Cost-Effective Techniques for Processor Reliability

By

David John Palframan

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2015

Date of final oral examination: 4/28/2015

The dissertation is approved by the following members of the Final Oral Committee:

Mikko H. Lipasti, Professor, Electrical and Computer Engineering
Nam Sung Kim, Associate Professor, Electrical and Computer Engineering
Katherine Morrow, Associate Professor, Electrical and Computer Engineering
Kewal Saluja, Professor, Electrical and Computer Engineering
Gurindar Sohi, Professor, Computer Sciences

ACKNOWLEDGMENTS

Graduate school is no small undertaking, and I am deeply grateful to all who have played a part in my journey. First and foremost, I would like to thank my advisors Mikko Lipasti and Nam Sung Kim. Starting out, I had no idea how hugely important the choice of advisor would be. In retrospect, I can say that I have been exceedingly lucky to work with Mikko and Nam over the past six years. I am very grateful for Mikko's unwavering optimism and wisdom and Nam's expertise. It has been a pleasure learning to be a researcher under their mentorship. I'd also like to thank the other members of my committee for their helpful feedback and insight.

I am lucky to have had the encouragement of many engineers and non-engineers alike. I would like to thank my parents for their unconditional support in my decision to stay in school for what I'm sure felt like an eternity. Their encouragement of my curiosity and "engineering side" has played a huge role in my academic and non-academic life. It has also been wonderful to stay in touch with undergraduate friends going through similar programs. I'd like to thank Susie Doughty for her commiseration and perspective over the past six years.

I would also like to extend my thanks to the Madison swing dance community. The wonderful people I've met through Jumptown have been instrumental in keeping me sane and happy during my studies. I consider setting aside Wednesday nights (despite constantly-looming deadlines) to be one of the best choices I made

while in Madison. I am grateful to all my dancer friends (both grads and non-grads) for their friendship and support.

I have also been lucky to have been surrounded by so many talented friends and peers who have taught me so much and made this time more enjoyable. In no particular order: Thanks to Trey Cain for your wonderful mentorship during my internship with Qualcomm. Thanks to Mitch Hayenga and Vignyan Reddy for sharing your microarchitecture expertise and enthusiasm. To Sean Franey, thanks for sharing you research methodology and beer-brewing anecdotes. To Arslan Zulfiqar, thanks for being a willing sounding board for research ideas. To Dibakar Gope, thanks for your company and impressive knowledge of who's who in computer engineering. Thanks to Amin Farmahini Farahani, Tony Gregerson, and Paula Aguilera for your companionship on our lunch/sailing outings. Thanks to Daniel Chang and Kyle Rupnow for making board game nights happen. Thanks also to Aishwarya Nagarajan, Andy Nere, Daniel Chang, and Felix Loh for your friendship and company in and out of the lab.

Finally, I'd like to thank my undergraduate advisor, Prof. Robert Nickel for selling me on graduate school in the first place. When I said that I wanted to switch fields to go into architecture, I'm sure he was relieved that I meant the computer kind.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

As technology scales, new circuit effects and interactions inevitably arise, leading to potential reliability issues. For storage arrays that use SRAM or DRAM, there is often a trade-off between storage density and reliability. Due to their high density, these structures are vulnerable to bit corruption due to process effects or soft errors. Existing protective techniques typically add redundancy in the form of spares or parity bits, allowing errors to be corrected or avoided. This work observes, however, that untapped redundancy is already present in modern processor architectures. Based on this observation, the concept of microarchitectural patching is introduced, in which existing superscalar structures and mechanisms prevent disabled hardware or invalid data from being accessed. For instance, data can be loaded from a structure like the store queue, avoiding a faulty copy in the cache. To highlight the utility of this μarch patching phenomenon, two novel applications are proposed.

First, a technique called iPatch is introduced to efficiently tolerate cache failures caused by voltage scaling and process variations. Voltage scaling is a powerful energy-saving technique, but its effectiveness is often limited by the potential for SRAM cell failures at low voltages. To extend voltage scaling, recent proposals suggest disabling failing portions of on-chip SRAM caches. Although this approach saves power, energy reduction is limited because reducing the usable cache space degrades performance. iPatch regains this lost performance and enables energy

savings by exploiting the μarch patching mechanism. By relying on existing superscalar structures and mechanisms to patch disabled cache subblocks, costly L2 accesses are avoided and energy efficiency is improved. Furthermore, because no critical paths or circuits are affected, normal-voltage operation is not impacted. For high cell failure rates, results show significant energy savings with iPatch as well as an 18% reduction in energy-delay product compared to prior work.

Second, an approach called COP is proposed to protect against bit flips in main memories that would normally not have error-correcting capabilities. Protecting main memories from such errors typically requires special dual-inline memory modules (DIMMs) which incorporate at least one extra chip per rank, increasing costs and power consumption. To avoid these costs, some proposals have suggested protecting non-ECC DIMMs by allocating a portion of memory space to ECC, shrinking the available memory space and degrading performance. Instead, COP uses block-level compression to accommodate ECC check bits along with each block of data in DRAM. COP tracks compressed blocks by using ECC as an indicator of compression, falling back on μarch patching for correctness. COP eliminates extra memory accesses due to ECC, and does not reduce the available memory space. Results show that COP can reduce the DRAM soft error rate by 93% with no storage overhead and negligible impact on performance. To protect incompressible data, a hybrid approach is also proposed that can reduce the storage overhead by 80% and improve performance by 5% compared to prior work.

1 INTRODUCTION

The scaling of process technology allows more transistors to be integrated on a single die, opening the door for increasingly complex and efficient processors. Additional transistors have historically been leveraged to enhance performance through advanced superscalar logic, bigger caches, and more processor cores. In order to fully reap the benefits of smaller transistors, however, various challenges must be overcome.

Technology scaling introduces a number of reliability concerns with potential to harm yields or interfere with correct operation. Issues like soft errors and process variations, for instance, are intensified by continued technology scaling [13, 51]. The growing number of transistors per chip increases the likelihood of a device being slow, faulty, or affected by a particle strike. In addition, saving power by reducing supply voltages can exacerbate such reliability issues. Both soft errors and process variations can result in silent data corruption if they are not properly dealt with. Since it is becoming infeasible to completely mitigate these issues at the circuit level, dedicated error protection circuitry is becoming increasingly common. If the error protection overhead is too high, however, it can eat into the budget that would otherwise be allocated to performance, diminishing the benefits of technology scaling. Thus, as process variations and soft errors become more significant, it is crucial that low-overhead protective mechanisms be employed.

Modern processors employ large SRAM caches and off-chip main memories

that are vulnerable to corruption by soft errors or variation-induced failures. Due to the large number of on-chip and off-chip resources devoted to such storage, protection against errors is critical yet potentially expensive. Aside from circuit-level approaches that modify the storage cells, higher-level techniques to protect these arrays often add additional storage. The extra space can be allocated to hold parity bits for error-correcting codes, serve as spare elements that can be swapped in for faulty ones, or hold redundant copies of data. A common practice in SRAM or DRAM arrays is to incorporate spare rows or columns that can be used as replacements [34]. Such redundancy-based techniques can become expensive, however, especially if high failure rates must be tolerated. As an alternative, other approaches disable failing portions of the array without additional redundancy, reducing the storage capacity [1, 7, 37, 77]. Although this method incurs no storage overhead, it can degrade performance due to decreased cache sizes.

Regardless of the specific approach, it is clear that any method to improve system reliability should make efficient use of the storage space available. To this end, this dissertation proposes a mechanism called microarchitectural patching that takes advantage of existing processor structures and mechanisms to reduce the performance, energy, and resource costs of reliably handling data. The benefits of this approach are demonstrated in two different contexts. First, microarchitectural patching is applied to improve energy efficiency and performance when process variations and voltage scaling require partial cache disabling due to cell failures.

Second, is used to enable a novel scheme to add error correcting codes to standard DRAM modules through data compression. The next section introduces and explains microarchitectural patching, after which an overview of these specific applications is provided.

1.1 Microarchitectural Patching

This dissertation introduces the idea of microarchitectural patching (μarch patching). μArch patching is a powerful phenomenon that can augment reliability-enhancing mechanisms. μArch patching can occur naturally due to the way modern microarchitectures are designed. Today's processors rely on a hierarchy of caches and buffers to significantly reduce data load-to-use latencies and ensure functional correctness. Such structures include data and instruction caches, store queues, fill buffers, and micro-op caches. A key observation is that data redundancy naturally exists across these units. For instance, to improve access latencies, a copy of a data block from main memory can also exist in one or more caches. Similarly, for correctness, loads can be serviced from the store queue instead of the data cache, although the same data block may exist both places. An example of this type of patching is illustrated in Figure 1.1.

There are two important implications of the redundancy that is inherent in modern superscalar processors:

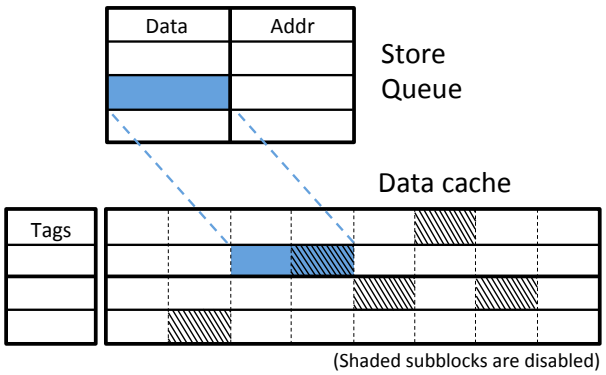


Figure 1.1: Example of microarchitectural patching with the store queue. The same data block exists in the store queue and L1 cache, but loads will be serviced from the store queue and not the partially faulty copy in the cache.

1. If one copy of a duplicated block is determined to be faulty, it is still possible to retrieve the correct data from another copy.

2. When the core requests data, only one copy of the data is accessed, and the copy to access is determined by a hard-coded preferential ranking. Typically, data will be supplied from the structure with the lowest access latency.

Thus, if the "non-preferred" copy of a data block is determined to be faulty or unreliable, no special action or hardware is needed as long as a "preferred" copy exists. For instance, if a data block in main memory is corrupted but a correct copy is cached on chip, it is the cached copy that will always be seen by the processor. Therefore, to best exploit μarch patching, supplemental hardware should focus on managing the data stored in preferred structures instead of explicitly providing a redirection mechanism to fetch fault-free data.

Redundancy in modern systems can also exist in other forms. For instance, the

data being manipulated can itself contain redundancy. Compression algorithms can be used to extract this redundancy as in one of the techniques in this work. Storage freed in this manner can then be used to patch any failures that arise. If the error location is unknown as in the case of soft errors, the "patching" can be accomplished through error-correcting codes.

1.2 Applications of μArch Patching

1.2.1 Addressing On-Chip Variations

Process variations can cause the delicately-balanced SRAM cells in caches to fail, particularly at low voltages [10]. Since the probability of failure increases at lower voltages, this phenomenon can prevent voltage scaling beyond a certain point. This is a significant limitation for low-power systems, since dynamic voltage scaling is one of the most effective energy-saving techniques available. For this reason, prior proposals suggest approaches to tolerate failing cells, opening the door to power and energy reductions. This issue can be particularly difficult to address in L1 caches, however, where design modifications could increase the critical path delay. Prior work by Abella et al. suggests dividing L1 cache blocks into subblocks that can be individually disabled due to failing cells [1]. Although effective, this approach can degrade performance when many subblocks are disabled at very low voltages, since accesses to disabled subblocks are forwarded to the higher-

latency L2 cache. This performance loss can potentially negate the energy savings of voltage scaling.

To address this problem, a technique called iPatch is proposed in this dissertation. iPatch relies on μarch patching to reduce the number of extra L2 accesses incurred because of subblock disabling. For instance, as illustrated in Figure 1.1, a load that is serviced from the store queue need not access the L1 cache. If this load addresses a disabled L1 subblock, the latency of the extra L2 access is avoided. iPatch can use a micro-op cache, store queue, or miss status handling registers to accomplish this patching. Through intelligent management of the contents of these structures, patching behavior is promoted, improving performance. Because iPatch is noninvasive and relies almost entirely on existing structures and mechanisms, it can provide benefits at low-voltages without negatively impacting high-voltage operation. Simulations show that iPatch can provide an 18% average reduction in energy-delay product compared to subblock disabling alone. Additionally, results show that iPatch improves the performance predictability across chips with different fault patterns.

1.2.2 Improving Main Memory Reliability

Main memories are particularly susceptible to failures (e.g. soft errors) due to the sheer number of bits stored. To make main memory more robust, error-correcting codes (ECC) can be incorporated by storing a number of parity bits along with each

block of data [29]. To allow the storage of parity bits in DRAM, special memory modules (DIMMs) with extra chips are typically employed. Using these ECC-enabled DIMMs increases the cost of the hardware as well as its power consumption. Although some prior work has suggested methods for implementing ECC protection in non-ECC DIMMs, these approaches have large performance overheads or substantial storage requirements that reduce the memory space available to software [79, 81].

This dissertation introduces an approach called COP to compress data and protect "non-ECC" DIMMs using ECC. COP uses simple compression schemes to mitigate both the performance and storage overheads of protecting non-ECC DIMMs. Each 64-byte block in DRAM is compressed by a small amount (e.g. to free four bytes), allowing the storage of ECC parity bits in the space created. Data redundancy is thereby exploited to "patch" bit failures using the added ECC. The usable memory space is not impacted and no additional latency is incurred due to ECC, since a compressed data block with parity occupies the same number of bits as an uncompressed block. COP also avoids compression-related addressing complications by maintaining the original 64-byte block alignment in memory. Because COP requires only a small amount of compression, it is effective where other performance-enhancing compression approaches would not be. As shown in Figure 1.2, even relatively incompressible datasets often offer a small amount of compressibility per block. To efficiently exploit this opportunity, this dissertation

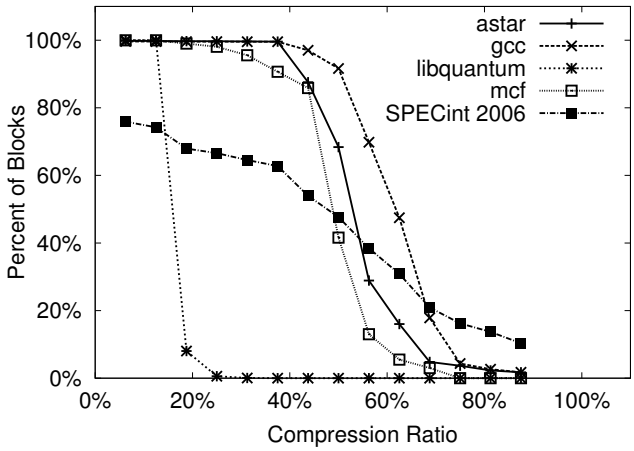


Figure 1.2: Data compressibility in SPEC benchmarks. Data is often slightly compressible, even if high compression ratios are not available. Frequent pattern compression is used [4].

proposes some simple compression techniques for use with COP that can compress over 90% of blocks on average.

COP stores incompressible blocks in memory in their original form, and uses μarch patching to enable a novel mechanism to detect uncompressed blocks without additional metadata. This is important, since, unlike caches where extra bits can be easily added, it is difficult to add compression metadata to DRAM without changing its architecture or reducing the storage capacity. Detection is accomplished by checking each 64-byte block read from DRAM for the valid ECC code words that would have been stored had the block been compressed. To guarantee that this approach will work correctly, incompressible data blocks naturally containing code words cannot be written to DRAM. In these rare cases, μarch patching is employed by retaining the block in the last-level cache so that it will patch the stale copy in DRAM. Simulations show that COP has a negligible impact

on performance while other approaches to protect non-ECC DIMMs can reduce performance by over 10%. A hybrid scheme is also proposed to protect incompressible blocks in addition to the compressible blocks protected by COP. Since this scheme (COP-ER) only requires extra storage to handle incompressible data, it requires about 80% less space than storing ECC for all data.

1.3 Thesis Contributions

The research presented in this thesis makes the following contributions:

- **Proposes microarchitectural patching:** As introduced in Section 1.1, μarch patching exploits standard superscalar structures and mechanisms to avoid undesired accesses. This technique is used by iPatch to improve performance by avoiding a high-latency reliability mechanism. COP also requires μarch patching to distinguish compressible and incompressible data.

- **Proposes iPatch to improve energy efficiency during aggressive voltage scaling:** iPatch uses the store queue, μop cache, and MSHRs to patch L1 cache subblocks that are disabled in accordance with the scheme proposed by Abella et al. [1]. When SRAM failures occur at low voltages, this approach avoids costly L2 accesses, improving energy savings and performance.

- **Proposes COP to protect standard DIMMs with ECC:** COP allows protection of non-ECC DIMMs with negligible performance and storage over-

heads by compressing memory blocks and adding ECC. Unlike traditional compression-based techniques, COP requires only a low compression ratio. Simple targeted compression techniques are therefore proposed for use with COP.

- **Introduces a method to track compression for "free":** A method is proposed to allow COP to distinguish uncompressed data from compressed data without explicitly storing metadata. Instead, COP checks for the parity bits that are included alongside compressed blocks to protect them. If an incompressible data block contains ECC code words, μarch patching is employed and the block is retained in the last-level cache.

- **Evaluation of energy savings with iPatch:** Experiments were performed to compare iPatch to prior state-of-the art proposals. A Monte Carlo approach is used to simulate chips with different fault patterns. Results show that iPatch can achieve energy savings and performance improvements in conjunction with aggressive voltage scaling.

- **Evaluation of COP:** Compression schemes for COP were evaluated and found to be highly effective at providing the required compression ratios. Performance simulations also compared COP to other proposals and demonstrate that COP's performance impact is negligible.

1.4 Thesis Organization

This thesis is organized as follows: Chapter 2 provides context and background for the reader including a discussion of process variations, soft errors, and existing solutions to these problems. An outline of memory system organization and existing compression techniques is also provided. Chapter 3 presents the iPatch technique for improving energy efficiency, and Chapter 4 evaluates its efficacy. Chapter 5 introduces COP, which allows ECC protection of non-ECC DIMMs by leveraging compression. Chapter 6 discusses the methodology to simulate COP and presents the simulation results. Finally, Chapter 7 concludes the document and discusses avenues for future work.

2 BACKGROUND AND RELATED WORK

This chapter provides background and context for the work presented in this dissertation. Two prevalent reliability challenges are outlined, and prior work is discussed. First, the issue of process variation is discussed in Section 2.1, including its impact on the processor execution core and storage structures. Process variations can inhibit voltage scaling by making caches unreliable at low voltages. This section also includes an overview of the subblock disabling technique that is used as a baseline for iPatch. Section 2.2 discusses particle-induced soft errors in logic and storage structures including caches and main memory. In particular, the cost of adding error protection to main memories is described. Finally, Section 2.3 provides an overview of main memory organization and existing compression techniques.

2.1 Process Variations

2.1.1 Overview

Parameter variations can impact both a processor's performance, power consumption, and overall reliability [76]. As Figure 2.1 shows, variations can be classified as either process variations or environmental variations. Environmental variations are typically dynamic variations that happen at runtime, such as temperature fluc-

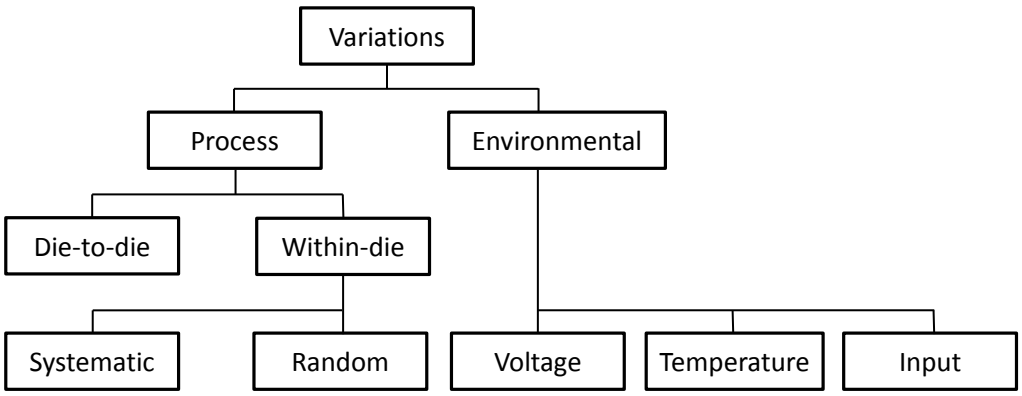


Figure 2.1: Overview of parameter variations that can impact correct circuit operation [76].

tuations or voltage droops. Process variations, on the other hand, are static, and are a consequence of the manufacturing process. Such concerns force designers to incorporate margins and safeguards that take into account all of the possible operating conditions. Furthermore, if testing reveals that the required guarantees cannot be met for a part, it may need to be discarded, impacting yield.

