31] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of the International Symposium on Computer Architecture*, pages 99–110, 2002. 13, 14, 15, 47, 49, 52, 53, 55, 56, 65, 66, 68

[32] T. J Slegel and et al. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, 19:12–23, 1999. 13, 47, 48, 49, 55, 56, 91

[33] Alan Wood, Robert Jardine, and Wendy Bartlett. Data Integrity in HP NonStop Servers. In *Proceedings of the Workshop on System Effects of Logic Soft Errors*, 2006. 13, 14, 47, 48, 49, 56, 91

[34] Carles Hernandez and Jaume Abella. Live: Timely error detection in light-lockstep safety critical systems. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 25:1–25:6, 2014. 14

[35] Infineon. AURIX - TriCore datasheet. Highly Integrated and Performance Optimized 32-bit Microcontrollers for Automotive and Industrial Applications. 14

[36] Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe. Reunion: Complexity-Effective Multicore Redundancy. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 223–234, 2006. 14, 15, 42, 47, 52, 66

[37] Jared C. Smolens, Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas Nowatzyk. Fingerprinting: Bounding Soft-error Detection Latency and Bandwidth. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, 2004. 15, 16, 47, 52, 53, 58, 66, 75, 76, 84

[38] Kypros Constantinides, Onur Mutlu, Todd Austin, and Valeria Bertacco. Software-Based Online Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation. In *Proceeding of the 40th Annual International Symposium on Microarchitecture*, pages 97–108, 2007. 15

[39] M. Li, P. Ramach, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008. 15, 33, 34, 44

[40] Nicholas J. Wang, Student Member, and Sanjay J. Patel. Restore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable Secure Computing*, 3:188–201, 2006. 15, 33, 41, 93

[41] Shuguang Feng et al. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *Proceeding of ASPLOS*, pages 385–396, 2010. 15

[42] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 289 –298, jun 1995. 16

[43] Prvulovic, Zheng Zhang, and Josep Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 111–122, 2002. 16, 33, 40, 76, 83

[44] R.E. Ahmed, R.C. Frazier, and P.N. Marinos. Cache-aided rollback error recovery (carer) algorithm for shared-memory multiprocessor systems. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 82 –88, jun 1990. 16

[45] K. L. Wu, W. K. Fuchs, and J. H. Patel. Error recovery in shared memory multiprocessors using private caches. *IEEE Trans. Parallel Distrib. Syst.*, 1(2):231–240, April 1990. 16

[46] Rishi Agarwal, Pranav Garg, and Josep Torrellas. Rebound: Scalable Checkpointing for Coherent Shared Memory. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 153–164, 2011. 16, 17, 68

[47] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the International Symposium on Computer Architecture*, pages 123–134, 2002. 16, 33, 40, 76, 83

[48] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002. 17

[49] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, May 1992. 17

[50] Daniel Gil, Joaquin Gracia, Juan Carlos Baraza, and Pedro J. Gil. Study, comparison and application of different vhdl-based fault injection techniques for the experimental validation of a fault-tolerant system. *Microelectronics Journal*, 34(1):41–51, 2003. 20, 30

[51] Nicholas J. Wang, Justin Quek, Todd M. Rafacz, and Sanjay J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 61–70, 2004. 20, 30

[52] Nathan L. Binkert et al. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26:52–60, 2006. 20, 21, 43, 67, 99

[53] *Alpha 21264 Microprocessor Hardware Reference Manual*. Compaq Computer Corparation, 1999. 20, 43, 67, 177

[54] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34:1–17, 2006. 20, 43, 67

[55] Man lap Li, Pradeep Ramach, Ulya R. Karpuzcu, Siva Kumar, Sastry Hari, and Sarita V. Adve. Accurate microarchitecture-level fault modeling for studying hardware faults *. In *International Symposium of High-Performance Computer Architecture*, 2009. 26, 27, 31