Process variations can be further classified as die-to-die or within-die. Die-to-die variations uniformly affect each chip, and can result from lot-to-lot variations, wafer-to-wafer variations, or occasionally within-wafer variations. The causes of such variations include differing manufacturing temperatures or differences in wafer polishing, which can affect electrical properties [13]. In addition to die-to-die variations, current and future technologies are increasingly susceptible to within-die variations [12]. These variations can be spatially-correlated (systematic) as is the case with lithographic effects on channel widths. Other phenomena such as dopant fluctuations can also result in random within-die variations that affect each

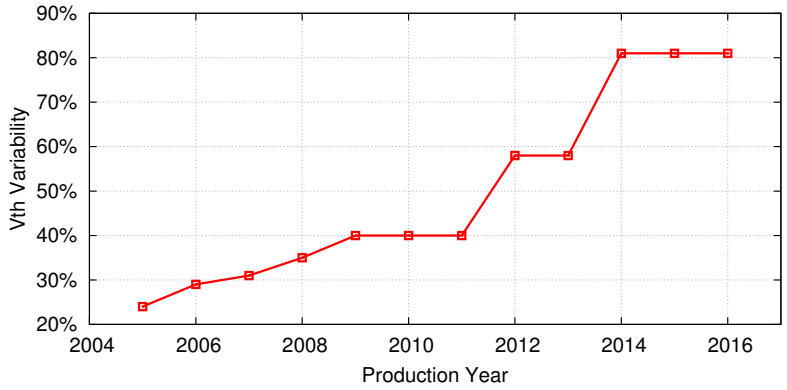


Figure 2.2: The ITRS prediction for transistor threshold voltage variability due to doping effects [31].

device individually [44]. This type of variation has the most significant impact on minimum-sized devices, which are commonly used in SRAM cells to enable high storage density.

Regardless of the variation source, process variation typically impacts the transistor threshold voltage (V_{th}), which is the gate voltage required to form a channel between the drain and source, switching the device "on." A higher V_{th} can make a device slower to switch, while a lower V_{th} can make it faster, but leakier. Since the slowest path determines the overall delay of a circuit, the impact of slower logic gates will dominate, even if some are made faster. For this reason, within-die variations are of particular concern, since they can shift the distribution of fabricated chips, decreasing the mean operating frequency [54]. It has been even been suggested that the impact of within-die variation could counteract the potential gains of technology generations to come [13]. Figure 2.2 shows near-future predictions by International Technology Roadmap for Semiconductors (ITRS) for

V_{th} variability in minimum-sized devices due to doping effects [31]. As shown, this problem is expected to increase substantially.

Process variations can also perturb the delicate balance of SRAM cells like the one shown later in Figure 2.3a [77]. Two different failure modes can occur due to process variation effects, particularly in conjunction with lower supply voltages. To read the value in the cell, the bitlines (BL) are precharged. When the wordline (WL) is activated, one of the bitlines will start to discharge through an access transistor (X0 or X1). If noise on this bitline exceeds the trip point of the inverter (due to V_{th} variation), a *read failure* can occur in which the cell flips. Likewise, a *write failure* occurs if a write operation is unable to toggle the cell when a voltage differential is applied to the bitlines. This can happen if a pull-up transistor (e.g. P0) is stronger than the access transistor (e.g. X0), keeping the node from being discharged. To avoid such failures, V_{min} (the minimum supply voltage allowed) must be set appropriately high [10]. Although techniques such as aggressive clock and power gating can be employed to reduce power, voltage scaling is one of the most effective power-saving mechanisms, since power scales quadratically with voltage [38, 30]. Thus, the most unstable SRAM cell can limit the power savings available to the whole processor.

2.1.2 Addressing Variations in Logic

Though frequency binning can be applied to improve yield, it is less efficient when within-die variations are present. Unlike die-to-die variations, within-die variations decrease the mean performance of fabricated chips, since a chip can only be clocked as fast as its slowest path. This phenomenon has motivated a host of techniques that specifically aim to counter the effects of within-die variations, including both systematic and random fluctuations in delays [19, 41, 42, 64, 73, 74, 75].

Techniques that target within-die variation range from circuit-level to architectural. In the circuit domain, recent work proposes modifying the body bias to compensate for varying threshold voltages [75]. This technique is best suited to compensate for systematic variations, since additional voltage domains are required. Still other techniques propose clocking a processor at its intended frequency and detecting any errors that may occur. In this case, the error rate can be decreased by purposefully modifying the latency of certain circuit paths [64].

As an alternative to disabling slow units, most architecture-level techniques compensate for slower stages or units with modified timing schemes. One possibility is to allow variable-latency functional units so that slow units can take additional clock cycles to complete [41, 42]. Relaxing constraints on operation latency, however, can potentially complicate scheduling and decrease the number

of instructions executed per cycle (IPC). To avoid these issues, other work proposes reducing the processor frequency only when there is sufficient instruction-level parallelism to require both the fast and slower functional units [19]. Yet another alternative is time borrowing, in which specific clock signals can be skewed to allow a slow stage to use some of the time originally allocated to a faster stage [74]. Because techniques to address systematic variations are not optimal for random variations, prior work by the author proposes a bitsliced design in which slow datapath slices can be decommissioned [55, 57].

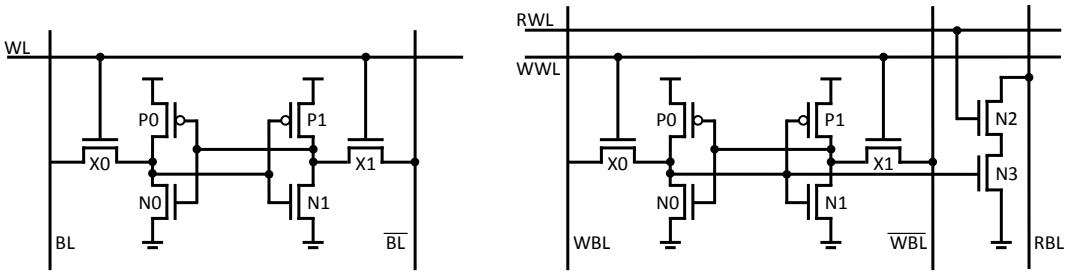
2.1.3 Addressing Variations in SRAM

Modern processors employ a hierarchy of storage structures to keep the most relevant data close to the core to reduce access latency. These structures, including large instruction and data caches, comprise a significant and growing amount of architectural state and die area. Given the large number of storage cells and the increasing impact of process variations as technology scales, it is important to employ techniques that can tolerate or prevent failing cells, particularly in the context of voltage scaling.

Since process variations may limit voltage scaling by causing SRAM cell failures at low voltages, a number of solutions have been proposed. For instance, ARM's big.LITTLE design takes an architectural approach by incorporating both a wide out-of-order core and a smaller, lower-performing core [27]. Although V_{min} is

set high enough for both cores to guarantee that no SRAM cells will malfunction, the smaller core consumes less power because it is less complex. When the DVFS mechanism can no longer reduce the voltage of the big core, work is migrated to the smaller core while power-gating the big core. Although effective, this scheme has some obvious inefficiencies due to the need for a second core and the migration overhead.

Ideally, similar savings could be achieved with a single non-heterogeneous processor if V_{min} could be further reduced while ensuring functional correctness. There are two general approaches to solving this problem. Circuit-level solutions modify transistor-level implementations to design cells that are more resistant to process variation-induced failure. Architectural solutions, on the other hand, can track which cells are failing so that they can be avoided, or make corrections using error-correcting codes (ECC). For large structures that are less latency-sensitive such as L2 and L3 caches, strong error correcting codes (ECC) can be deployed, or more complicated word-substitution approaches can be used to consolidate fault-free words into usable lines [5, 7]. A different solution is required for the L1, however, where high-latency error correcting codes or complex schemes are less feasible. Although the simplest L1 solution is to disable faulty cache lines with failing cells, the resulting increase in program execution time would nullify any potential energy savings from lowering the voltage. The remainder of this section discusses existing circuit and architectural solutions.



(a) Standard 6T SRAM cell. (b) Robust 8T SRAM cell.

Figure 2.3: SRAM cell comparision. While less likely to fail due to voltage scaling, the 8T cell requires more area and reduces array density.

Circuit-Level Solutions

One way to address process variations is through circuit-level approaches that modify the cell design. For instance, using larger devices (upsizing) is an effective measure against dopant fluctuations in both SRAM and combinational logic. Upsizing increases the size of the device channel, making V_{th} less sensitive to dopant variations. Upsizing can be expensive, however, as it can can substantially increase power consumption and decrease array density.

As an alternative to upsizing SRAM cells, modifying the cell design is a more efficient and effective approach. For instance, a robust 8-transistor SRAM cell like the one in Figure 2.3b can be used, which adds an explicit read bitline and wordline to decouple read operations from write operations and avoid accidental read upsets [15]. Though stability is enhanced, the modified cell reduces SRAM density due to the extra devices and wires required [16]. To reduce the overhead of using robust cells, some proposals suggest hybrid cache architectures that integrate

different cells, though this increases design complexity [46, 48]. For even more robustness, there are a number of alternative proposals for robust cell designs requiring anywhere from 10 to 18 transistors, each with varying overheads and reliability [32, 80]. For very large array structures such as caches, however, the overhead of even the 8-transistor cells is expensive, making an architectural solution more desirable.

Architectural Solutions

Because process variation effects determine which cells fail at low voltages, the same cells are always unreliable for a given chip and voltage. Therefore, post-fabrication testing can be used to locate unreliable cells, enabling architectural solutions to work around the failures and guarantee correctness. The most straightforward architectural approach to dealing with faulty cells is to disable all entries in a structure that contain at least one failing cell. In the case of caches, this can mean disabling cache lines and consequentially reducing the number of ways in the affected sets, as proposed in [37]. Although this approach is effective at moderate failure rates, at lower voltages with more failures it can significantly reduce cache capacity and therefore degrade performance and energy efficiency. To better tolerate many failures, strong ECC could be used, but this is not an ideal solution for L1 caches due to increased latency and the read-modify-write operation required for partial writes.

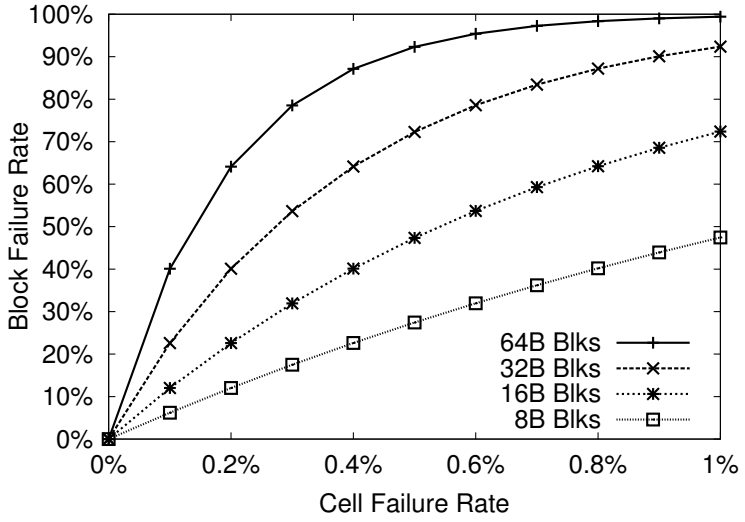


Figure 2.4: Relationship between cell failure rate and block failure rate for different block sizes. A block is considered faulty when it has one or more faulty cells.

Much prior work aims to extract high performance from partially faulty SRAM arrays by exploiting the fact that the failure rate of larger blocks of cells is much higher than that of smaller blocks. In other words, it is more likely that one or more cells will fail in a 64-byte block than a 16-byte block. Figure 2.4 shows the block failure rate for different block sizes and cell failure rates. To take advantage of the lower failure rate of smaller block sizes, prior work divides cache lines into smaller subblocks that can be individually disabled. The remaining valid subblocks can be merged into fault-free lines or used in a way that enables functional correctness.

Table 2.1 is a comparison of the available cache space at different failure rates for various architectural techniques from [1]. The comparison includes generic techniques as well as approaches proposed by prior work. In this evaluation, any failure in a line's tag or metadata requires disabling the entire line. Data is shown

Table 2.1: Available cache space with different disabling schemes (from [1]). Data is shown for a 32KB 8-way cache with 64B lines.

Faulty bit rate	Way disable	Set disable	Line disable [37]	Word disable [77]	Bit fix [77]	Subblock-disable [1]
0.01%	2.7%	63.7%	94.5%	49.7%	74.6%	98.8%
0.1%	0.0%	1.1%	56.9%	45.7%	68.8%	89.0%
1%	0.0%	0.0%	0.3%	3.5%	25.1%	31.2%

for a 32KB, 8-way associative cache. The disabling techniques shown are as follows:

1. **Way disabling.** In a set associative cache, all ways containing at least one failure are disabled. In an 8-way set associative cache, disabling one way reduces the usable cache space by 1/8th.

2. **Set disabling.** In a set associative cache, all sets containing at least one failure are disabled. In an 8-way set associative cache, each set includes 8 lines. Index remapping is required to cache data that would have mapped to the disabled set.

3. **Line disabling.** Ladas et al. suggest disabling lines with one or more failures to allow low-voltage operation [37]. In this approach, each set can have a different associativity, depending on which lines are disabled. The replacement logic must be made aware of which lines in a set are disabled.

4. **Word disabling.** In the word-disable approach proposed by Wilkerson et al., each word in a line can be individually disabled [77]. The fault-free words from two adjacent lines are then combined to form a single functional line.

This approach, while guaranteeing functional correctness, cuts the cache capacity in half.

5. **Bit fix.** Bit-fix reallocates every 4th line to hold pointers and 2-bit values to fix individual faulty bits in 3 other lines [77]. This scheme reduces cache space by a minimum of 25% even at low failure rates, resulting in reduced performance. Bit-fix may also impact cache access latency, as a number of multiplexing layers are required to accomplish the "fixing."

6. **Subblock disabling.** Subblock-disable simply disables faulty subblocks without the complexity of merging or fixing bits [1]. In this approach, some cache lines are left with "holes" due to disabled subblocks. When the core tries to request data from a disabled L1 subblock, the hardware must retrieve it from a lower level cache (e.g. L2).

As shown in Table 2.1, way disabling is the least effective scheme, since it is the most coarse-grained. Of the techniques shown, subblock-disable provides the most available cache space at each failure rate. Studies show that this extra space translates to improved performance at high cell failure rates. This is important because performance losses can translate into increased energy consumption, potentially negating the benefits of voltage scaling. The work in this dissertation builds on subblock-disable due to its simplicity and effectiveness.

Other work addressing cache failures suggests adding additional structures and

complexity to mitigate the problem. ZerehCache and its follow-on Archipelago attempt to more efficiently combine faulty lines to form usable lines [7, 8]. Similarly to word-disable, faulty words or subblocks are disabled, and the valid words are consolidated to form usable lines. The Archipelago approach divides the cache into groups called islands, each of which contains a sacrificial word-line that can be used to repair parts of the other lines in the group. This approach adds significant complexity, however, as an additional layer of indirection is required for each cache lookup. In addition, an optimization problem must be solved to configure every chip post-fabrication.

Some approaches also suggest adding cache assist structures such as buffers or victim caches [2, 37, 43, 47]. In the RVC approach, cache lines are disabled but replaced with victim cache entries in order to provide performance guarantees in a real-time system [2]. The authors also suggest an alternative in which existing buffers are enlarged, allowing the extra entries to be used as replacements for faulty lines. Although this idea bears some similarity to one the mechanisms in this work, the proposal in this dissertation does not require adding buffers or enlarging existing buffers and can support higher failure rates. Other work building on subblock-disable reorders data subblocks before they are written to the cache such that more useful data will not map to disabled subblocks [18]. This proposal incorporates subblock reordering, a fault-free fill buffer, victim caches, and modified prefetching to reduce the performance impact. These changes add

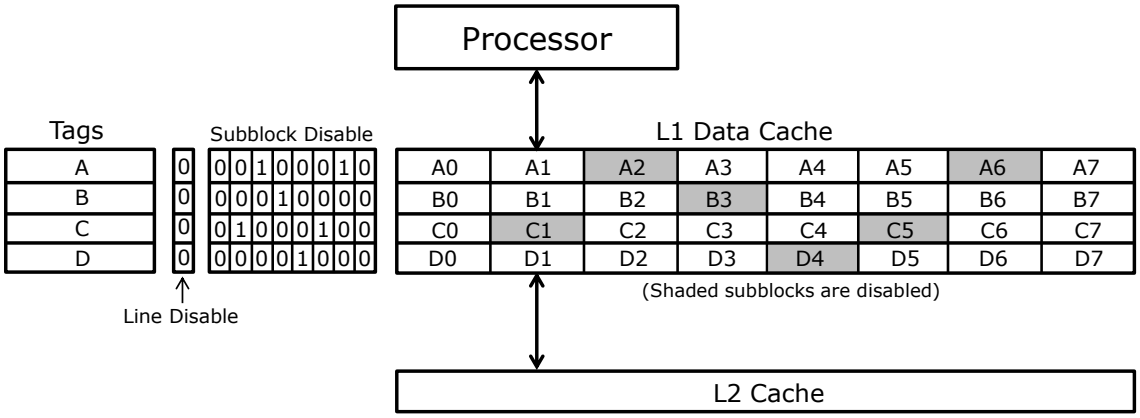


Figure 2.5: Illustration of subblock disabling scheme.

complexity and overhead to the subblock-disable approach, motivating the design in this dissertation as a simpler and effective alternative.

2.1.4 Subblock Disabling

This section discusses the subblock-disable technique [1] in greater detail, since it is used as a baseline in this work. In this technique, the disabled subblocks within a line simply become inaccessible, so a valid copy of this data must be kept in the L2. A fault map containing a disable bit per L1 subblock is required to track whether or not each subblock is disabled (due to unreliable cells) at the current operating voltage, as illustrated in Figure 2.5. Post-fabrication testing can determine which cells will fail due to process variation effects at each voltage. After a voltage change, the subblock disable map for each L1 line can be updated accordingly.

In the subblock-disable approach, read and write operations are handled normally as long as they access the non-disabled portions of a cache line. Accesses to

disabled subblocks are treated similarly to misses and must access the L2, since these requests cannot be serviced by the L1. These accesses are called *false hits*, since there is a tag match in the L1 (just like a hit) but the data must be retrieved from the L2.

When a false hit occurs, the line in question is relocated to a different way in the L1 set. Since, probabilistically, each physical line will have a different pattern of disabled subblocks, relocating a line reduces the chances of consecutive false hits due to repeatedly accessing the same subblock. To relocate the line, the cache replacement mechanism and policy (e.g. LRU) is used. The line in question is first marke in the L1 as most recently used. Then the replacement policy selects a victim line to evict (or simply invalidate if clean). When data is retrieved from the L2, the line is written to the vacated way.

Since too many disabled subblocks in a hot (frequently-accessed) set could produce performance outliers, subblock-disable uses a performance predictability mechanism in which the L1 address-to-set mappings are periodically changed. This mechanism flushes the L1 every 500,000 cycles so data can be remapped to different L1 sets using a simple hash that XORs a counter with the original set index. This approach significantly reduces the performance variation across chips, with the flushing/remapping happening infrequently enough that performance is not affected.

One of the main advantages of the subblock-disable approach is its simplicity. It

incurs low overhead while making the maximum amount of cache space available, while many of the substitution-based approaches may have "wasted" storage. In addition, modification of the critical L1 datapath is minimal, meaning that there will be few complications in high-performance mode with no failing cells thanks to a higher supply voltage. At lower voltages with higher cell failure rates, however, the increase in the number of disabled subblocks can significantly degrade performance due to additional false hits. This effect imposes a limit on subblock-disable's energy-saving potential, since despite the lower voltage, execution time will increase. With simplicity in mind and to enable a wide range of operating voltages, this document describes a solution called iPatch, which builds on the subblock-disable approach to significantly reduce the false hit rate at low voltages by using hardware already common in modern superscalar designs. By improving performance in this manner, energy savings are extended to lower voltages.

2.2 Soft Errors

2.2.1 Overview

Unlike failures due to process variations, transient errors do not necessarily reflect an issue with the hardware. Soft errors from particle strikes are a growing concern as technology scales, since smaller device sizes and lower supply voltages dramatically decrease the amount of charge stored per node. Soft errors occur

when an alpha or neutron particle passes through a semiconductor device, creating electron-hole pairs. If this generated charge is collected by a transistor source or drain, it can interfere with correct circuit operation by causing bit flips in storage elements or transient pulses in combinational logic, either of which can lead to silent data corruption [51].

Soft errors can affect many parts of a system, including the processor and memory (i.e. DRAM). On chip, much previous work focuses on protecting storage elements, where particle strikes can cause bit flips known as single-event upsets (SEUs). It is also possible for soft errors to affect combinational logic in what are known as single-event transients (SETs). In these cases, a transient pulse is created and propagates through the circuit, potentially being latched at the output. Finally, DRAM is also susceptible to SEUs, and error-correcting codes (ECC) are commonly employed for this reason. The remainder of this section gives an overview of these different types of soft errors and prior work on mitigating their impact.

2.2.2 Soft Errors in Logic

Some estimates indicate that the soft error rate (SER) of combinational logic may soon rival that of memory [69]. To address this problem, methods for SET mitigation can be applied at the process, circuit, or architecture level [51]. Silicon-on-insulator (SOI) is an example of a process technology that can somewhat reduce the logic SER. Circuit-level approaches often involve hardening certain nodes against errors

through local duplication [53] or gate resizing [62]. Architectural solutions can either be error-detecting or error-correcting. Since processors often include a mechanism for replaying instructions to recover from misspeculations, a method for soft error detection can employ this framework. Duplication of a circuit can provide error detection and rely on roll back for correction, while more expensive triplication triplication can provide correction without an additional mechanism Instead of complete duplication, another class of SET detection techniques adds dedicated error-detection logic. For example, this logic could duplicate only part of a circuit [50], or check the validity of the circuit output based on a set of rules governing output characteristics. These characteristics can be predetermined [65] or predicted in real time [22].

A third type of SET detection techniques exploits time redundancy. Since a circuit output is guaranteed to be stable during the latching window, any perturbation detected indicates the presence of a transient error [52, 23]. To create this window of stability, additional setup or hold time constraints are imposed. One time redundancy solution involves double sampling using a "shadow" latch is added to sample at a different time than the main latch, but during the stable window. If a transient pulse occurs, the shadow latch may capture the correct value while main latch captures an erroneous value. The shadow latch can be made to sample late as in the Razor approach [24] or early as in the BISER solution by Mitra et al. [49]. To avoid the overhead of the shadow latches, prior work by

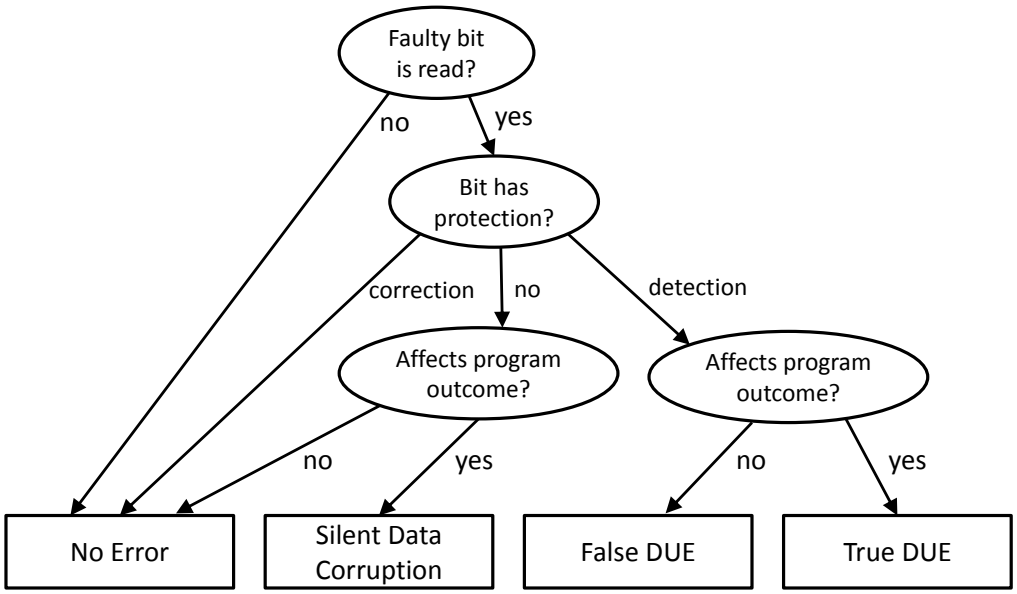


Figure 2.6: Possible outcomes of a bit flip in a processor [51]. DUE refers to Detected Unrecoverable Errors.

the author has also proposed using XOR parity trees to monitor circuit outputs, thereby reducing the number of transition detectors required [56].

2.2.3 Soft Errors in Storage Arrays

Modern processors include many storage arrays to facilitate the efficient processing of large data sets. To reduce memory access latencies, large SRAM caches ranging from kilobytes to megabytes in size are typically employed. These caches are usually backed by gigabytes of off-chip DRAM storage. All of these stored bits are potentially vulnerable to soft errors, and this vulnerability is increasing as systems demand more and more storage [9]. Soft errors that flip bits in unprotected or under-protected storage arrays may cause silent data corruption, as shown in

Figure 2.6. Silent data corruption occurs when an unprotected faulty bit is read that affects the program outcome. If error protection is present it can be in the form of error correction, detection, or a combination of the two. If a bit is flipped but error correction is present, an error is prevented. If only error detection is available (not correction), the failure results in a detected unrecoverable error (DUE).

In arrays, the standard solution to this reliability concern is to employ error-correcting codes (ECC) that can correct bit flips. In both SRAM and DRAM, employing ECC incurs some overhead, since additional parity bits must be stored. Although there are many different types of codes that can be used to detect and correct soft errors, the most widespread are single error correcting, double error detecting (SECDED) codes. Of these SECDED codes, (72,64) codes are commonly used, where the total code word is 72 bits long and 64 of those bits are data [29]. This code therefore stores one byte of check bits for every 8-byte word for a storage overhead of 12.5%.

Protecting Caches

There are various approaches to reduce the cost of error protection in on-chip caches. For instance, wider codes can be used, although such codes may reduce reliability by spreading their error correcting capability over more bits. If this tradeoff is acceptable, caches are well-suited for this optimization since they store data in blocks (e.g. 64 bytes). To further reduce costs, a simple parity bit scheme

can be used for write-through L1 caches. Since a write-through design guarantees that the data in the L1 is "clean," meaning that another copy exists in another cache or memory, L1 cache blocks that fail the parity check can be discarded. Prior work also suggests a method for using parity protection in non-write-through L1 caches [45]. This approach, called the correctable parity protected cache (CPPC), is attractive because SECDED can complicate L1 writes by requiring a read-modify-write operation to update the ECC. The CPPC adds two XOR registers into which all data added and removed from the cache is XORed. On a parity failure, correction is performed via the time-consuming process of XORing all cache data with and the two XOR registers.

Another way to reduce ECC overheads in caches is to exploit an existing cache compression scheme. For instance, prior work by Chen et al. has suggested exploiting the unused fragments inherent in compressed last-level caches (LLCs) to hold ECC parity bits [17]. This approach requires storing extra bits along with each cache line to track the compression type and parity bit locations. This scheme is less useful in the case of main memory, since DRAM standards are relatively inflexible, making it more challenging to store the required metadata.

Protecting Main Memory

When ECC is applied to DRAM, special dual-inline memory modules (DIMMs) are typically required to facilitate storage of the ECC check bits. Such ECC-enabled

DIMMs add an extra chip or chips to each rank to store ECC check bits, and must be supported by the memory controller. For instance, a standard $\times 8$ DIMM uses ranks composed of 8 chips each, while an ECC DIMM adds a 9th DRAM chip to each rank. This configuration can easily support the previously-mentioned (72,64) SECDED code [29]. Adding the extra DRAM chip makes ECC DIMMs more expensive than their standard counterparts, however, in terms of the up-front cost as well as power consumption.

To reduce the costs of error protection, prior work has suggested approaches to improve the resilience of non-ECC DIMMs. Such approaches allocate dedicated portions of main memory to hold ECC metadata. For instance, Yoon et al. propose Virtualized ECC, which allocates full memory pages for ECC [79]. When Virtualized ECC is used with a non-ECC DIMM, each data block retrieved from DRAM requires an additional read to retrieve the ECC check bits. To locate the ECC page containing the check bits, a page-table-like structure is used. Finally, to avoid the high cost of an ECC address table walk, a 2-level ECC address translation cache is employed.

There are a number of downsides to this in-memory ECC storage approach. First, it significantly reduces the overall usable memory space. For instance, if (72,64) SECDED is used to protect an 8GB main memory, 910MB must be reserved for ECC bits. The DRAM accesses to retrieve ECC check bits can also reduce performance by increasing contention and access latencies. If a page-based scheme

is used, extra hardware is required for ECC address translation. To avoid this translation hardware, an alternate implementation could dedicate a contiguous region of physical memory to ECC, but the storage and performance overheads would remain.

To reduce the latency overheads of accessing ECC metadata stored in non-ECC DIMMs, prior work also suggests distributing the metadata throughout memory so that it is collocated in the same DRAM row as the data [81]. With an open-row policy, this "embedded ECC" configuration can improve the ECC access latency, although the same storage overhead as Virtualized ECC is imposed. When embedded ECC is used, other work has suggested an optimization using memory compression so that when a 64-byte block is compressible, its parity bits and data can be stored in 64 bytes, avoiding the need for a second memory access for ECC [67]. This approach is only a performance optimization, however, and the same amount of space must still be reserved for ECC as embedded ECC in case of incompressible blocks.

For DRAM systems requiring very high reliability, chipkill codes can be employed [20]. These codes are able to correct whole-chip failures and typically rely on ECC DIMMs, although they can also be used in a Virtualized ECC implementation (possibly increasing overheads). SECDED can be used to implement chipkill by interleaving such that each bit in a code word comes from a different chip. If $\times 4$ DIMMs are used, each access become four times wider. Chipkill can also be

implemented with more complex symbol-correcting codes such as Reed-Solomon codes [25]. If a 4-bit Reed-Solomon code is used with ×4 DIMMs, a 144-bit channel is required to access 36 chips in parallel for each burst [35]. Some modern systems also use 8-bit Reed-Solomon codes in conjunction with ×4 DIMMs and more standard 72-bit channels. In this approach, only 18 chips (72-bits) are accessed in parallel, with each chip providing half of an 8-bit symbol on each access [35].

2.3 Memory Organization and Compression

This section provides an overview of main memory organization. Because the work in this dissertation relies on memory compression, background on various memory compression techniques is also provided.

2.3.1 Memory System Organization

Main memories are typically implemented using Dynamic Random Access Memory (DRAM) technology. DRAM chips provide high-density storage, since each bit cell requires only one access transistor. Bits are stored capacitively, requiring periodic refreshes and a write following each read, as reads are destructive. DRAM chips commonly have either 4 or 8 data pins (×4 or ×8), although other widths are possible [35, 79]. Each DRAM chip contains a number of banks, each of which can only have a single "row" open at a time, due to the destructive nature of DRAM

reads.

DRAM chips are organized into ranks, with each chip providing a number of bits during each access. For instance, with a 64-bit data bus, eight $\times 8$ chips can comprise a rank with each chip contributing 8 bits towards the total 64. In a system using $\times 4$ DRAM chips, 16 chips are needed per rank. Because memory is generally accessed at the granularity of cache blocks (often 64 bytes each), multiple bus cycles are required to transfer a block. For example, with a 64-bit bus and 64-byte blocks, eight beats (comprising a burst) are required to transfer a block. Multiple ranks can be integrated into a Dual-Inline Memory Module (DIMM). Finally, a memory system can include multiple DIMMs and data channels, with each channel dedicated to one or more DIMMs.

2.3.2 Memory Compression

This section discusses various compression schemes. Although some of these approaches were originally proposed for compressed caches, they can all be adapted for main memory.

Frequent Pattern Compression

Frequent pattern compression (FPC) uses a 3-bit prefix for every 32-bit word to encode common patterns such as repeated bytes or sign extended values [4]. Table 2.2 shows the different 3-bit prefixes in FPC and the patterns that they

Table 2.2: Encoding for Frequent Pattern Compression (from [3]).

Prefix	Pattern Encoded	Data Size
000	zero run	3 bits (# words)
001	4-bit sign-extended	4 bits
010	one byte sign-extended	8 bits
011	halfword sign-extended	16 bits
100	halfword padded with a zero halfword	16 bits
101	two halfwords, each a byte sign-extended	16 bits
110	word consisting of repeated bytes	8 bits
111	uncompressed word	32 bits

match. For data that contains these patterns, FPC is very effective and produces compression ratios on the order of 2×. For data with limited compressibility, however, FPC is ineffective due to the metadata required. For instance, encoding an incompressible 512-bit block using FPC would require 560 bits, thus incurring negative space savings.

Base Delta Immediate Compression

Base delta immediate (BDI) compression can efficiently compress a cache line containing values that are similar in magnitude [58]. It works by storing each data block as a base value and a set of deltas. To decompress the block and retrieve the original values, each delta is simply added to the base. Because fewer bits are required to store the deltas, the block can be significantly compressed. For instance, if the base is a 4-byte word and each delta is one byte, then a 64-byte block can be compressed into only 19 bytes, compressing the data by 70%. To be more effective, BDI can support up to two bases per block where one base is specified and the

other is an implicit zero. For flexibility, variants with different word sizes can also be supported.

Zero Content Augmented Compression

Zero content augmented (ZCA) caches are able to efficiently store all-zero cache lines [21]. This approach incurs very low overhead for compression, since it simply detects and tracks null 64-byte blocks. The coarse granularity of this scheme, however, makes it ineffective for many workloads that do not contain large strings of zeros.

Frequent Value Compression

Frequently value compression attempts to detect and exploit a few frequently-occurring values by storing them using fewer bits [78]. In this approach, the same frequent values are extracted from all data blocks, making this scheme ineffective for blocks that do not contain these values. Profiling may also be required to tailor the frequent values to a particular application.

Compressing Memory

In both caches and memory, simplicity in a compression scheme is attractive in order to minimize latency (and power) overheads. Unlike caches, main memory can introduce addressing complications when compression is applied, since compression can alter data alignment. In particular, if 64-byte blocks are compressed

to different sizes, reading any particular block requires knowing the sizes of all the others before it. To solve this problem, Pekhimenko et al. propose compressing all 64-byte blocks in the same memory page (e.g. 4kB region) by the same amount [59]. Using this approach, it becomes trivially easy to locate a compressed block if the starting address of the page and compressed block size are known. In this dissertation, the addressing problem is avoided by maintaining the alignment of all 64-byte blocks. Because the primary goal in this work is to enhance reliability, the space created by compressing a block is used to store error-correcting metadata.

2.4 Summary

The chapter discusses two important reliability challenges for future microprocessors. The effect of process variation on SRAM caches is first described. When voltage is reduced to save power, this effect becomes particularly prevalent and can cause bit cells to fail. A prior approach to address this problem through subblock disabling is presented. Though effective, this approach can sacrifice performance and reduce energy savings when more subblocks are disabled at lower voltages. The issue of soft errors is also discussed, in which particle strikes can cause silent data corruption in storage arrays. Prior approaches to protect non-ECC DRAM DIMMs by embedding parity bits are outlined. These approaches generally reduce the memory space or impose a performance penalty. Finally, DRAM organization and potential compression techniques are described.

3 FAULT OCCLUSION WITH IPATCH

This chapter proposes iPatch, a technique that uses μarch patching to efficiently tolerate failing SRAM cells in L1 caches. This technique can be used to improve performance and energy savings at low voltages. Section 3.1 provides an overview of iPatch, which can exploit various existing superscalar structures to patch failures in the instruction and data caches. The remaining sections discuss each of the iPatch mechanisms in detail. Section 3.2 describes patching the i-cache using a micro-op cache. Section 3.3 discusses patching with miss status handling registers, which can be applied to both the i-cache and d-cache. Section 3.4 proposes a third mechanism using the store queue to patch the d-cache. Finally, Section 3.5 discusses combining these mechanisms, and Section 3.6 summarizes the chapter.

3.1 Overview

As previously discussed, process variations can cause SRAM cells to fail when voltage is reduced to save power. Although tolerating these failures is desirable to enable energy savings, doing so can be complicated in performance-critical L1 caches. iPatch uses μarch patching to hide these failures from the processor, allowing additional energy savings. It uses the micro-op cache, miss status handling registers, or the store queue to patch failures in L1 caches. Because iPatch relies on standard processor structures and data movement mechanisms, it is very

low-cost and does not interfere with high-voltage performance. With a mechanism to compensate for failing cache cells, large cache arrays can be designed with dense 6T cells instead of more expensive robust cells. The smaller structures exploited by iPatch to create the patching effect can then be protected with robust cells much more inexpensively.

iPatch can be employed as a stand-alone mechanism or to augment a cache disabling approach. As an example of a stand-alone implementation, consider a case in which there is a single failing cell in the L1 data cache. If a load to the associated address is forwarded from the store queue, the faulty cell will not be read and correctness is guaranteed. As long as iPatch maintains the store queue entry, the failure is patched and will not cause an error. A similar patching behavior can occur when the processor front end reads an instruction from a small micro-op cache instead of accessing the L1 instruction cache. In this stand-alone approach, patches must stay locked in their associated structures to ensure functional correctness. For this reason, the stand-alone approach is not scalable as more failures arise at lower voltages, since a structure like the store queue cannot hold enough entries to patch all faults in the much-larger L1.

To allow scaling to lower voltages, the iPatch implementation described in this chapter uses μarch patching to supplement the subblock-disable technique previously described in Section 2.1.4. Remember that when using subblock-disable, accesses to disabled subblocks (false hits) result in costly L2 requests that can

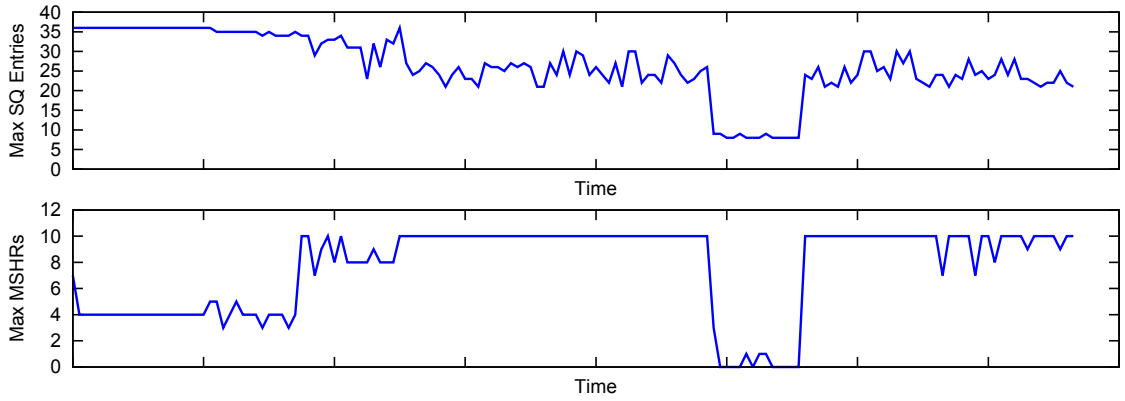


Figure 3.1: Maximum store queue and data cache MSHR usage over time when executing bzip2.

degrade performance and inhibit energy savings. By patching the most useful subset of disabled subblocks, iPatch seeks to avoid these false hits as much as possible. Because all disabled subblocks cannot be patched simultaneously, iPatch falls back on the subblock-disable mechanism when necessary. By combining various patching mechanisms, iPatch improves performance, thereby unlocking additional energy savings.

iPatch works by distinguishing between *patch* and *non-patch* entries in the various buffers and structures used for patching. A *patch* entry is one in which the subblock in the cache that corresponds to the entry's address is disabled. A *non-patch* entry's data, on the other hand, is valid in the L1 and can be read without triggering a false hit. Due to the benefits of *patch* entries over *non-patch* entries, iPatch attempts to create and promote *patch* entries when possible. The processor's data forwarding mechanisms then automatically accomplishes the patching by accessing data in the *patch* entries instead of the cache. One additional observation

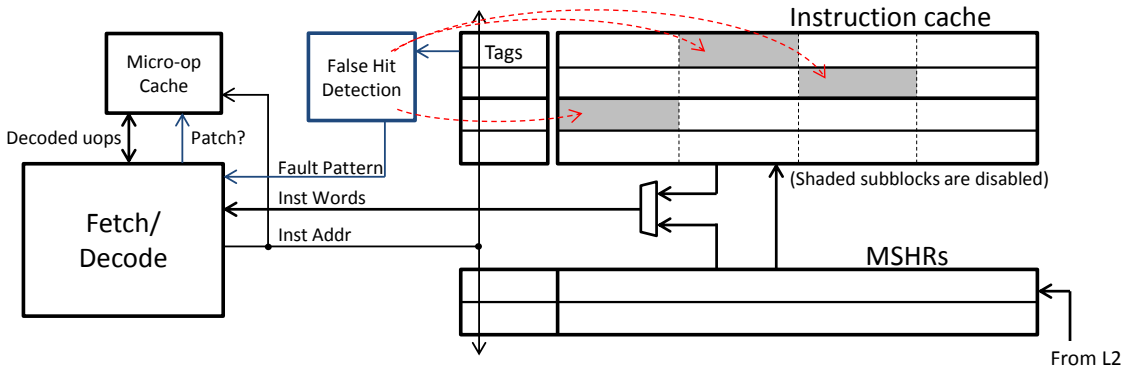


Figure 3.2: Processor front end with fault patching additions. In order of preference, instructions are read in decoded form from the micro-op cache or undecoded from the L1 MSHRs or L1 instruction cache. The false hit logic tracks faulty sections of the cache and forces a miss (false hit) if the front end tries to access these subblocks.

inspires the proposed iPatch implementation: most buffers that can hold *patch* entries are not fully utilized all of the time. As an example of fluctuating resource use over time, Figure 3.1 shows the maximum number of allocated store queue entries and MSHRs while executing bzip2. During the phases in which resource usage is lower, iPatch can repurpose the unused entries as patches.