[56] Naghmeh Karimi, Michil Maniatakos, Abhijit Jas, and Yiorgos Makris. On the Correlation Between Controller Faults and Instruction Level Errors in Modern Microprocessors. In *International Test Conference*, 2008. 27

[57] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, 2008. 27, 31, 93

[58] Nicholas J. Wang, Aqeel Mahesri, and Sanjay J. Patel. Examining ace analysis reliability estimates using fault-injection. In *34th Annual International Symposium on Computer Architecture*, pages 460–469, 2007. 27, 29, 30

[59] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 29–40, 2003. 29

[60] Michail Maniatakos, Naghmeh Karimi, Chandra Tirumurti, Abhijit Jas, and Yiorgos Makris. Instruction-level impact analysis of low-level faults in a modern microprocessor controller. *IEEE Transactions on Computers*, 99, 2010. 30

[61] J. Gracia, L. J. Saiz, J. C. Baraza, D. Gil, and P. J. Gil. Analysis of the influence of intermittent faults in a microcontroller. In *Proceedings of the 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, pages 1–6, 2008. 31

[62] Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V. Adve. mSWAT: Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 122–132, 2009. 33, 34, 40, 54

[63] Ron A. Oldfield, Patricia J. Teller, Maria Ruiz Varela, Philip C. Roth, Sarala Arunagiri, Seetharami Seelam, and Rolf Riesen. Modeling the Impact of Checkpoints on Next-generation Systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–43, 2007. 34

[64] Man lap Li, Pradeep Ramach, Swarup K. Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *Proceedings of the 38th Intrnational Conference of Dependable Systems and Networks*, 2009. 34, 38, 54

[65] Tim Harris, Adrián Cristal, Osman S. Unsal, Eduard Ayguade, Fabrizio Gagliardi, Burton Smith, and Mateo Valero. Transactional memory: An overview. *IEEE Micro*, 27(3): 8–29, 2007. 35, 86

[66] Maurice Herlihy, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*, pages 289–300, 1993. 35

[67] Allon Adir, Dave Goodman, Daniel Hershcovich, Oz Hershkovitz, Bryan Hickerson, Karen Holtz, Wisam Kadry, Anatoly Koyfman, John Ludden, Charles Meissner, Amir Nahir, Randall R. Pratt, Mike Schiffli, Brett St. Onge, Brian Thompto, Elena Tsanko, and Avi Ziv. Verification of transactional memory in power8. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 58:1–58:6, 2014. 35

[68] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 19:1–19:11, 2013. 35

[69] S. Sanyal, S. Roy, A. Cristal, O. S. Unsal, and M. Valero. Dynamically Filtering Thread-Local Variables in Lazy-Lazy Hardware Transactional Memory. In *Proceedings of the International Conference on High Performance Computing and Communications*, pages 171–179, 2009. 43, 67

[70] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceeding of the International Conference*

on Architectural Support for Programming Languages and Operating Systems, pages 257–268, 2000. 49

[71] Anantha Chandrakasan, William J. Bowhill, and Frank Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001. 56

[72] Dmitry V. Ponomarev, Gurhan Kucuk, Oguz Ergin, and Kanad Ghose. Energy efficient comparators for superscalar datapaths. *IEEE Transactions on Computers*, 53:892–904, 2004. 56

[73] CVE-2008-1368. Predictive technology model, November 2010. [online] http://ptm.asu.edu. 56

[74] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers, 2007. 58

[75] Vladimir Gajinov, Ferad Zyulkyarov, Osman S. Unsal, Eduard Ayguade, Tim Harris, Mateo Valero, and Adrian Cristal. QuakeTM: Parallelizing a Complex Sequential Application Using Transactional Memory. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 126–135, 2009. 63

[76] Gokcen Kestor, Vasileios Karakostas, Osman S. Unsal, Adrian Cristal, Ibrahim Hur, and Mateo Valero. RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems. In *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering*, pages 335–346, 2011. 63