3.2 Patching with Decoded Micro-ops

Many modern processors employ micro-op caches or buffers that are very effective at saving fetch and decode power through clock gating. As shown in Figure 3.2, decoded micro-ops will be read from the micro-op cache if they are present. On a micro-op cache miss, instructions are read from the i-cache or its associated MSHR buffers. After they are decoded, the instructions are saved to the micro-op cache for later use. For iPatch, if an instruction fetch hits in the micro-op cache,

the micro-op cache entry may act as a patch for the instruction cache, since the instruction cache will not be accessed. Since not all parts of the i-cache will always be patched, the false hit detection logic for subblock-disable keeps a fault map (bit vector) to track the disabled subblocks in each line and forces a miss if the front end tries to access one.

In the iPatch implementation, when the fetch stage reads instructions from the i-cache, the i-cache also provides a fault pattern of the disabled subblocks within the block of instructions read. If the size of the instruction block requested is less than or equal to the i-cache subblock size, the fault pattern is simply a single bit indicating whether the data being provided is mapped to a disabled subblock. If this is the case, the instructions will have been read from the L2 or an i-cache MSHR buffer. In either case, the cache management logic determines the current or destination way in the i-cache and is able to forward the associated fault pattern. Using this fault pattern information, the decoder can track which micro-ops are derived from disabled i-cache subblocks. When these micro-ops are written into the micro-op cache, the destination entry is then designated as a *patch* entry using an extra per-entry *patch* bit.

Using the extra bit to indicate which micro-op cache entries are patches, iPatch modifies the replacement policy to favor keeping these entries. This modified policy must be carefully chosen, since the micro-op cache can provide both energy and performance benefits. If we only allowed *patch* entries in the micro-op cache,

for instance, both of these areas could be negatively affected. As the performance penalty of false hits (and, by proxy, energy) rises at higher cell failure rates, the importance of retaining patches increases. Thus, iPatch attempts to find a balance between the number of *patch* entries and *non-patch* entries. This is accomplished using a patch threshold parameter that indicates the target percentage of *patch* entries in each μop cache set. If the number of patch entries falls below the threshold, the replacement decision favors replacing a non-patch entry. If the number of patch entries exceeds the threshold, the replacement decision (e.g. LRU) is made without any modification. The patch threshold can be tuned for optimal performance at each voltage point and its associated cell failure rate.

3.3 Patching with MSHRs

Each cache is equipped with a number of miss status handling registers (MSHRs), as depicted in Figure 3.2. MSHRs are used to track outstanding misses that must be serviced from a lower-level cache or memory. They allow non-blocking memory requests by storing the information needed to continue the operation once the data is returned. Each MSHR has an associated fill buffer entry to hold the data before it is written into the cache. Since cache line data may not be furnished by the lower-level cache all at once, each MSHR contains valid bits to track which subblocks are currently valid in the associated fill buffer. Once all subblocks are valid, the line is written into the cache from the buffer and the MSHR is freed. For

best performance, MSHRs are able to service loads from partially accumulated cache blocks or blocks that have not yet been written to the cache. A load will check its address against the block address stored by the MSHR to see whether the data required is currently valid in the buffer. On a match, the load can be serviced directly from the fill buffer. Otherwise, the load misses, allocating a new MSHR or adding itself to an existing MSHR.

iPatch takes advantage of the ability to service loads from MSHRs in order to use MSHRs as patches for faulty lines in the cache. Unlike the μop cache implementation in which a *patch* entry patches only part of a cache line, an MSHR can be used to patch an entire cache line. This approach is highly efficient, therefore, in the case of cache lines with multiple disabled subblocks.

To allow intelligent patch management decisions, iPatch augments each MSHR with a patch bit and a reference bit, as shown in Figure 3.3. The patch bit indicates whether or not each entry is a patch. The reference bit is set when a load is serviced from the MSHR, and allows iPatch to find a not-recently-used patch entry to invalidate in case a new MSHR is needed but the buffer is full. The reference bits of all patch entries are reset if all patches have been referenced.

To use MSHRs as patches, iPatch simply keeps the entries and data valid after a cache miss/fill is complete. On a cache miss, an MSHR is allocated to track the request status. Once the entire cache line has been accumulated by the MSHR, it is written into the cache, as usual. Instead of being freed, the MSHR entry is retained

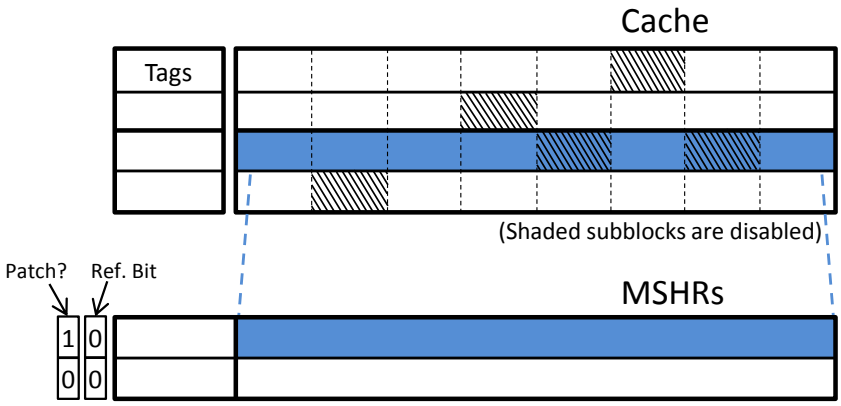


Figure 3.3: MSHR patching illustration. For lines with many disabled blocks, a duplicate fault-free copy can be retained in an MSHR to service loads.

and its patch bit is set to denote that it is not actively tracking an outstanding request. As depicted in Figure 3.3, a copy of the same block is now present in both the MSHR and the cache. In this scenario, iPatch requires that a load hit in an MSHR will take precedence over a cache hit so that as many loads as possible will be serviced from MSHR buffers, avoiding disabled cache subblocks. Patching in this manner does not reduce performance, since the buffer access latency is the same as the cache hit latency.

Because MSHRs are retained after they would normally be deallocated, with the iPatch approach, more MSHRs are in use at any given time. However, the patch management policy guarantees that patches will not reduce performance or cause deadlock. Because the data from patch entries has already been written into the cache, a patch entry can simply be invalidated if all MSHRs are in use and the cache needs to allocate an MSHR to handle a new miss. In this scenario, the MSHR patch to overwrite is selected using the previously-mentioned reference

bits. Once the patch MSHR is overwritten, the copy of the line in the cache is now unpatched, exposing its missing subblocks to the execution core. By taking this approach of keeping entries longer than usual but immediately invalidating them as necessary, iPatch does not reduce the number of MSHRs available to handle misses compared to a system without iPatch. Since the cache never blocks due to patch entries, performance is not reduced. Although this work looks at L1 caches, the MSHR patching approach could be applied to any cache, since it patches whole cache lines.

iPatch also invalidates patch entries when a line with an MSHR patch is written to. Because the L1 cannot write directly into its own MSHRs without an additional write port, the fill buffer data no longer matches the cache after a write, so the patch must be invalidated for correctness. A more aggressive incarnation could also reset the MSHR subblock valid bits depending on which section of the line was written. In either case, this policy does not impact the i-cache, since there are no writes to cause this invalidation.

3.4 Patching with SQ entries

The load and store queues in an out-of-order processor allow memory instructions to be executed out of order while maintaining program-order dependencies [68]. They are also responsible for managing and squashing speculative memory accesses, if necessary. The store queue holds pending stores that have not yet been

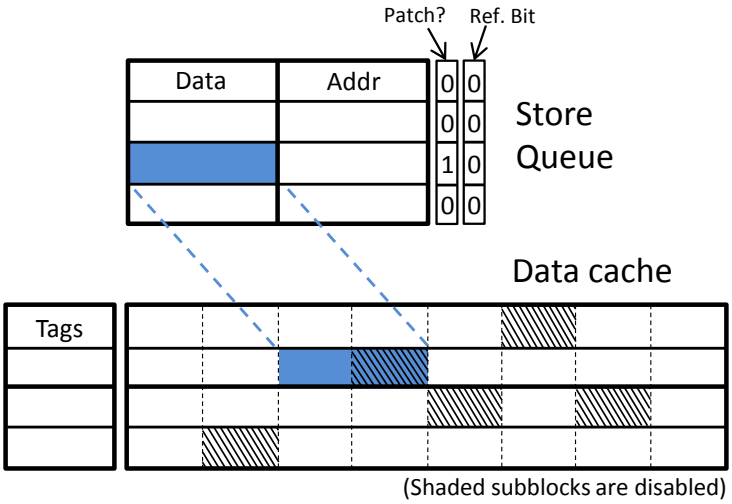


Figure 3.4: Illustration of using a store queue entry to patch a faulty cache subblock. Loads to the highlighted block are serviced from the store queue and not the partially faulty copy in the cache.

written to the data cache. Each load instruction is assigned a store color corresponding to the most recent store instruction (in program order). Using its store color, a load checks the store queue for older stores to the same address. If a match is found, data is forwarded from the matching SQ entry.

Just as μop cache entries can patch the i-cache, store queue entries can be used to patch the d-cache by exploiting the store-to-load forwarding mechanism. Figure 3.4 demonstrates how a store queue entry can act as a patch for faults in the data cache. The highlighted data exists in both the data cache and the store queue due to a store that has not yet completed. As shown, the cache copy overlaps with a faulty subblock and cannot not be reliably read from the cache without an L2 access. Since loads to that address will be forwarded from the store queue, however, the disabled L1 subblock is no longer a concern.

A store is considered to be completed after its data is written to the d-cache. In most designs, these completed stores are then removed from the SQ. iPatch, modifies this behavior to keep some completed stores in the queue. Normally, there would not be much utility in keeping such entries after they are written back due to the equivalent latency for store-to-load forwarding and data cache accesses, since the structures are searched in parallel. For iPatch, however, keeping completed stores provides considerable benefit if these entries are patches. Furthermore, allowing completed SQ entries does not degrade performance or cause deadlock, since if a new SQ entry must be allocated, a completed entry can immediately be invalidated. Because allowing completed store queue entries implies having multiple copies of the same data, the store queue is kept coherent with the rest of the cache hierarchy.

By allowing completed stores in the store queue, some parts of the data cache will be patched as a side effect. Relying solely on this natural patching, however, does not provide the maximum benefit. Depending on the SQ implementation, iPatch can be more proactive about keeping and creating patch entries. Although the store queue is traditionally implemented as a circular buffer, prior work has developed practical approaches to managing SQ entries in an unordered fashion [28, 66]. Unordered store queue management allows late entry allocation, reducing the pressure on the queue and enabling larger instruction windows. This technique also benefits weakly-ordered ISAs (e.g. ARM) by allowing a unified SQ

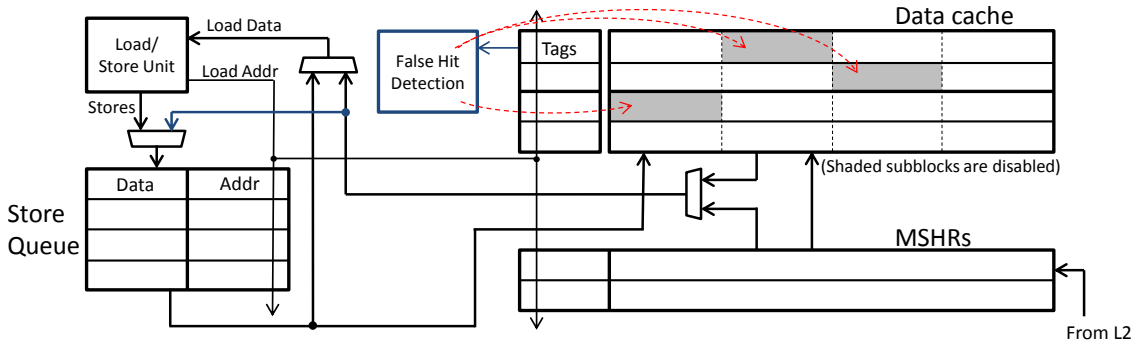


Figure 3.5: Load/store hardware with fault patching additions. On an address match, loads are serviced from (in order of preference) either the store queue, the L1 MSHRs, or the L1 data cache. The false hit logic tracks faulty sections of the cache and forces a miss (false hit) if a load tries to access these subblocks.

from which completed stores to non-overlapping addresses can be removed out of order. For optimal efficiency, iPatch can benefit from unordered SQ management by removing completed non-patch entries out of order while allowing patch entries to be persistent. An alternative to a unified store queue that still benefits from out-of-order store write-back could combine a circular store buffer with an additional unordered committed store buffer. In this implementation, iPatch can still perform aggressive d-cache patching by managing completed entries in the committed store buffer.

Patches can be inserted into an unordered store queue as lines are loaded into the L1 cache. To enable this mechanism, iPatch adds an additional path allowing the data cache MSHRs to write directly into the store queue, as depicted in Figure 3.5. To avoid adding SQ ports, insertion can be performed when the write port is not otherwise needed. Cache replacements are performed in the L1 as usual,

except that patches are written to the SQ from the fill buffer before the associated MSHR is freed. Once the destination way is determined during a replacement, the fault pattern stored by the subblock-disable logic is used to determine which SQ entry-sized segments of the line must be inserted into the SQ as patches.

When patch entries are inserted into the store queue, they are marked as completed entries, which are, by definition, the oldest "stores" relative to others in the queue. iPatch adds two extra metadata bits to each store queue entry to facilitate resource management. One of the extra bits is a "patch" bit, which indicates that the entry is both a patch and has completed. The second extra bit is a reference bit, which is set when the entry forwards its data to a load. These bits are used to determine which completed entry to free (through invalidation) when a new SQ entry needs to be allocated. In order of decreasing preference, iPatch prefers to free untouched/non-patch entries first, followed by touched/non-patch, untouched/-patch, and finally touched/patch entries. If all patch entries have their reference bits set when searching for a replacement candidate, all reference bits are reset. Finally, if an older entry was a patch (according to its "patch" bit) a newly-completed store to an overlapping address inherits this status before the older entry removed. If no such overlap exists when a store completes, iPatch cannot know if it is a patch without checking the d-cache. This potentially complicated check is avoided by assuming that such an entry is not a patch.

3.5 Combining iPatch Mechanisms

Two iPatch techniques have been presented for both the d-cache and the i-cache. Since they are orthogonal, these techniques can be combined. In the i-cache, μop cache patching can be combined with MSHR patching. Likewise, in the d-cache, SQ patching can also be combined with MSHR patching.

One possibility when combining the two mechanisms is to partition the lines in the cache such that a subset are patched with only one mechanism (e.g. μop cache patching) and the remainder are patched only with the other mechanism (e.g. MSHR patching). Experiments to this effect were unable to find a partitioned configuration that performed better than one applying both mechanisms to all lines. To combine approaches in the i-cache, lines that are MSHR-patched upon insertion still send their fault pattern to the front end for tagging in the μop cache, even though the data is read from the fault-free MSHR. On the d-cache side, the same approach is adopted when adding patches to the store queue. iPatch provides the maximum benefit when both mechanisms are implemented in the i-cache and d-cache.

3.6 Summary

This chapter presents iPatch, a mechanism that can improve energy savings when L1 cache disabling is used to tolerate SRAM failures from voltage scaling. iPatch intelligently manages μop cache entries, store queue entries, and MSHRs to exploit μarch patching and avoid costly L2 accesses. Certain entries are flagged as patches, allowing the management logic to favor keeping them due to their performance-enhancing potential. In doing so, iPatch mitigates IPC degradation due to disabled subblocks, making additional energy savings accessible. iPatch mechanisms can be combined in both the instruction and data caches for maximum benefit.

4 EVALUATION OF IPATCH

This chapter describes the methodology for evaluating iPatch and discusses the simulation results. Section 4.1 provides an overview of the fault model and describes the approach used to model random process variations and voltage scaling. Section 4.2 describes the simulation infrastructure and methodology in detail. Section 4.3 presents and discusses the simulation results. Finally, Section 4.4 summarizes the chapter.

4.1 Fault Model

iPatch provides performance and energy benefits when L1 SRAM cells fail at very low supply voltages, requiring portions of the L1 to be disabled. Process variations randomly cause the threshold voltage to vary across devices. Due to this effect, for a given chip, certain cells will be predisposed to fail at low voltages. Furthermore, for each chip, the same cells will be unreliable at each low voltage point, with more cells failing at lower voltages. These unreliable cells can be located through post-fabrication testing. To quantify the benefits of iPatch, a number of chips were simulated with different faulty cell locations. This Monte Carlo approach is similar to the methodology used by previous work [1, 37].

To simulate a given operating voltage, the SRAM failure rate for that voltage was determined using data from [36], as shown in Figure 4.1. Monte Carlo simulations

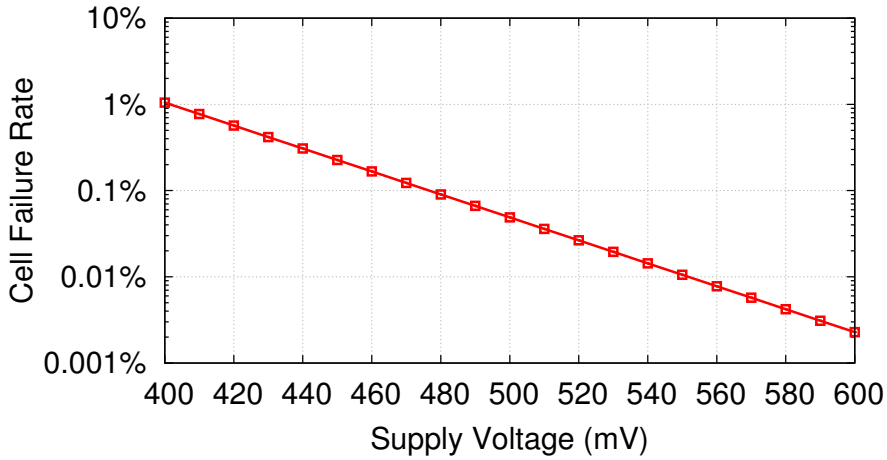


Figure 4.1: SRAM cell failure rate as a function of voltage for 32nm technology [36].

were then performed in which the failing L1 cells were randomly chosen based on the cell failure rate for the voltage. For each chip (and voltage) simulated, a random number was generated for every L1 SRAM cell and compared to a threshold to determine if the cell should be modeled as faulty. The faulty cells chosen in this manner were considered to remain unreliable for all benchmarks run on each simulated chip at a given voltage. For functional correctness, the baseline (subblock-disable) and iPatch disable all L1 subblocks containing one or more unreliable cells. Because performance will vary somewhat depending on fault locations, a number of chips are simulated for each voltage/configuration by using different random seeds to select different faulty cells. The L1 address remapping mechanism discussed in Section 2.1.4 was implemented to significantly improve the performance predictability of the chips simulated. Other non-cache SRAM elements like tags and buffers (e.g. SQ and MSHRs) were assumed to be

Table 4.1: Simulator configuration

Category	Configuration
OoO Core	Fetch/commit: 4-wide Issue: 7-wide Reorder buffer: 192 entries Instruction queue: 54 entries Physical registers: 160 INT/144 FP Load queue: 64 entries Store queue: 36 entries
Execution Units	Integer ALUs: 3 (1 cycle, 3 cycle multiply) Memory: 2 (1 cycle AGU) FP adder/multiplier: 2 (5 cycles) FP div/square-root: 1 (10 cycles)
Memory/Caches	µop Cache: 32 set/8-way L1 instr: 32 KB/8-way/4 MSHR, 3 cycles L1 data: 32 KB/8-way/10 MSHR, 3 cycles L2: 256 KB/8-way, 12 cycles L3: 4 MB/16-way, 30 cycles Memory latency: 30 ns

implemented with more robust cells, allowing them to remain fault-free. Because these structures are much smaller than caches, switching to more robust 8T cells is relatively inexpensive.