[77] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *SIGARCH Compututer Architecture News*, 23:24–36, May 1995. 67, 99

[78] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008. 67

[79] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized

Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005. 68

[80] Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Cherry-mp: correctly integrating checkpointed early resource recycling in chip multiprocessors. In *International Symposium on Microarchitecture*, 2005. 73

[81] Ruben Gil Titos and et al. Hardware transactional memory with software-defined conflicts. In *TRANSACT*, 2010. 73

[82] Daniel Sanchez, Juan M. Cebrian, Jose M. Garcia, and Juan L. Aragon. Soft-error mitigation by means of decoupled transactional memory threads. *Distributed Computing*, pages 1–16, 2014. 80, 86

[83] Jons-Tobias Wamhoff, Mario Schwalbe, Rasha Faqeh, Christof Fetzer, and Pascal Felber. Transactional encoding for tolerating transient hardware errors. In *Stabilization, Safety, and Security of Distributed Systems: 15th International Symposium (SSS 2015)*, volume 8255, pages 1–16. November 2013. 80, 86, 94

[84] Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, Adrià Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. Eazyhtm: Eager-lazy hardware transactional memory. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 145–155, 2009. 81

[85] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: log-based transactional memory. *International Symposium on High-Performance Computer Architecture*, 12:254–265, 2006. 81

[86] H. Esmaeilzadeh, E. Blem, R.S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 38th Annual International Symposium on*, pages 365–376. IEEE, 2011. 85, 105

[87] Jaydeep P. Kulkarni et al. A 160 mV Robust Schmitt Trigger Based Sub-threshold SRAM. *IEEE Journal of Solid-State Circuits*, 42(10):2303–2313, October 2007. 85

[88] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011. 86, 102

[89] Y. Zhang and K. Chakrabarty. Fault recovery based on checkpointing for hard real-time embedded systems. In *Defect and Fault Tolerance in VLSI Systems, Proceedings. 18th IEEE International Symposium on*, pages 320–327. IEEE, 2003. 86

[90] C. Fetzer and P. Felber. Transactional memory for dependable embedded systems. In *7th Workshop on Hot Topics in System Dependability (HotDep)*, pages 223–227. IEEE, 2011. 86, 92

[91] A. Gautham, K. Korgaonkar, P. Slpsk, S. Balachandran, and K. Veezhinathan. The implications of shared data synchronization techniques on multi-core energy efficiency. In *Proceedings of the USENIX conference on Power-Aware Computing and Systems*, pages 6–6. USENIX Association, 2012. 86, 87

[92] Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Energy reduction in multiprocessor systems using transactional memory. In *Proceedings of the international symposium on Low power electronics and design*, ISLPED '05, pages 331–334, 2005. 87

[93] Intel® Turbo Boost Technology 2.0, http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html. 87

[94] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, 1994. 87

[95] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*, pages 13–25, 1995. 87

[96] Tim Harris. Exceptions and side-effects in atomic blocks. In *CSJP'04: Proceedings of the Workshop on Concurrency and Synchronization in Java programs*, pages 46–53, 2004. 88

[97] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'05)*, pages 48–60, 2005. 88

[98] Christof Fetzer and Pascal Felber. Improving program correctness with atomic exception handling. *Journal of Universal Computer Science*, 13(8):1047–1072, 2007. 88

[99] Derin Harmanci, Vincent Gramoli, and Pascal Felber. Atomic boxes: Coordinated exception handling with transactional memory. In *Proceedings of the 25th European Conference on Object Oriented Programming (ECOOP'11)*, Jul 2011. 88

[100] Jeffrey Oplinger and Monica S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 184–196, 2002. 88

[101] Ioannis Doudalis and Milos Prvulovic. Euripus: A flexible unified hardware memory checkpointing accelerator for bidirectional-debugging and reliability. In *International Symposium on Computer Architecture*, pages 261–272, 2012. 89