4.2 Simulation Infrastructure

To evaluate the performance and energy benefits of iPatch, the gem5 simulator was used in conjunction with McPAT [11, 39]. A future high-end ARM processor was modeled as detailed in Table 4.1. All relevant buffer structures (i.e. MSHRs and the store queue) were sized to be consistent with modern high-performance architectures such as Intel's Sandy Bridge. A sensitivity study showed nontrivial

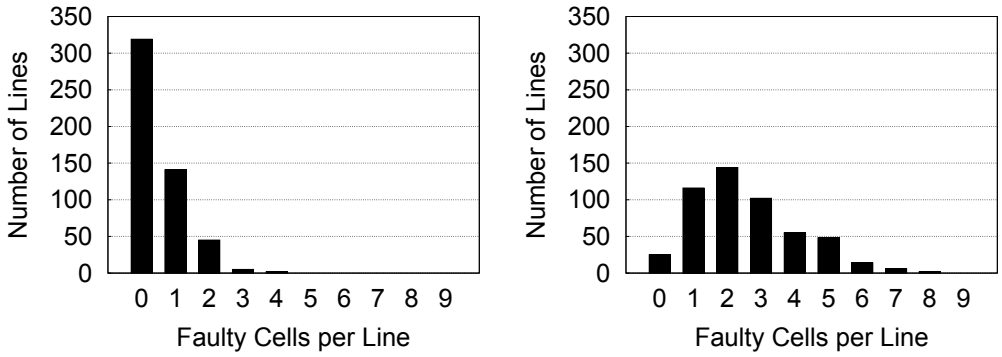
performance impact from downsizing these buffers. Since current ARM processors employ a decoded loop buffer to save front end power, the model assumes that future generations will upgrade this loop buffer to a μop cache as implemented in many other high end designs from other vendors. The μop cache implemented achieves an 80% hit rate on average, consistent with the results in [71]. Gem5's store queue implementation was modified to model an unordered store queue that can complete and remove stores out of order, thereby taking advantage of the ISA's weak memory ordering. The store-to-load forwarding latency was also updated to match the L1 hit latency.

In addition, gem5's cache model was modified to model a write-through L1 cache. This configuration is useful in energy-conscious designs, since it allows the core to quickly enter a low-power state without flushing dirty data from the L1. Likewise, when updating the L1 address mappings for the performance predictability mechanism, there is no dirty data to write back before invalidation. When the predictability mechanism is enabled, the L1 is invalidated every 500,000 cycles so data can be remapped to different L1 sets using a simple hash. For a 1-IPC application, for instance, 200 remappings will occur for every 100 million instructions, incurring a negligible performance impact. For the benchmarks studied, using a write-through L1 also has minimal performance impact when compared to a write-back cache. Note that iPatch would require little modification to work with a write-back cache, which is used by the subblock-disable implementation

described in [1].

All iPatch and subbock-disable simulations model 8 subblocks per cache line that can be individually disabled. No special false hit management is required for writes, since all writes are sent to the L2 by default. A read, however, can trigger a false hit if it attempts to read data from a disabled subblock, requiring data to be fetched from the L2. While waiting on the request, other reads to fault-free subblocks in the same line can still be satisfied as long as the line is not written to. When data is returned from the L2 following a false hit, the L1 line is relocated to a new way selected by the replacement policy, as discussed in Section 2.1.4.

To simulate voltage and frequency scaling, a frequency scaling factor was generated by measuring the frequency of a 24-stage FO4 inverter chain across a range of V values while simulating 32nm devices in HSPICE. A nominal frequency of 3.3GHz was assumed for the simulated processor and scaled using the trend observed in the HSPICE experiment to obtain a voltage/frequency curve for DVFS. McPAT was used to compute power and energy consumption [39]. The execution core was configured to use 32nm low operating power technology, while the L2 and L3 were set to use low static power devices. To simulate DVFS, power was first computed for each trial with McPAT configured for nominal voltage and frequency. Dynamic power was scaled down from the values reported by McPAT using the previously-computed scaling factor. A dummy circuit modeled in HSPICE was also used to compute a leakage scaling factor. The dummy circuit for leakage



(a) 0.1% failure rate (b) 0.5% failure rate

Figure 4.2: Example of faulty cell distribution for a 32KB cache with different cell failure rates.

current modeling consists of a large number gates (INV: 50%, NAND: 30% and NOR: 20% effective widths) where randomly selected input states are applied to each gate to measure the leakage power, as in [70].

To evaluate iPatch, a representative set of integer and floating-point SPEC2006 benchmarks were simulated [61]. The SimPoint tool was used to select a section of 100 million instructions from each benchmark when using the train input set [60]. For various operating voltages, subblock-disable was simulated by itself as a baseline and in conjunction with various iPatch techniques. For the µop cache patching scheme, the patch threshold was tuned for each operating voltage. For each configuration and voltage, a total of 50 chips were simulated using a different random seed for each chip when the generating faulty cell locations. For all results, the mean performance or energy across the 50 chips is reported. Thanks to the performance predictability mechanism, the mean performance across the 50 chips was

determined with a high confidence to be within $\pm 2\%$ of the true mean.

4.3 Results

To evaluate the performance benefits of each iPatch technique, a relatively high cell failure rate of 0.5% was simulated. Figure 4.2 shows example data from one of the Monte Carlo experiments indicating the number of failing cells per line for a 32KB cache with 64-byte lines. As shown, for a failure rate of 0.5%, the majority of lines will contain one or more failing cells. In this scenario, the performance implications of a line-disabling technique would be prohibitive, since 95% of the cache would be disabled. The subblock-disable approach, on the other hand, must disable only around 30% of the cache space assuming 8-byte subblocks are used. Even so, the subblock-disable approach can noticeably degrade performance at this failure rate due to false hits, which have an impact similar to that of L1 misses. The results in this section show how iPatch is able to improve the performance of subblock-disable by reducing the effective L1 miss rate. First, benefits in the instruction and data caches are examined separately, and then overall performance and energy improvements are presented.

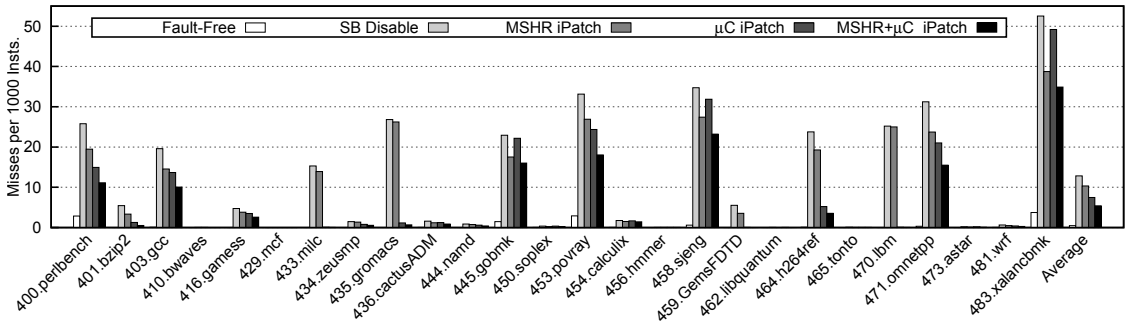


Figure 4.3: Instruction cache miss rate with different iPatch techniques. False hits are counted as misses. The fault-free miss rate for most applications is nearly zero.

4.3.1 Instruction Cache Patching

To evaluate the benefits of iPatch in the instruction cache, a failure rate of 0.5% was simulated in the i-cache only, leaving the d-cache fault-free. Figure 4.3 shows the i-cache miss rates for subblock-disable and various iPatch configurations. The fault-free i-cache miss rate is also included for comparison, with all miss rates presented as misses per 1000 instructions (MPKI). As shown, most of the benchmarks simulated have very low i-cache miss rates since they are composed of loops containing a relatively small number of instructions. For many benchmarks, the majority of instructions can be supplied from the µop cache, which can hold the equivalent of 256 instructions. Subblock disabling can substantially increase the miss rate for benchmarks with higher µop cache miss rates due to false hits. In the data shown, false hits are counted as misses, since the latency penalty is equivalent. This increase in miss rate relative to the fault-free case can degrade performance for applications with no other significant bottlenecks.

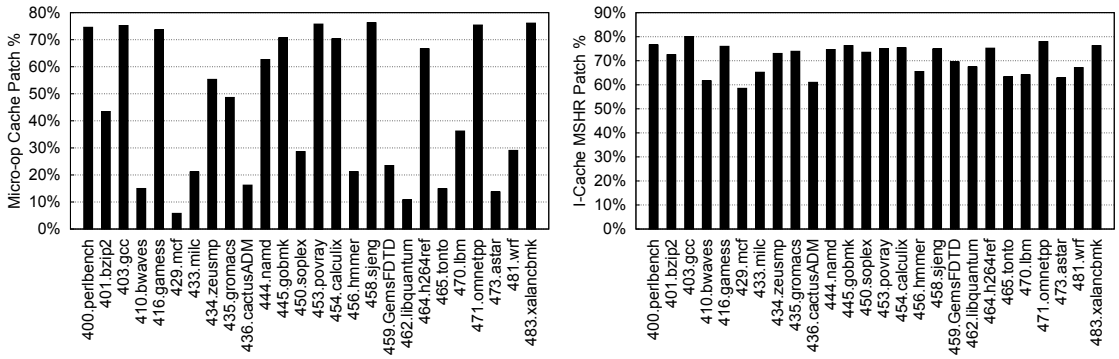


Figure 4.4: Average utilization of instruction cache structures with iPatch.

When iPatch is employed, the miss rate can be reduced, since, some of the time, instruction words "from" disabled i-cache subblocks can supplied from the μop cache or MSHRs. As Figure 4.3 shows, using the μop cache for patching can very significantly reduce the miss rate. This is because the entire μop cache can be repurposed to hold patches as patching becomes a more effective energy-saving mechanism than reducing front end power. Figure 4.4 supports this assertion, showing that, on average, most of μop cache is used for patching in many cases. The applications that do not devote a large portion of the μop cache to patches generally have a high μop cache hit rate. These applications can retrieve most instructions from the μop cache, so L1 false hits are not a concern. As Figure 4.3 shows, MSHR patching also provides a modest reduction in miss rate for some applications, since most of the MSHRs can be used for patching.

Figure 4.5 shows the performance of the subblock-disable scheme and different iPatch configurations in the i-cache. All results are normalized to the performance of the ideal case with no failing cells. As shown, false hits reduce the performance

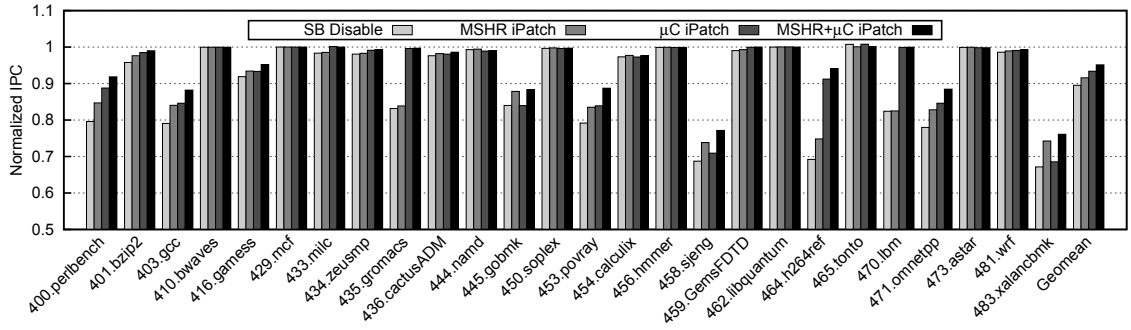


Figure 4.5: Performance benefits of different iPatch techniques with a 0.5% cell failure rate in the L1 instruction cache. IPC is normalized to the fault-free case.

of the subblock-disable scheme by 11% percent on average. This performance degradation is lower than what it would be without the filtering provided by the µop cache, and a number of benchmarks have negligible performance degradation due to their high µop cache hit rates, with performance degradation for the others around 20-30%. Many of these fault-sensitive benchmarks derive significant benefit from micro-op cache (µC) patching. The performance improvements with iPatch correspond to the reductions in miss rates previously shown. On its own, MSHR patching provides less benefit than µC patching, since the i-cache does not have many MSHRs. The combination of approaches reduces the performance degradation to under 5% on average.

4.3.2 Data Cache Patching

An experiment was also performed to evaluate the benefits of iPatch in the data cache. Figure 4.6 shows the d-cache miss rates for subblock-disable and various iPatch configurations. The fault-free d-cache miss rate is also included for com-

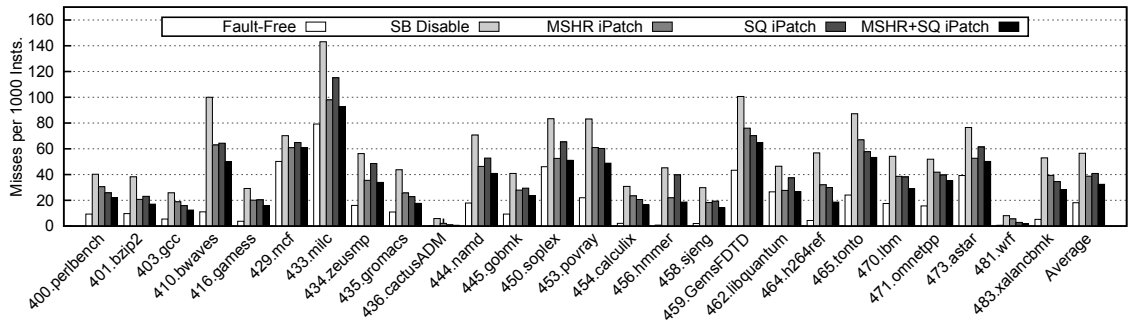


Figure 4.6: Data cache miss rate with different iPatch techniques. False hits are counted as misses.

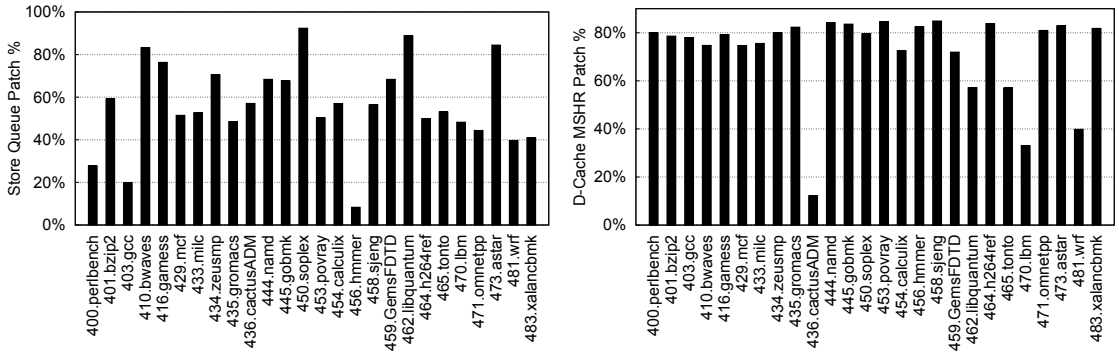


Figure 4.7: Average utilization of data cache structures with iPatch.

parison. False hits are counted as misses, since the latency penalty is equivalent. For the failure rate simulated, subblock-disable can more than double miss rates relative to the fault-free case for most applications. When iPatch is employed, this miss rate can be reduced, since, some of the time, data "from" disabled subblocks can supplied from the store queue or MSHRs. If no patch exists in these structures, however, the missing subblock data must be retrieved from the L2. To minimize such false hits, iPatch's simple patch replacement scheme exploits temporal locality, ensuring that blocks with the highest false hit potential remain patched while evicting less-useful patches if necessary. As the figure shows, both

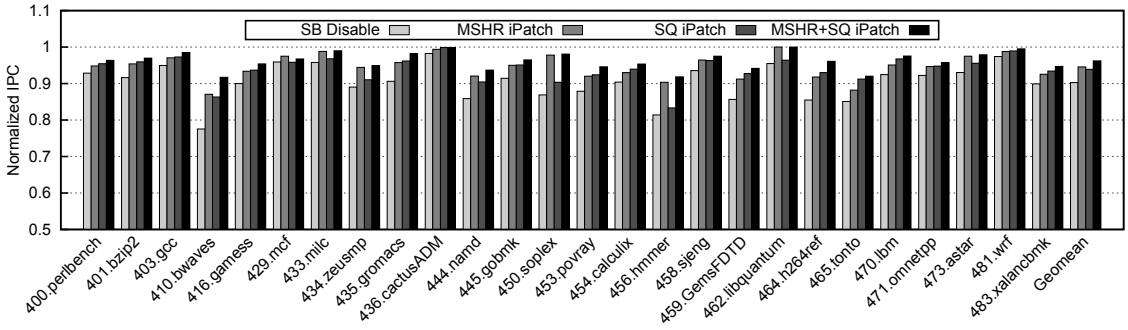


Figure 4.8: Performance benefits of different iPatch techniques with a 0.5% cell failure rate in the L1 data cache. IPC is normalized to the fault-free case.

store queue patching and MSHR patching are able to provide substantial miss rate reductions by avoiding false hits, with the combination providing the most benefit. Figure 4.7 shows the average percent of each buffer that is used to store patches. Due to different resource demands, some applications are not able to maintain many patches, while other utilize a sizable portion of each structure. For instance, if the store queue is fully occupied between stores and patches and a new store must be inserted, a patch can be discarded. This way, patches do not cause the processor to stall due to a full store queue. MSHR patches are managed in a similar manner and also invalidated on writes, so the buffer is not always filled with patches, as shown.

Figure 4.8 shows performance benefits of iPatch over subblock disabling alone in the d-cache. Unlike the μop cache on the instruction side, the store queue provides very little "built-in" patching for the d-cache when simulating the subblock-disable scheme. This is because it is a smaller structure and completed stores are removed by default without iPatch. As in the case of the i-cache, some benchmarks like

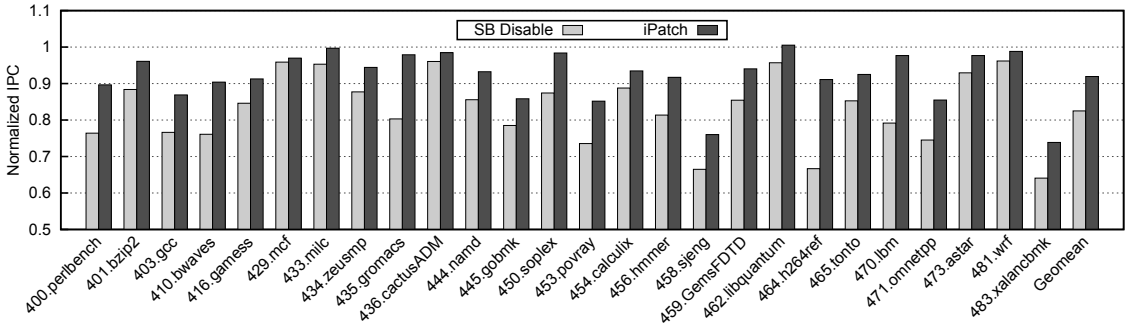


Figure 4.9: Performance comparison of the subblock-disable approach and iPatch when simulating a 0.5% cell failure rate in the i-cache and d-cache.

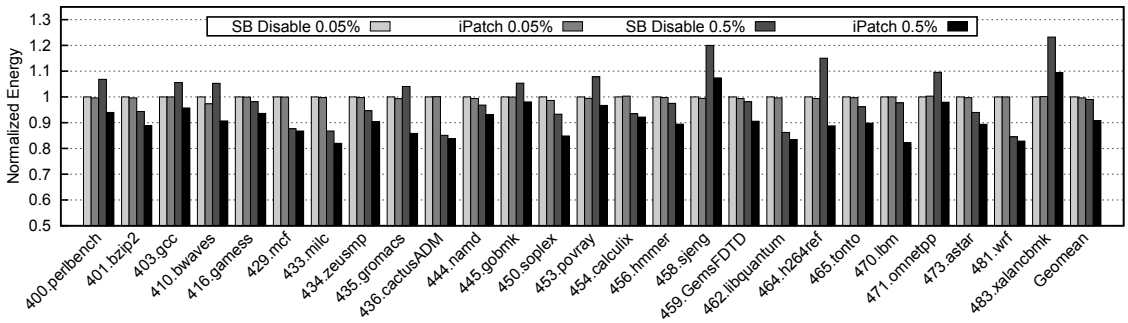


Figure 4.10: Energy consumption with subblock-disable and iPatch at different operating points. Subblock-disable performs well with a 0.05% failure rate, but energy is not saved at the lower-voltage 0.5% point without iPatch.

GemsFDTD and h264ref perform well with SQ patching alone, while others like hmmer and soplex prefer the MSHR approach. In all cases, the combined iPatch approach performs best, though the benefit is not additive.

4.3.3 Overall Results

Figure 4.9 shows the results of simulating failing cells in both the i-cache and d-cache. The performance of the subblock-disable scheme is compared with a configuration that combines all iPatch techniques. As shown, iPatch improves performance over the subblock-disable approach by 11% on average.

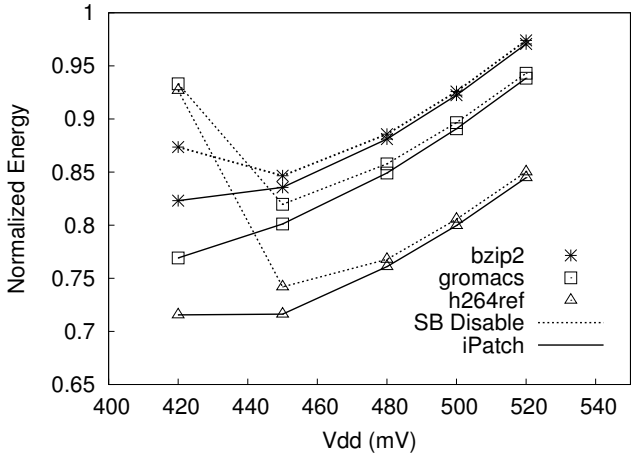


Figure 4.11: Energy scaling curves for three benchmarks comparing subblock-disable and iPatch. iPatch allows further scaling.