[102] D.M. Andrews. Using executable assertions for testing and fault tolerance. In *9th Fault-Tolerance Computing Symp*, pages 20–22, 1979. 92

[103] L.L. Pullum. *Software fault tolerance techniques and implementation*. Artech House Publishers, 2001. 92

[104] A. Mahmood and E.J. McCluskey. Concurrent error detection using watchdog processors-a survey. *Computers, IEEE Transactions on*, 37(2):160–174, 1988. 92

[105] N.G. Leveson, S.S. Cha, J.C. Knight, and T.J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *Software Engineering, IEEE Transactions on*, 16(4):432–443, 1990. 92

[106] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, 2001. 92

[107] T. Knauth, C. Fetzer, and P. Felber. Assertion-driven development: Assessing the quality of contracts using meta-mutations. In *Software Testing, Verification and Validation Workshops. ICSTW. International Conference on*, pages 182–191. IEEE, 2009. 92

[108] T. Harris and S.P. Jones. Transactional memory with data invariants. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06), Ottowa*, 2006. 92

[109] Dong. Chen. Local rollback for fault-tolerance in parallel computing systems, united states patent application, 12/696780. 93

[110] D.H. Yoon and M. Erez. Memory mapped ecc: low-cost error protection for last level caches. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 116–127. ACM, 2009. 94

[111] P. Forin. Vital coded microprocessor principles and application for various transit systems. In *IFAC/IFIP/IFORS Symposium*, pages 79–84, 1989. 94

[112] U. Wappler and M. Müller. Software protection mechanisms for dependable systems. In *Proceedings of the conference on Design, automation and test in Europe*, pages 947–952. ACM, 2008. 94

[113] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *Computer Safety, Reliability, and Security*, volume 6351. Springer Berlin / Heidelberg, 2010. 94

[114] S.K. Sahoo, M.L. Li, P. Ramachandran, S.V. Adve, V.S. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks With FTCS and DCC, DSN. IEEE International Conference on*, pages 70–79. IEEE, 2008. 95

[115] G. Gerosa, S. Curtis, M. D'Addeo, Bo Jiang, B. Kuttanna, F. Merchant, B. Patel, M. Taufique, and H. Samarchi. Efficient Cache Architectures for Reliable Hybrid Voltage Operation Using EDC Codes. In *IEEE Asian Solid-State Circuits Conference*, pages 17–20, 2008. 96, 112

[116] Jaydeep P. Kulkarni et al. A 160 mV Robust Schmitt Trigger Based Sub-threshold SRAM. *IEEE Journal of Solid-State Circuits*, 42(10):2303–2313, October 2007. 105, 109

[117] Shailendra Jain, Surhud Khare, Satish Yada, V. Ambili, Praveen Salihundam, Shiva Ramani, Sriram Muthukumar, M. Srinivasan, Arun Kumar, Shasi Kumar, Rajaraman Ramanarayanan, Vasantha Erraguntla, Jason Howard, Sriram R. Vangal, Saurabh Dighe, Gregory Ruhl, Paolo A. Aseron, Howard Wilson, Nitin Borkar, Vivek De, and Shekhar Borkar. A 280mV-to-1.2V wide-operating-range IA-32 processor in 32nm CMOS. In *IEEE International Solid-State Circuits Conference*, pages 66–68, 2012. 105

[118] Bojan Maric, Jaume Abella, and Mateo Valero. APPLE: Adaptive Performance-Predictable Low-Energy Caches for Reliable Hybrid Voltage Operation. In *The 50th Annual Design Automation Conference*, page 84, 2013. 105

[119] Gregory K. Chen et al. Yield-Driven Near-Threshold SRAM Design. In *ICCAD'07*, pages 660–666. 105

[120] R. G. Dreslinski et al. Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits. *Proceedings of the IEEE*, 98(2):253–266, 2010. 105

[121] Michael A. Bajura et al. Models and Algorithmic Limits for an ECC-Based Approach to Hardening Sub-100-nm SRAMs. *IEEE Trans. Nuclear Science*, 54(4):935–945, August 2007. 106, 110

[122] Zeshan Chishti et al. Improving Cache Lifetime Reliability at Ultra-Low Voltages. In *MICRO*, pages 89–99, 2009. 106, 109, 110, 111, 117, 118, 133, 136, 143

[123] Mu Hsiao et al. Orthogonal Latin Square Codes. *IBM Journal of Research and Development*, 14(4):390–394, July 1970. 106, 107, 111, 112, 133

[124] Timothy Miller et al. Parichute: Generalized Turbocode-Based Error Correction for Near-Threshold Caches. In *MICRO*, pages 351–362, 2010. 106, 109, 110, 111, 117, 118, 124, 136

[125] Stefan Rusu, Harry Muljono, David Ayers, Simon Tam, Wei Chen, Aaron Martin, Shenggao Li, Sujal Vora, Raj Varada, and Eddie Wang. 5.4 Ivytown: A 22nm 15-core enterprise Xeon® processor family. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 102–103, 2014. 106

[126] Arup Chakraborty et al. E < mc2: Less energy through multi-copy cache. In *CASES*, pages 237–246, 2010. 106, 107, 124

[127] Wei Zhang. Replication Cache: A Small Fully Associative Cache to Improve Data Cache Reliability. *IEEE Transactions on Computers*, 54:1547–1555, 2005. 106, 107, 124

[128] Amit Agarwal and et al. Process Variation in Embedded Memories: Failure Analysis and Variation Aware Architecture. *IEEE Journal of Solid-State Circuits*, 40(9):1804–1814, 2005. 109

[129] Robert Baumann. Soft Errors in Advanced Computer Systems. *IEEE Design and Test*, 22:258–266, 2005. 110

[130] C. L. Chen and M. Y. Hsiao. Error-Correcting Codes for Semiconductor Memory Applications: A State-Of-The-Art Review. *IBM Journal of Research and Development*, 28 (2):124–134, March 1984. 110

[131] Jangwoo Kim et al. Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding. In *MICRO*, 2007. 111

[132] Doe Hyun Yoon and Mattan Erez. Memory Mapped ECC: Low-Cost Error Protection for Last Level Caches . In *Proceedings of the 36th International Symposium on Computer Architecture*, 2009. 111

[133] Chris Wilkerson et al. Trading off Cache Capacity for Reliability to Enable Low Voltage Operation. In *ISCA*, pages 203–214, 2008. 111, 112, 123, 124

[134] Jaume Abella et al. Low Vccmin Fault-Tolerant Cache with Highly Predictable Performance. In *MICRO*, pages 111–121, 2009. 111, 124

[135] Amin Ansari et al. ZerehCache: Armoring Cache Architectures in High Defect Density Technologies. In *MICRO*, pages 100–110, 2009. 111, 112, 122

[136] Cheng-Kok Koh, Weng-Fai Wong, Yiran Chen, and Hai Li. Tolerating process variations in large, set-associative caches: The buddy cache. *ACM Transactions on Architecture and Code Optimization*, 6(2):8:1–8:34, July 2009. 112

[137] Yasuhiro Morita, Hidehiro Fujiwara, Hiroki Noguchi, and Yusuke Iguchi. An Area-Conscious Low-Voltage-Oriented 8T-SRAM Design under DVS Environment. *IEEE Symposium on VLSI Circuits*, pages 256–257, June 2007. 112

[138] Jaydeep P. Kulkarni, Keejong Kim, and Kaushik Roy. A 160 mV, Fully Differential, Robust Schmitt Trigger Based Sub-threshold SRAM. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, pages 171–176, 2007. 112

[139] Bojan Maric, Jaume Abella, and Mateo Valero. Efficient Cache Architectures for Reliable Hybrid Voltage Operation Using EDC Codes. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 917–920, 2013. 112