Figure 4.10 shows the energy of subblock-disable compared to iPatch (all techniques) when failing cells are simulated in both the i-cache and d-cache. Results for two different operating voltages (and cell failure rates) are shown. At the lower cell failure rate, subblock-disable performs well, and iPatch is not needed, as shown. As the voltage is reduced and the cell failure rate increases, the desired outcome is a reduction in both power and energy. Power is reduced, but as shown, energy consumption actually increases when using subblock-disable due to the performance degradation caused by false hits. When iPatch is enabled, however, this performance degradation is significantly reduced, enabling energy savings despite the higher cell failure rate.

Figure 4.11 details the voltage vs. energy curves for selected benchmarks. The two curves for each application show the energy consumption with subblock-disable and with iPatch. As shown, iPatch extends the energy curve to allow

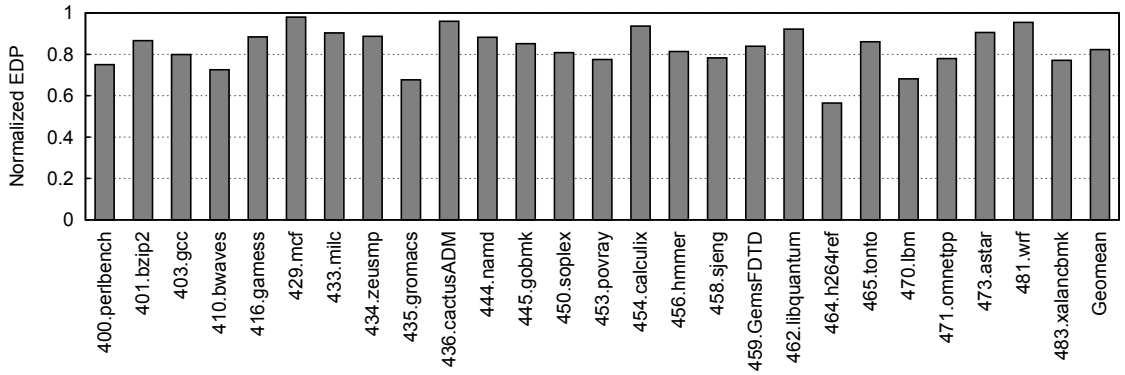


Figure 4.12: Energy-delay product for iPatch normalized to the subblock-disable approach for a 0.5% failure rate. On average, iPatch reduces EDP by 18%.

continued savings at lower voltages. The subblock-disable curve, however, turns upwards due to the extra energy consumption incurred by longer execution times. Finally, Figure 4.12 shows the energy-delay product for iPatch with a 0.5% cell failure rate. Since iPatch provides both lower energy and execution time than subblock-disable, EDP is reduced by 18% on average.

One additional benefit of iPatch is a reduction in performance variation across chips. Table 4.2 shows the performance variation and margin of error across all simulated chips for each configuration at a 0.5% failure rate. To represent the performance variation, the 3σ value is shown as a percentage deviation from the reported mean, where σ is the sample standard deviation. For a normal distribution, this 3σ range comprises 99.7% of all values. The margin of error for the 95% confidence interval is also included. For all configurations and benchmarks, we therefore have 95% confidence that the true population mean is within $\pm 2\%$ of the reported mean. Despite the reduction in performance deviation provided by

Table 4.2: Performance Variation and Margin of Error

	SB-Disable		**iPatch**	
	3σ (%)	**ME (%)**	**3σ (%)**	**ME (%)**
400.perlbench	4.04	0.37	1.28	0.12
401.bzip2	2.98	0.28	0.86	0.08
403.gcc	5.45	0.50	3.12	0.29
410.bwaves	5.97	0.55	1.52	0.14
416.gamess	2.74	0.25	1.84	0.17
429.mcf	0.56	0.05	0.34	0.03
433.milc	1.10	0.10	0.55	0.05
434.zeusmp	1.60	0.15	1.12	0.10
435.gromacs	16.53	1.53	1.61	0.15
436.cactusADM	1.53	0.14	0.34	0.03
444.namd	2.62	0.24	1.20	0.11
445.gobmk	2.04	0.19	1.17	0.11
450.soplex	2.41	0.22	1.12	0.10
453.povray	3.79	0.35	2.01	0.19
454.calculix	2.41	0.22	0.67	0.06
456.hmmer	3.85	0.36	2.77	0.26
458.sjeng	3.83	0.35	2.70	0.25
459.GemsFDTD	8.92	0.82	3.42	0.32
462.libquantum	0.61	0.06	0.57	0.05
464.h264ref	12.51	1.16	3.33	0.31
465.tonto	1.77	0.16	1.77	0.16
470.lbm	11.35	1.05	0.91	0.08
471.omnetpp	5.04	0.47	3.54	0.33
473.astar	0.94	0.09	0.23	0.02
481.wrf	0.62	0.06	0.15	0.01
483.xalancbmk	4.36	0.40	3.40	0.31

the L1 address remapping mechanism, some benchmarks have higher performance variation across chips, with gromacs having the highest at 16.5%. iPatch is able to reduce the performance variation across chips by reducing the number of false hits, since false hits are the cause of the variation. As shown, with iPatch, the variation

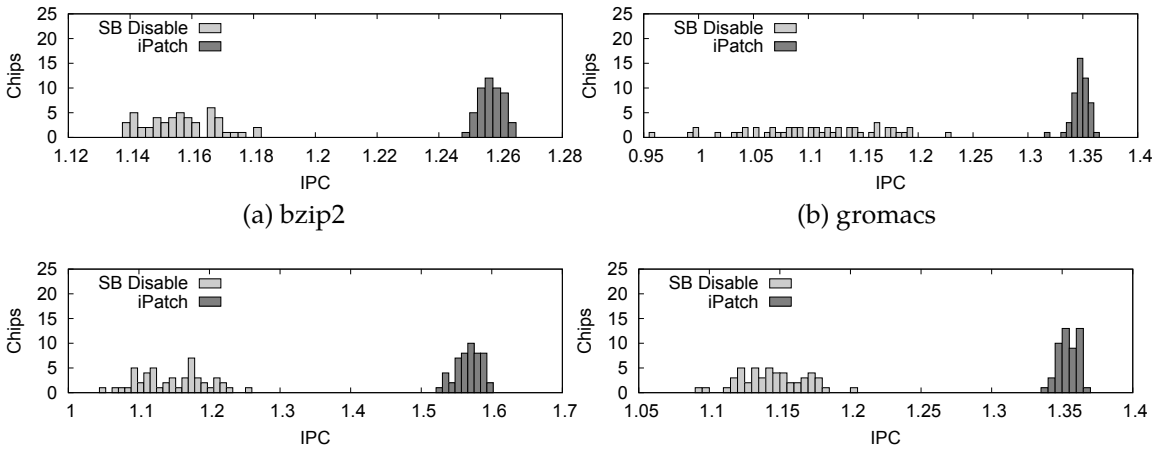


Figure 4.13: Performance distributions for different benchmarks with and without iPatch. As shown, iPatch improves both the mean performance and the performance variability.

range for gromacs drops to 1.6%, which is a 90% improvement.

To help the reader visualize the impact on the performance distribution across chips, Figure 4.13 shows IPC histograms for select benchmarks. In each case, the IPC distribution is shown that corresponds to the data in Table 4.2. As shown, enabling iPatch both shifts and narrows the performance distribution. Both of these changes are desirable from a vendor's perspective. As previously mentioned, gromacs exhibits the most extreme example of this behavior.

4.4 Summary

This chapter presents an evaluation of the iPatch technique. For a particular cell failure rate, results show that iPatch can improve performance by 11% over subblock

disabling. This improvement translates to an 8% energy savings on average, as well as an overall 18% improvement in energy-delay product. Results show that iPatch can extend the benefits of voltage scaling to operating points that would not make sense with subblock disabling alone. A statistical analysis also shows that iPatch significantly improves performance variation as well as the average performance.

5 DRAM PROTECTION USING COP

This chapter introduces COP, a low-cost mechanism "to compress and protect" non-ECC DIMMs from errors due to bit flips, which can result from particle strikes. COP compresses each memory block just enough to insert ECC check bits to protect the block against errors. Because the compressed data and ECC bits are the same size as the original data block, no additional memory accesses are needed and no extra storage space is required to accommodate the ECC. Section 5.1 presents an overview of COP and its basic operation. Next, Section 5.2 describes COP's novel method for tracking compression status. Section 5.3 discusses the pros and cons of using different compression schemes with COP and proposes some simple and effective schemes. Finally, Section 5.4 extends COP to provide protection for incompressible blocks and Section 5.5 summarizes the chapter.

5.1 Overview

The cost of ECC-enabled DRAM DIMMs can be unattractive for commodity systems, yet resilience to bit flips from soft errors is still desirable. The COP approach can significantly enhance the reliability of non-ECC DIMMs with negligible overhead. COP compresses each 64-byte block of memory just enough to insert ECC check bits to protect the block against errors. Because the compressed data and ECC bits are the same size as the original data block, no additional memory ac-

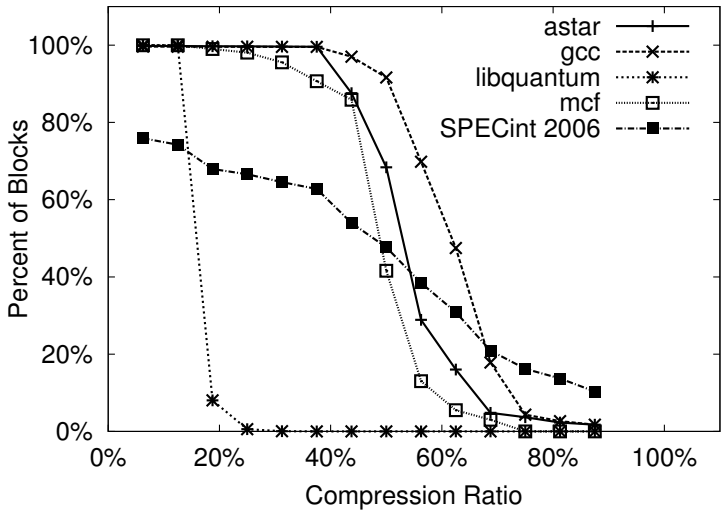


Figure 5.1: Percent of blocks that can be compressed using FPC given a target compression ratio. More blocks can be compressed if less compression is required.

cesses are needed and no extra storage space is required to accommodate the ECC. In addition, maintaining block alignment in memory means that addressing is not affected, which is a potential issue with memory compression.

Traditional intuition regarding cache and memory compression says that a portion of application data is likely incompressible. This is partially because low compression ratios are not considered useful, particularly in context of compressed caches which are often statically segmented. Figure 5.1 shows the compressibility of blocks for selected SPEC benchmarks using the frequent pattern compression (FPC) algorithm proposed by Alameldeen et al. [4]. When only a low compression ratio is required, many more blocks can be considered compressible. As shown, the data for certain applications such as libquantum is not very compressible overall, yet the majority of blocks can be compressed by a small amount (e.g.

10%). COP works well because it only requires a small amount of compression per block, unlike traditional compression applications that seek to provide at least a 2x (50%) compression ratio on average. Furthermore, algorithms such as FPC are engineered to provide high compression ratios overall and are less optimized for lower compression ratios. To address this inefficiency, compression algorithms and optimizations for COP are proposed later in this chapter. These optimized compression approaches are able to compress over 90% of blocks on average, allowing high error coverage.

In COP, the amount of compression achieved for each block determines the number of ECC check bits that can be inserted, and therefore which codes can be used. Although it is theoretically possible to use stronger codes for more compressible data blocks, for simplicity, COP targets the same compression ratio for each block. For COP, compression to free either 8 bytes or 4 bytes per block was considered. When only 4 bytes must be freed, COP is able to compress most blocks, constituting a compression ratio of 6.25%. For this reason, the remainder of this discussion will describe the 4-byte case, although the mechanism can be applied for other configurations. In the preferred scenario, each compressible 64-byte block in memory is compressed to 60 bytes of data, making room to add 4 bytes of ECC check bits. Instead of using a single code to protect the block, COP divides each block into four (128,120) SECDED codes. Each code requires one byte of check bits to protect 15 bytes of data (a total of 4 bytes of parity for the whole line). This

design benefits COP in two primary ways. First, it requires only a relatively simple SECDED code, since the (128,120) code is the full version of the commonly-used (72,64) truncated code. Second, this approach allows COP to detect whether or not a given block is compressed (and protected) or was stored unprotected because it could not be compressed. This latter mechanism relies on the μarch patching technique that is the theme of this dissertation.

5.2 Tracking Compression for "Free"

Although COP requires only a small amount of compression for each block (e.g. 6.25%), some blocks may not be compressible at all. The default COP approach cannot protect these blocks, since parity bits cannot be included. One option in this case is to simply leave incompressible blocks unprotected, since COP can (ideally) protect the vast majority of other blocks. This approach still achieves a high level of error protection while incurring minimal performance and storage costs.

COP can seamlessly integrate uncompressed blocks alongside compressed blocks in memory, provided that it can tell the difference. Distinguishing between the two is critical and is a key contribution of COP. Without using extra space in DRAM, however, it is not possible to store metadata for all blocks to track which are compressed and which are not. For instance, to track the compression status of each block, a bitmap could be stored in memory with a bit for every 64-byte block. For an 8GB memory, however, this approach would require a 16MB bitmap,

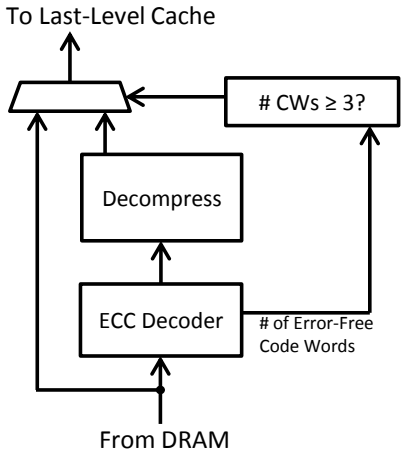


Figure 5.2: ECC decoder/decompression logic. The ECC decoder counts the number of valid error-free code words seen. If enough code words are seen, the data (minus ECC check bits) is decompressed and sent to the LLC. If not enough code words are seen, the data is passed unmodified to the processor.

which is comparable in size to a large L3 cache. Another alternative is to compress each compressible block enough to add both the parity bits and a special sequence of bits (a magic word) to indicate that the block is compressed. In addition to requiring more compression, this approach also introduces the (relatively high) possibility that an uncompressed block will be mistaken as compressed if its data happens to contain the magic word.

Instead of checking for a magic word to signify compression, COP simply checks each memory block for valid ECC code words. In this context, a valid code word is defined as a 128-bit word that produces a zero syndrome when passed through a particular (128,120) SECDED decoder. In traditional applications, a zero syndrome indicates a lack of errors. As shown in Figure 5.2, if fewer than 3 blocks contain valid code words, the block is treated as unprotected/uncompressed data

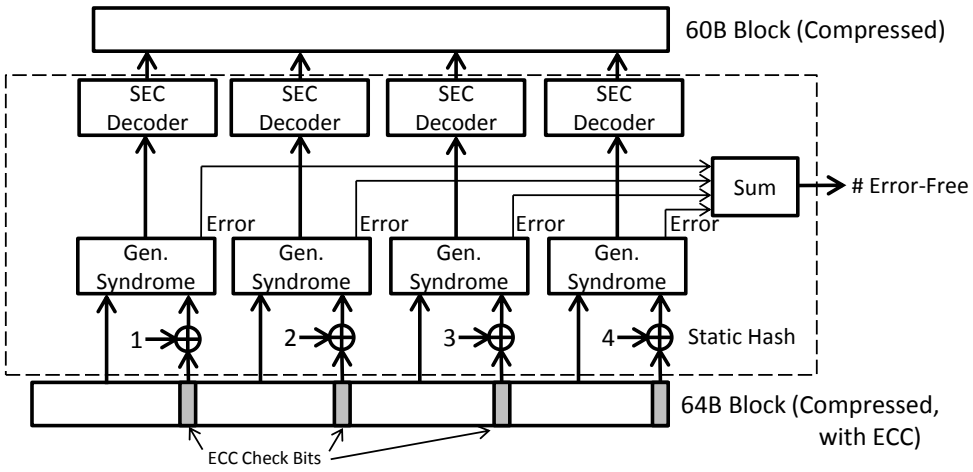


Figure 5.3: ECC decoder in detail. A counter totals the number of zero syndromes. A different static hash is applied to each code word before decoding. (The same hash is applied on writes.)

and passed to the last-level cache and processor unmodified. Unlike the previously-discussed approach with the special bit sequence, the chances of an uncompressed data block containing multiple valid code words are extremely low, and in the very rare cases when this occurs, COP *guarantees* correctness by leveraging μarch patching.

Figure 5.3 shows the decoder logic in greater detail. After reading a block from DRAM, COP treats it as if it contains four (128,120) code words and passes all of them through the ECC check logic. If the block was compressed/protected before being written to memory and no error has occurred, the ECC logic will detect 4 code words with zero errors. If the block was not protected, the entire 64 bytes contains uncompressed application data. In this case, it is unlikely that the ECC decoder will detect a single valid code word and highly unlikely that the

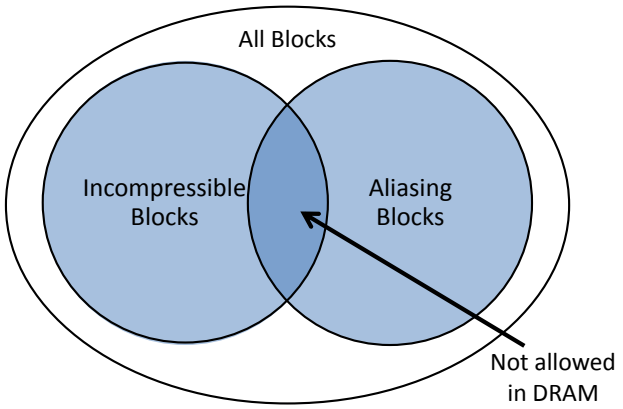


Figure 5.4: Illustration of blocks allowed in DRAM with COP (not to scale). If a block's data contains 3 or more code words, it is considered an alias. For the majority of blocks that are compressible, aliasing is not a concern. Incompressible aliases cannot be written to memory because the decoder will misinterpret them.

block happens to contain multiple valid code words. In this implementation, if the memory controller detects 3 or more valid code words in a block, it will treat the block as compressed data. If a soft error has occurred in a protected block, 3 of the 4 code words will remain valid, and the ECC can be used to correct the invalid code word before the block is decompressed.

Although it is very unlikely, it is possible that a block of application data could happen to contain 3 or more code words, which would confuse the decoder. These blocks with 3 or more valid code words will be referred to as *aliases* because they would appear to the decoder to be compressed even if they are not. For functional correctness, COP must guarantee that it is impossible for an uncompressed block to be erroneously treated as a protected/compressed block because it is an alias. This can be accomplished very simply using μarch patching. COP does not write aliasing blocks to DRAM and instead keeps all such blocks in the last-level cache

so that they will patch the stale data in DRAM.

Figure 5.4 illustrates the possible types of blocks in COP. Most blocks will be compressible and will be stored to DRAM in compressed format along with ECC parity bits. As shown, a subset of blocks are incompressible, and a subset of blocks can also be considered aliases. Compressible blocks that are aliases in their uncompressed form are not a concern because they will be compressed in DRAM. As shown, however, the extremely rare blocks that are both incompressible and aliases are not allowed in DRAM and must reside in the LLC to ensure that the decoder works correctly. Note that COP need not keep incompressible blocks containing only 2 valid code words in the LLC, even though a soft error could theoretically create a 3rd valid code word. Although this scenario would cause the block to be misinterpreted as compressed, the error is guaranteed to result in data corruption in any case since the block was unprotected.

To give the reader an idea of the probability of a block being an alias, consider the (128,120) code mentioned before. Because there are 120 data bits, this code allows for 2^{120} valid code words, while there are 2^{128} possible values that can be represented by the 128 bits when including the parity bits. Given a random 128-bit value, there is then a 0.39% chance that it will be a valid code word. Given a 512-bit block containing 4 random 128-bit values, there is a 0.00002% chance of the block containing 3 or more valid code words. Since the majority of blocks are compressible and COP must only retain incompressible aliases in the LLC, very

few blocks, if any, will fall into this category. To remember that a block in the LLC is an incompressible alias, COP requires an extra metadata bit for each LLC block. Upon a writeback to memory, the compressor/ECC encoder logic checks for incompressible aliases and rejects writebacks of these blocks, requiring them to be kept the LLC with the "alias" bit set. Alternatively, a check could be added to writebacks to the LLC to proactively set this bit. When a replacement is needed, the replacement decision is modified to avoid evicting any incompressible aliases. Upon a writeback, the data may have changed or become compressible, so the alias bit is cleared.