[140] C McNairy and D Soltis. Itanium 2 Processor Microarchitecture. *IEEE Micro*, 23: 44–45, 2003. 121

[141] Cameron Mcnairy and Rohit Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 2005. 122

[142] Daniel J. Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the 29th ISCA*, pages 123–134, 2002. 124

[143] Azam Seyedi et al. Circuit Design of a Dual-Versioning L1 Data Cache for Optimistic Concurrency. In *GLSVLSI*, pages 325–330, 2011. 124

[144] Adria Armejach et al. Using a Reconfigurable L1 Data Cache for Efficient Version Management in Hardware Transactional Memory. In *PACT'11*. 124

[145] Predictive technology model, http://ptm.asu.edu/. 125, 136

[146] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P Jouppi. CACTI 5.1. Technical report, HP Laboratories, 2008. 125, 136

[147] Stefan Cosemans, Wim Dehaene, and Fransky Catthoor. A 3.6 pJ/Access 480 MHz, 128 kb On-Chip SRAM with 850 MHz Boost Mode in 90 nm CMOS with Tunable Sense Amplifiers. *IEEE Journal of Solid-State Circuits*, 44:2065–2077, 2009. 126

[148] The Electric VLSI Design System: http://www.staticfreesoft.com. 142

[149] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, June 2005. 146

[150] NXP Semiconductors Application Note. AN10860 LPC313x NAND flash data and bad block management, Aug 2009. 146

[151] Yan Li, Seungpil Lee, Yupin Fong, and Feng Pan. A 16 Gb 3-Bit Per Cell (X3) NAND Flash Memory on 56 nm Technology With 8 MB/s Write Rate. *IEEE Journals of Solid-State Circuits*, 44(1):195–207, January 2009. 149, 155

[152] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Um, and Jin-Ki Kim. A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme. In *Solid-State Circuits Conference*, pages 128–129, 1995. 150

[153] Hyunyoung Shim, Seaung-Suk Lee, Byungkook Kim, and Namjae Lee. Highly Reliable 26nm 64Gb MLC E2NAND Flash Memory with MSP Controller. In *Symposium on VLSI Technology*, pages 216–217, 2011. 151

[154] Yu Cai, Onur Mutlu, Erich F. Haratsch, and Ken Mai. Program interference in mlc nand flash memory: Characterization, modeling, and mitigation. In *ICCD*, pages 123–130, 2013. 151, 152

[155] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. Threshold voltage distribution in mlc nand flash memory: Characterization, analysis, and modeling. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1285–1290, 2013. 151

[156] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. Error patterns in mlc nand flash memory: Measurement, characterization, and analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 521–526, 2012. 151, 152, 153, 157

[157] Yangyang Pan, Guiqiang Dong, Qi Wu, and Tong Zhang. Quasi-nonvolatile SSD: Trading Flash Memory Nonvolatility to Improve Storage System Performance for Enterprise Applications. In *High Performance Computer Architecture*, pages 179–188, 2012. 152

[158] John S. Bucy and Gregory R. Ganger. The DiskSim Simulation Environment Version 3.0 Reference Manual. Technical report, 2003. 165

[159] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70, 2008. 165

[160] IOzone.org, "IOzone Filesystem Benchmark" , http://iozone.org. 165

[161] Open Source software at HP Labs, http://tesla.hpl.hp.com/opensource. 165

[162] Jeffrey Katcher. PostMark: A New File System Benchmark. Technical Report TR3022. Technical report, Network Appliance Inc, 1997. 165

[163] SNIA: IOTTA Repository, http://iotta.snia.org/tracetypes/3. 165

[164] UMass Trace: http://traces.cs.umass.edu/index.php/Storage/Storage. 165

[165] Yu Cai, Erich F. Haratsch, Mark McCartney, and Ken Mai. FPGA-Based Solid-State Drive Prototyping Platform. In *International Symposium on Field-Programmable Custom Computing Machines*, pages 101–104, 2011. 165