Although the previous example discusses the chances of aliasing for completely random data, application data is not usually random. For some applications, blocks may even contain the same word repeated multiple times. In this case, if the repeated data happens to be a valid code word, the block will contain multiple valid code words, significantly changing the odds discussed above. To avoid this scenario, COP introduces a static hash that is XORed into each compressed block when it is written by the encoder and before it is processed by the decoder. By using a different hash for each 128-bit segment as shown in Figure 5.3, repeated values will not skew the odds.

A diagram of the ECC encoder/compressor is not included because it is simply the opposite of the decoder/decompressor shown, with the stages in reverse order. In the encoder, the compression logic first compresses the 64-byte block down to 60

bytes. The SECDED logic then computes 4 bytes of parity bits for the compressed data, after which the static hash is applied. The compressed/protected block can be written to DRAM at this point. If the compressor cannot compress the block, the 64 bytes of data are written to DRAM as-is, and no hashing is applied.

The observant reader will have noticed that although 4 SECDED codes are used when protecting a compressed block, the decoder implementation described limits COP to correcting only one bit error per block or detecting a double error within one 128-bit code word. If two errors occur and corrupt different SECDED code words, there will be only two valid code words remaining, resulting in data corruption when the decoder erroneously passes the compressed block to the processor without decompressing it. To extend correction to this scenario, the code word threshold could be reduced from 3 to 2, although the number of aliases would increase by orders of magnitude. In a different COP implementation using 8 bytes of ECC metadata per block, COP could divide each compressed block into 8 (64,56) SECDED words, allowing the decoder to still require a high valid code word count (e.g. 5 out of 8) but enabling single-bit correction in multiple code words. As previously discussed, however, requiring more compression reduces the overall number of blocks that can be compressed/protected.

It is possible, though extremely unlikely, for too many incompressible aliases to map to the same set in a set-associative LLC, causing an overflow condition for that set. There are various approaches to handling this corner case. One solution

is to enable compression on a per-page basis. This approach would require an extra bit per page table entry, which would need to be passed along with each request to the LLC. When encountering the overflow condition, compression could be disabled for the affected page(s). A simplified version of this approach could disable compression for all of memory, falling back on a technique like Virtualized ECC [79]. Another solution is to spill the addresses (and/or data) of the incompressible aliases to a small region of DRAM (likely less than a page in size). Spilling of the set could then be accomplished by adding an overflow flag bit per LLC set and reserving one of the tag fields to use as a pointer to the overflow blocks. Any misses to an overflowed set must then follow this pointer to search the overflowed blocks for a hit before performing conventional miss handling. The overflow blocks can be arranged as a linked list, allowing an arbitrary number of collisions to be handled, albeit with additional latency. Since this overflow scenario is exceedingly rare, this slow mechanism is only needed for correctness and does not create a performance bottleneck. These approaches can also be applied in case the LLC needs to be power gated or flushed for any other reason.

5.3 Compression Schemes

Because COP does not require or benefit from high compression ratios, it can use simpler and less aggressive compression algorithms. A key insight is that many compression approaches that are capable of providing high compression ratios are

ineffective at compressing blocks with more limited compressibility. This can occur when the overhead of the compression metadata dominates the space reduction achieved. For instance, frequent pattern compression (FPC) requires a 3-bit prefix per 32-bit word, incurring a cost of 48 bits of metadata per block [4]. To free 4 bytes (32 bits), FPC must recoup the cost of this metadata and extract a total of 80 bits of redundancy from the block. By using a compression algorithm requiring less metadata, blocks with limited compressibility can be protected.

The remainder of this section proposes compression schemes and optimizations for use with COP. An overview of compression schemes can be found in Section 2.3.2. For each scheme used with COP, the target compression amount is increased by two bits (to free 34 bits overall) to allow COP to combine compression schemes for maximum benefit. In the combined approach, COP uses two bits of every compressed block to indicate which compression scheme was used.

5.3.1 MSB Compression

This approach is inspired by the (more-complex) base delta immediate (BDI) algorithm [58]. Standard base delta immediate (BDI) can provide a compression ratio much higher than what is required by COP. An implementation of BDI for COP could be much less aggressive and target a wider dynamic range of values. A wider dynamic range would normally be less beneficial, since this increases the size of the deltas that must be stored. Because COP does not need high compression,

this opens the door for significant simplifications.

When applying BDI compression, the most significant bits of the base value typically remain unchanged after the deltas are added during decompression. For instance, if a delta is one byte while the base is four bytes, a long carry chain is required to change the MSBs of the result when the delta is added to the base. A less complex compression scheme could therefore simply check for matching MSBs across each value in the block. COP's MSB compression compares the 5 MSBs of each of the 8 8-byte words in a block. If these bits match, 7 of the redundant 5-bit sequences can be removed. This removal frees 35 bits, making room for 32 bits of ECC and 2 bits to indicate the compression scheme. To free more than 4 bytes per data block, more bits can be included in the comparison. The MSBs of 8-byte-aligned values are used to allow blocks of both 64-bit and 32-bit values to be compressed. If the block contains 32-bit values, half of them will be omitted from the comparison. This simplification of the compression algorithm reduces the logic required since only a comparison is needed, unlike BDI, which relies on addition and can support multiple delta sizes.

Traditional BDI compression is also not very effective for blocks containing floating point values, since the significands of floating point values are left-normalized. The bit comparison used by MSB compression, however, overlaps the floating-point exponents, allowing it to compress floating point values with similar exponents. The most significant bit in standard floating point representation is the sign bit,

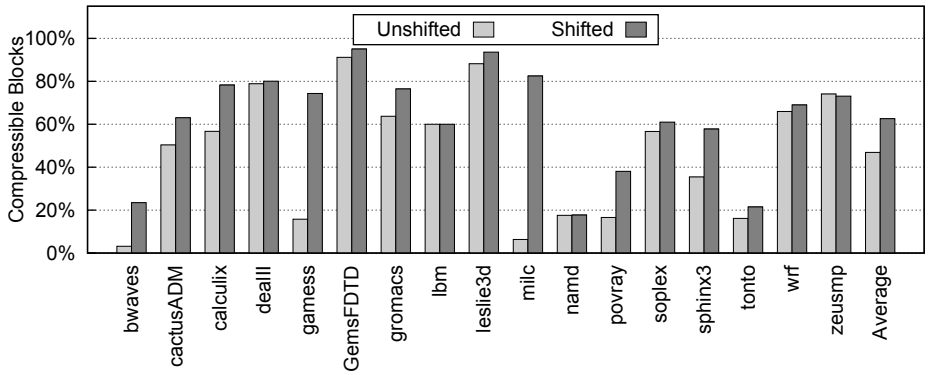


Figure 5.5: Compressibility improvement using MSB compression when the comparison is shifted by 1 bit. To be considered compressible, 4 bytes must be freed in a block.

however. If we include the sign bit in the MSB comparison, blocks containing values with different signs cannot be compressed. To optimize overall compressibility by including this case, COP slightly modifies the previously-discussed approach by shifting the 5-bit comparison over by one bit, such that it ignores the most significant bit. This optimization can significantly benefit the compressibility of floating point applications. To demonstrate this benefit, Figure 5.5 shows the compressibility of SPECFP 2006 benchmarks using MSB compression with both unshifted and shifted comparisons. As shown, by shifting the MSB comparison by 1 bit, compressibility improves by 15% for these applications.

5.3.2 Frequent Pattern Compression

Because FPC is a well-known algorithm in the realm of cache compression, its effectiveness was evaluated in conjunction with COP [4]. Results indicate that while FPC is best-suited for achieving high compression ratios it is less effective

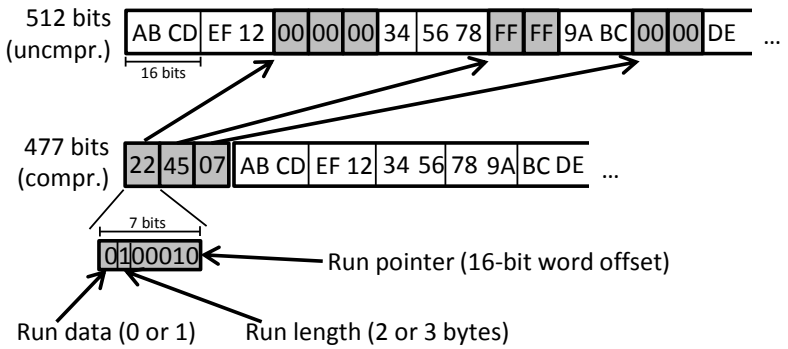


Figure 5.6: Example of compression using run length encoding. Each run is encoded using 7 bits of metadata.

for less-compressible blocks due to the fixed metadata overhead. In the context of COP, a simplified run length encoding can extract some of the same redundancy as FPC with less metadata overhead, allowing more blocks to be compressed.

5.3.3 Run Length Encoding

Run length encoding extracts redundancy in the form of runs of binary 1s or 0s [33]. The benefit of this approach over a compression algorithm such as FPC is that only a small number of runs must be extracted, requiring less metadata. COP's implementation requires 7 bits of metadata for each run, where runs can be either 2 or 3 bytes long. Therefore, accounting for the metadata, COP needs only 2 3-byte runs, 4 2-byte runs, or a combination or 2-byte and 3-byte runs in order to free 4 bytes for ECC. For instance, in the case with 2 3-byte runs, 6 bytes of redundancy are removed while adding two 7-bit metadata sequences, freeing the required 34 bits.

The 7-bit metadata to encode each run is structured as shown in Figure 5.6. The first bit indicates if it is a run of 0s or a run of 1s. The next bit indicates whether the run is 2 bytes long or 3 bytes long. Finally, the next 5 bits point to the 16-bit word offset in the block where the run begins. To compress a block, metadata for each run is placed at the start of the block, and the runs described by the metadata are omitted from the rest of the block.

Only the minimum number of runs must be encoded to create space for ECC. Thus, the number of runs encoded per block can vary depending on the length and number or runs present. When decompressing a block, the number of runs (and metadata chunks) can be determined by examining the metadata chunks and counting the number of bytes freed by each one. The decompression logic looks at each metadata chunk in sequence, computing the number of run bytes encoded. Once enough bytes have been freed to store the ECC (e.g. 4 bytes), this signals that the metadata stops and the actual data begins, since no more runs would be needed.

5.3.4 Text Compression

Many widely used applications are responsible for processing large amounts of text. Latin characters that fall within the range of ASCII encoding are very common. The ASCII standard is a 7-bit encoding that defines 128 possible characters [6]. Historically, one byte was transmitted per character with the 8th bit being used

for parity. In more modern storage of ASCII characters, each character is stored in a byte with an extra zero as the most significant bit. If an entire memory block contains ASCII characters, the MSB of all bytes will be zero, and can be omitted to compress the block. This approach works well in the context of COP, since it is capable of providing only low compression ratios. For instance, COP could theoretically free 62 bits in a 64-byte line (a compression ratio of 12%), assuming 2 bits to indicate the compression type.

The more modern Unicode formats enable additional characters, but maintain backwards compatibility with ASCII. For instance, UTF-8 is a variable length encoding that mixes ASCII symbols with longer symbols. UTF-16, on the other hand, uses a fixed 2-byte representation per character. To convert ASCII characters to UTF-16, one byte of zero padding is simply added. Therefore, even when considering Unicode text, if a data block contains only ASCII characters (which is even more likely for UTF-16, as a block contains half the number of characters), it can be compressed using this approach. Even for languages with non-latin characters, characters that fall within the ASCII range are still commonly used, as is the case for HTML.

5.4 Protecting Incompressible Blocks

As described thus far, COP is able to protect compressible blocks, which comprise a high percentage of application data. For certain applications or hardware

implementations, it may be desirable to protect all of memory against bit flips, including incompressible data. Prior proposals (e.g. Virtualized ECC) can protect non-ECC DIMMs by allocating space for ECC check bits in memory [79]. As previously discussed, there are two downsides to this approach. First, extra memory accesses are required to retrieve the check bits associated with the data, which can degrade performance. Second, making room for ECC check bits in main memory substantially reduces the usable memory space.

In cases where it is imperative to protect all data from errors, a hybrid version of COP can allocate a small ECC region in memory. This version of COP will be referred to as COP-ER. In a naïve implementation, the same storage overhead as Virtualized ECC is required, since incompressible blocks are not always adjacent, so ECC space could be reserved for all blocks to facilitate addressing of the ECC region. In this manifestation, the benefit of the combined approach is in performance, since most of the time the check bits can be retrieved with the compressed data, and the ECC region need not be consulted.

Since the ECC region is only used to store check bits for incompressible blocks, an optimized implementation can significantly reduce its storage overhead such that the available memory space is minimally impacted. Through appropriate engineering of the ECC region, COP-ER can use the space as efficiently as possible, tightly packing ECC data. When ECC entries are not present for all blocks, a simple offset computation is no longer effective for finding a block's parity bits in the ECC

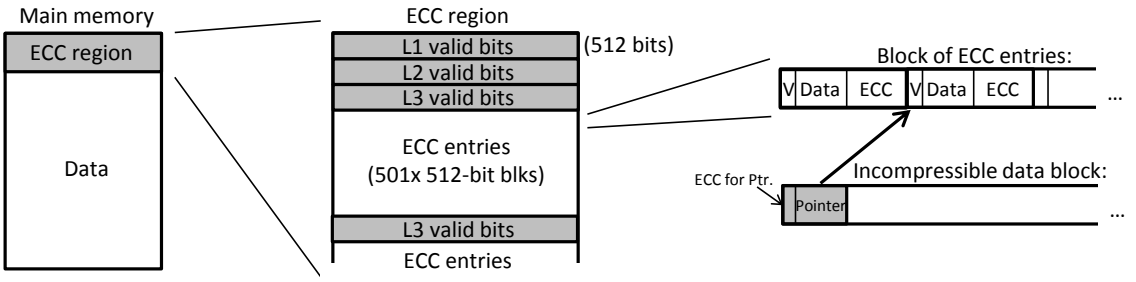


Figure 5.7: ECC region organization for COP. A small portion of main memory is allocated for ECC. Each uncompressed data block is truncated to include a pointer to an ECC entry containing the displaced data and parity bits for the block.

region. There are a number of possible solutions to architecting the ECC region to solve this problem. For instance, it could be structured like a set-associative cache using a hash on a block's address to determine which set its parity bits are store in. This approach, however, creates potential for set overflows and does not efficiently use the allocated space. In the solution proposed in this work, COP-ER displaces a portion of each incompressible block's data and inserts a pointer to point directly to an entry in the ECC region. Each entry in the ECC region is then comprised of a valid bit, the displaced data, and the ECC check bits needed to protect the whole block.

Figure 5.7 shows the structure of COP-ER's optimized ECC region. As shown, the ECC region occupies a portion of the memory space and can grow dynamically as needed. To allow the region to be resized, the operating system can avoid allocating the nearby pages until memory is near capacity. As shown, the bulk of the ECC region is comprised of blocks containing ECC entries. If 28-bit pointers are used in incompressible blocks to point to an ECC region block/entry offset,

an additional 6 parity bits are required to correct any bit errors in the pointer, displacing a total of 34 bits from each incompressible block. Each ECC entry, therefore, must contain 34 bits of data plus 11 bits of parity for the incompressible block plus a valid bit for a total of 46 bits per entry, allowing 11 entries per 64-byte ECC region block.

On a read to an incompressible block, the decoder first detects that the block is not in compressed format (see Figure 5.2). Next, a read is performed to access the ECC region block containing the entry indicated by the embedded pointer. Note that ECC region blocks can be cached to improve performance. The missing data and parity bits are retrieved, and the block is checked for errors. COP-ER adds an additional bit to be stored with each L3 cache block to indicate if the block was uncompressed when originally read from DRAM.

On an LLC replacement, if the victim line is clean, it can be silently invalidated and overwritten with the new block. If the victim is dirty and the "was uncompressed" bit is set, COP-ER knows that an ECC entry already exists for the block. In this case, the pointer to the ECC entry is read from memory. If the block is now compressible, the original ECC entry is invalidated and the block is written in compressed format. If the block is still incompressible, the existing ECC entry can be reused and updated with the new data/parity bits. If the "was uncompressed" bit is *not* set and the dirty data being written back is incompressible, a new ECC entry must be allocated.

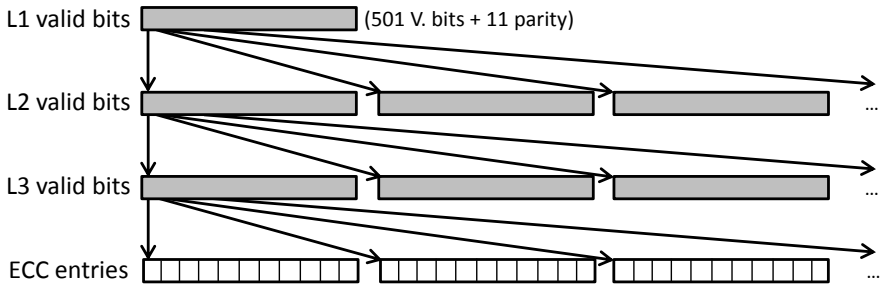


Figure 5.8: Valid bits form a high-radix tree to speed the location of free entries.

To find a free spot for a new ECC entry, the valid bit hierarchy shown in Figure 5.8 is used. In a worst-case scenario, a large main memory storing a lot of incompressible data could require millions of ECC entries, so COP-ER needs an efficient way of maintaining and searching the free list. Each L3 valid bit block contains 501 valid bits in addition to 11 bits of parity to protect the valid bits. Each valid bit corresponds to one block of ECC entries, as shown. When all 11 ECC entries in an ECC region block are valid, its L3 valid bit is set. The memory controller stores a pointer to the most recently used block of L3 valid bits. To allocate a new ECC entry, it looks for a 0 valid bit, indicating a block of ECC entries with a free entry. If all 501 valid bits are set, the 3-level valid bit hierarchy is walked. Each block of L2 valid bits corresponds to a block of L3 valid bits, and a similar arrangement applies for the L1 bits.

This valid bit hierarchy allows free ECC entries to be efficiently found and filled without a (time-consuming) exhaustive search. When the last ECC entry in a block is allocated or an entry is freed in a full block, the tree structure is updated

appropriately. By filling free entries this manner, COP-ER limits the size of the ECC region in case the data compressibility changes or memory for an application is deallocated.

Another benefit of using COP-ER over COP is that COP-ER can virtually eliminate the incompressible alias problem. Recall that COP-ER displaces some data in incompressible blocks to store an ECC region pointer. This provides some control over the bits that get stored in DRAM. If the bits used to form the pointer are selected such that they overlap with all four code words as seen by the decoder, the ECC entry allocated can be adjusted so that the block is no longer an alias when incorporating the pointer.

5.5 Summary

This chapter proposes COP, a technique to protect non-ECC DRAM DIMMs from bit flips. COP leverages simple compression techniques to compress blocks of data, making room for ECC parity bits. Various simple compression techniques to be used with COP are described. Because the compressed data and ECC take up the same amount of space as the uncompressed data, COP imposes minimal storage and performance overheads, if any. To extend protection to incompressible data, an approach called COP-ER is also proposed. COP-ER uses a dynamically-resizable storage region to store parity only for blocks that cannot be compressed.

6 EVALUATION OF COP

This chapter describes the methodology for evaluating COP and discusses the simulation results. Section 6.1 describes the simulation infrastructure and approach. Section 6.2 evaluates the simple compression schemes proposed for COP. Section 6.3 models COP's reliability with a simple fault model. Section 6.4 discusses COP's performance compared to prior work, evaluates the potential for aliasing, and compares COP's storage requirements with other approaches. Finally, Section 6.5 summarizes the chapter.

6.1 Simulation Infrastructure

To evaluate COP, the interval simulation methodology was used to model performance [26]. This approach divides execution into intervals between long-latency miss events, which will have largest performance impact and overshadow lower-latency accesses. This methodology allowed the efficient simulation of large instruction regions using a custom simulator built on top of DRAMSim2 [63]. The SPEC2006 benchmark suite was used with reference inputs as well as the PARSEC suite with native inputs. A section of 1 billion instructions was run from each SPEC benchmark and was chosen using the SimPoint tool [60]. The PARSEC workloads were run in 4-threaded mode for 4 billion instructions in the parallel region of interest.

Table 6.1: Simulator configuration

Category	Configuration
OoO Core	3.2 GHz Issue: 4-wide Window size: 128
Caches	L1 Instr: 32 KB/4-way, 4 cycles L1 Data: 32 KB/8-way, 4 cycles L2: 256 KB/8-way, 9 cycles L3: 4 MB/16-way, 34 cycles
Memory	Bus speed: 1600MHz Bus width: 64 Total capacity: 8GB Channels: 2 DIMMs per channel: 1 Ranks per DIMM: 2 Chips per rank: 8

Sniper, a Pin-based simulator, was used to capture a trace of references to the L3 cache along with the data contents of each referenced cache block for compressibility analysis [14]. These references were divided into epochs, each containing independent (overlappable) requests. The perfect-L3 IPC was also recorded, allowing the simulator to compute the performance impact of L3 cache misses. A 4MB L3 cache was modeled as well as a dual-channel 8GB main memory. Table 6.1 shows the details of the simulated system. The results shown in this chapter highlight the memory-intensive benchmarks shown in Table 6.2. For benchmarks with multiple input sets, the input set with the largest footprint is used. The highlighted subset includes 20 benchmarks from the SPEC and PARSEC suites. Averages for each suite are also shown in the result figures.

Table 6.2: Memory-intensive benchmarks

SPECint 2006	SPECpf 2006	PARSEC
astar	bwaves	canneal
bzip2	cactusADM	fluidanimate
gcc	GemsFDTD	streamcluster
mcf	lbm	x264
omnetpp	milc	
perlbench	soplex	
sjeng	wrf	
xalancbmk	zeusmp	

6.2 Compression Scheme Evaluation

First, the effectiveness of the different compression schemes was evaluated. Each benchmark was simulated while noting the compressibility of each DRAM block accessed. The total number of DRAM accesses was counted in addition to the number of accesses to compressible blocks. Two compression ratios were also evaluated for this experiment to free either 8 bytes or 4 bytes per line to accommodate ECC.

Figures 6.1 and 6.2 show the results for freeing 8 or 4 bytes, respectively. As shown, COP can compress significantly more blocks if only 4 bytes must be freed. As shown in Figure 6.2, text compression (TXT) is particularly effective for certain benchmarks such as perlbench. MSB compression, on the other hand, is very effective overall and able to compress approximately 70% of blocks on average. Run length encoding (RLE) is similarly effective overall, though some benchmarks favor one of MSB or RLE. An evaluation of frequent pattern compression (FPC)

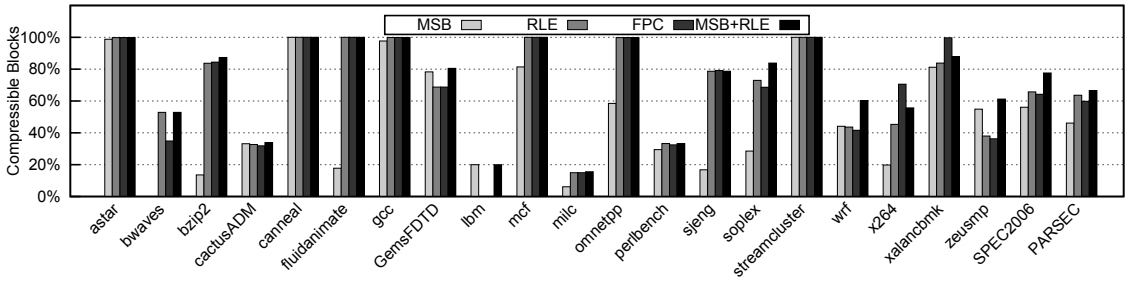


Figure 6.1: Compressibility when freeing 8 bytes per 64-byte block.

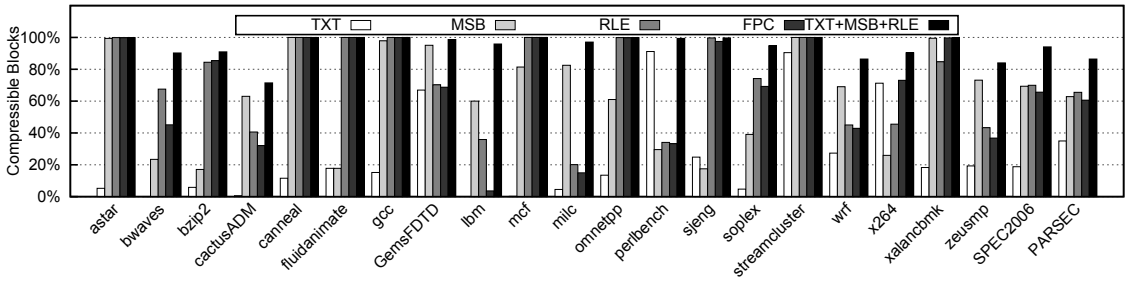


Figure 6.2: Compressibility when freeing 4 bytes per 64-byte block.

is also included for comparison. Because RLE generally outperforms FPC and has a simpler hardware implementation, FPC is not included in the combined compression algorithm. The combined algorithm includes the best of all of the schemes, incorporating TXT, MSB, and RLE using 2 bits of the compressed block to select one of 3, as previously discussed. As shown, the combined approach is highly effective and able to compress 94% of blocks on average.

6.3 Reliability Analysis

To evaluate the reliability benefits of COP, a methodology inspired by the PARMA reliability model was used to compute the soft error rate (SER) [72]. To compute

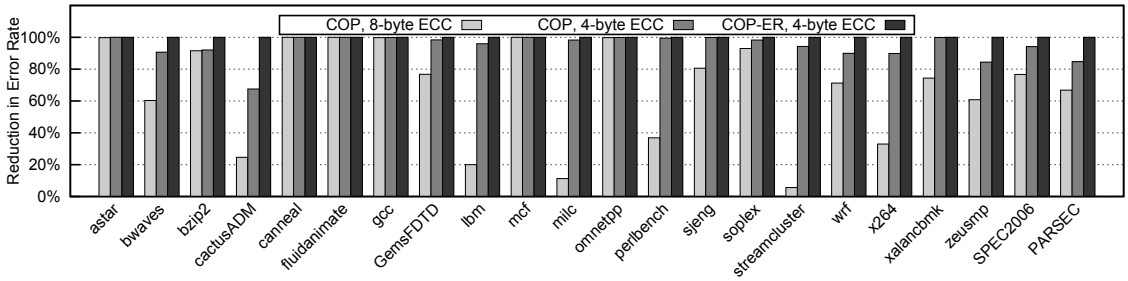


Figure 6.3: Soft error rate reduction with COP. The 4-byte ECC requires less compression, allowing protection of more blocks.

the SER for L2 caches, PARMA introduces the idea of a "vulnerability clock" that counts the number of cycles that each data block is vulnerable in the cache before it is read. Using a raw SER (per bit) the analytical model can compute the SER due to a block being vulnerable for a given amount of time. This approach was adapted for use in DRAM, tracking the amount of time that each block was vulnerable in DRAM before being read into the L3. The SER reported is due to DRAM soft errors, and reflects the rate of erroneous data from DRAM making it into the L3. Because applications have different sized footprints and L3 miss rates, a unique SER was computed for each benchmark. The reliability evaluation of COP was based on a raw soft error rate of 5000 FIT/Mbit, as in [40].

Figure 6.3 shows the computed SER reduction for each benchmark using COP. Results are shown for either 8 bytes or 4 bytes of ECC per compressed block. In all cases, one byte of parity bits is used per code word, with the 8-byte version incorporating 8 (64,56) SECDED codes per block, and the 4-byte version using 4 (128,120) code words. As shown, the 4-byte version provides better reliability, with

a 93% reduction in SER on average. This result is interesting, because for protected blocks, a block with 8 code words will be more error-tolerant than a block with 4 code words. Because the 4 code word case protects more blocks overall, however, it is superior. The error rate reduction provided by COP-ER is also shown, and is nearly 100% in all cases, since COP-ER can correct all single-bit errors.

6.4 Performance and Storage Overheads

In order for COP to perform well and function correctly, the number of aliasing blocks kept in the LLC to patch DRAM must be minimal. To evaluate the potential for aliasing in the benchmarks evaluated, a (128,120) SECDED code was generated and tested. The H-matrix for this code is shown in Figure 6.4. The ones in each row of the matrix indicate the bits to be XORed together to generate each syndrome bit. To check a block of data for valid code words, it is divided into four 128-bit parts. A different hash is applied to each part by inverting certain bits, as previously shown in Figure 5.3. The 8-bit syndrome is then calculated for each part using the H-matrix. A syndrome of all zeros indicates that the 128-bit input is a valid code word.

Table 6.3 shows the theoretical percent of blocks that are both incompressible and contain valid code words, as well the percent measured using the SPEC and PARSEC benchmarks. One column of results from the analytical evaluation assumes that all blocks are incompressible, and the other assumes that 80% are

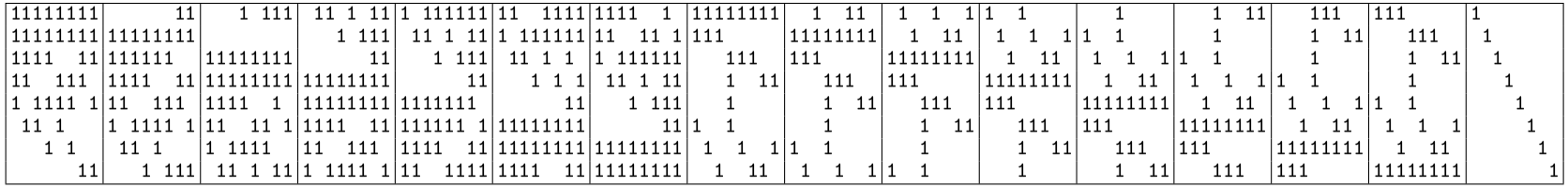


Figure 6.4: H-matrix for the (128,120) SECDED code used for COP. Each row of ones indicates the bits that are XORed together for each bit of the syndrome. The identity matrix to the right corresponds to the 8 parity bits.

Table 6.3: Code words in incompressible data blocks

	Analytical		Measured	
Num. code words	**0% Compr.**	**80% Compr.**	**SPEC+PARSEC**	**Equiv. 8GB mem. blocks**
1	1.5%	0.3%	1.4%	1879048
2	0.009%	0.002%	0.005%	6710
3	0.00002%	0.000005%	0.000002%	3
4	0.00000002%	0.000000005%	0.0%	0

compressible. In both cases, incompressible blocks are assumed to contain completely random data. Results are also shown for the benchmarks evaluated by applying the (128,120) SECDED code to the data. To help the reader interpret the measured result, the equivalent number of blocks in a fully-used 8GB main memory is also shown. Only blocks with 3 and 4 code words are considered aliases, although the table shows other cases for completeness. Out of the applications studied, only a single incompressible data block containing 3 code words was observed, and no blocks containing 4 code words were observed.

The performance of COP was modeled and compared to other approaches that allocate ECC space in memory, similar to Virtualized ECC [79]. The baseline implementation in this work (ECC Reg.) differs from the Virtualized ECC proposal in two ways. First, because Virtualized ECC allows the usage of arbitrary pages for ECC, it requires a layer of ECC address translation. The implementation in this dissertation avoids this additional hardware and complexity by allocating a contiguous ECC region. The ECC address for any data block can then easily be computed with a simple offset calculation. Second, the ECC region baseline uses

a wider error-correcting code that can protect an entire block with a single code word. A wide (523,512) code is used for the baseline to ensure a fair comparison with COP, since COP also uses wide codes. Since each 64-byte data block requires at least 11 bits for error protection, the contiguous ECC region is allocated with a 2-byte entry per data block to facilitate addressing. To improve the performance of this approach as well as for COP-ER, ECC metadata is cached in the L3.

As an additional comparison point, an embedded ECC approach as suggested by Zheng et al. was also modeled [81]. This approach locates the ECC parity bits in the same DRAM row as the data. When an open-row policy is used (as in this evaluation), this approach can substantially reduce the ECC access latency. This evaluation uses a DRAM configuration with 1024-byte rows, which translates to 16 64-byte blocks. Thus, one block is reserved per row to hold the ECC metadata for the other blocks. Incorporating an ECC block per row without changing the DRAM architecture requires an address "translation" before accessing DRAM to make sure that addresses from the processor skip over the special ECC block. Depending on the implementation of this translation, it could take up to a cycle. In this evaluation, a best-case scenario was modeled, and no translation penalty was imposed. In order for the L3 to cache ECC metadata, all addresses used in the L3 must be the translated version.

For all performance simulations, a 4-core system was modeled with private L2 caches and a shared L3, with other details shown in Table 6.1. For SPEC2006,

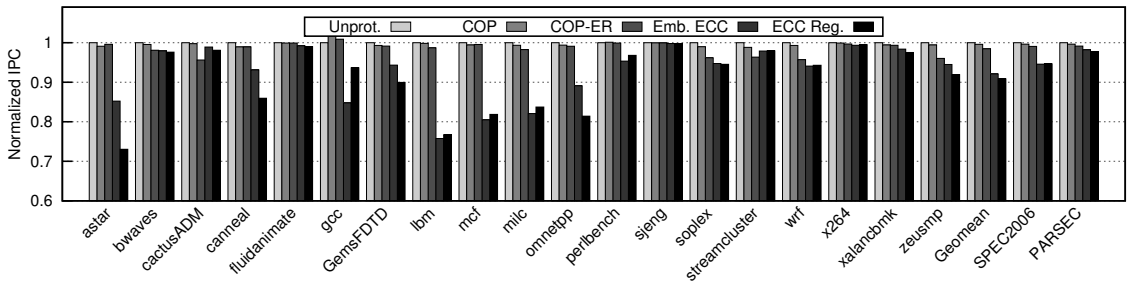


Figure 6.5: Performance of COP compared to other approaches. COP and COP-ER slightly increase the memory access latency, but put less pressure on the memory system.

a copy of the of the same application was run on each core, while for PARSEC the 4-threaded version of each program was run. All simulations therefore run 4 billion instructions in total. For the COP and COP-ER configurations, an additional decode/decompress latency of 4 cycles was assumed.

Figure 6.5 shows the performance results comparing COP and COP-ER to an unprotected system and the different in-memory ECC approaches. For COP, performance is slightly degraded as a result of the increased memory latency due to decompression. The performance of COP-ER, which protects all data, is slightly worse than COP due to the extra memory accesses to retrieve check bits for incompressible blocks. Because COP-ER places less pressure on the memory system (due to ECC region accesses), it degrades performance much less than the ECC region baseline. As shown, the embedded ECC approach generally performs better or on par with the ECC region. In some cases, embedded ECC performs worse than ECC region because it is less space-efficient, reserving more ECC space per block. For the applications shown, COP-ER performs about 8% better than the

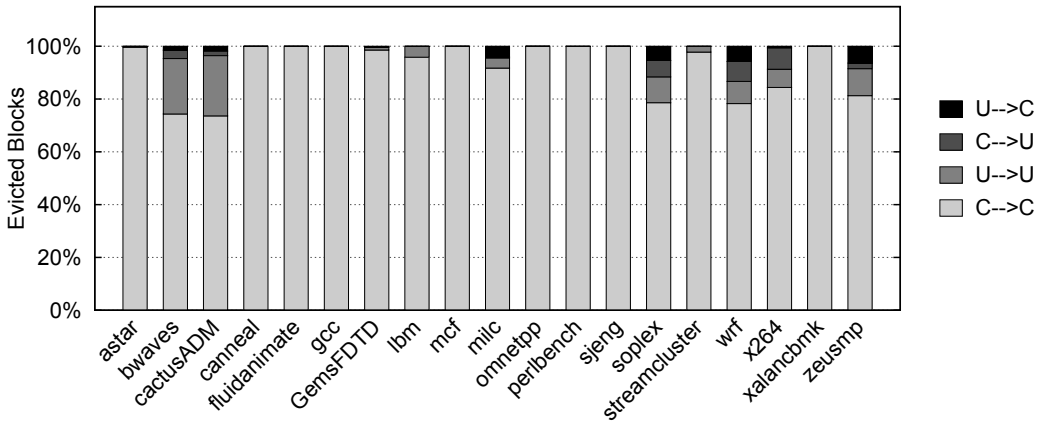


Figure 6.6: Compressibility change for evicted (dirty) blocks. The difference in compressibility is shown between when the block was loaded from DRAM and when it is written back. For instance, U–>C means a block was loaded from DRAM uncompressed but written back compressed.

ECC region alone or embedded ECC while providing the same error coverage.

A change in a block's compressibility while it is in the cache can affect the layout of COP-ER's ECC region. Figure 6.6 shows changes in compressibility while blocks are resident in the cache. Data is shown only for blocks that are modified and then evicted from the L3, requiring a write to DRAM. As shown, the compressibility of the majority of blocks does not change. In other words, they are compressible when loaded and still compressible when evicted or incompressible when loaded and still incompressible when evicted. The remaining blocks transition for compressible to incompressible, or vice-versa. Such transitions are in the minority however. Note that the breakdown shown does not include invalidation of clean blocks for which compressibility cannot not transition.

Although COP-ER relies on an ECC region to protect incompressible blocks, it

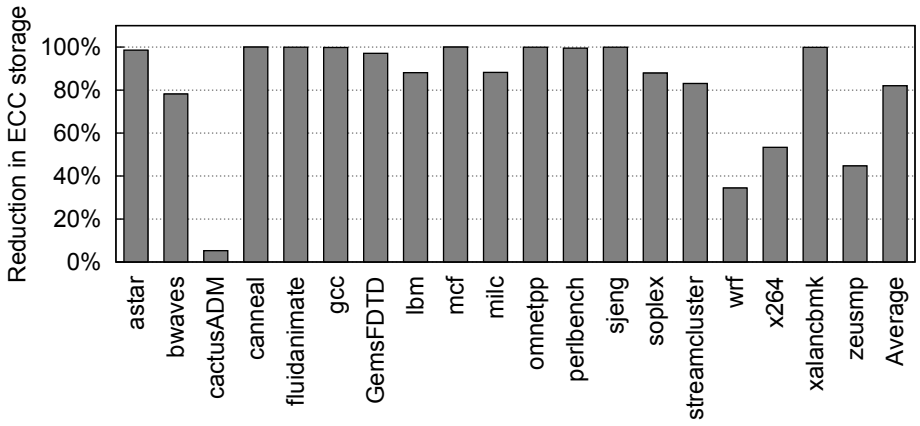


Figure 6.7: Reduction in ECC region size using COP-ER compared to using a static ECC region.

is able to significantly reduce the ECC storage overhead since the ECC region does not need to store ECC for all blocks. The amount of ECC storage required by each application was computed for both COP-ER and the static ECC region. Figure 6.7 shows the reduction in ECC storage space for each benchmark. As shown, COP-ER can reduce the space requirements by 80% on average.

6.5 Summary

This chapter presents an evaluation of COP. Results show that combining simple compression techniques allows compression of 94% of blocks on average. When COP is employed, this results in a 93% reduction in the soft error rate with negligible performance impact, while other approaches substantially degrade performance. When all data is protected using COP-ER, the performance impact remains very slight. It is demonstrated the COP-ER requires on average 80% less storage space than prior approaches.

7 CONCLUSION

As technology trends continue, future microprocessors will be required to dedicate more resources to offset reliability concerns such as process variations and soft errors. As these issues grow, there will be a strong incentive for designers to develop cost-effective solutions, allowing continued improvements in areas such as performance and power consumption. Because a large portion of modern systems is devoted to storage in the form of on-chip SRAM caches or off-chip DRAM, it is particularly important to efficiently protect these elements. This need is further increased since maximizing the storage density of these elements can come at the cost of reliability. To address bit failures in SRAM caches, prior work has suggested various disabling schemes, although such approaches can impact performance as cache space is reduced [1, 37, 77]. To reduce the cost of DRAM reliability, prior work has suggested embedding ECC in standard DIMMs, reducing the overall memory space and hurting performance [79, 81].

This dissertation proposes the idea of microarchitectural patching, a technique that can efficiently repurpose existing storage structures in the interest of enhancing system reliability. The technique works by exploiting the natural redundancy that exists between structures like caches, MSHRs, store queues, micro-op caches, and main memory. By modifying the retention policy for entries in these structures, existing data movement mechanisms are exploited to promote improved reliability

or performance. Reusing existing resources for this purpose avoids the additional circuitry and structures that would otherwise be needed to accomplish these goals. This dissertation proposes two techniques that use μarch patching to further the reliability of high-performance processors.

An approach called iPatch is proposed to enhance the energy and performance benefits of aggressive voltage scaling, which can cause SRAM failures. Existing proposals to guarantee correctness at low voltages rely on techniques like subblock disabling, which can degrade performance and negate energy savings. iPatch uses μarch patching to patch disabled parts of L1 caches, avoiding the higher-latency L2 accesses that would otherwise be required. By relying on existing components in the processor's memory datapath, iPatch is noninvasive and requires no changes to performance- or latency-critical structures or circuits. Results show that iPatch enables energy savings at high cell failure rates as well as an 18% average reduction in EDP compared to prior work when 0.5% of SRAM cells are failing.

This dissertation also proposes COP, which protects non-ECC DRAM DIMMs using compression to embed ECC into each memory block. This approach avoids the cost and energy overhead of ECC DIMMs by adding a small amount of logic to the memory controller. Prior proposals to protect non-ECC DIMMs sacrifice memory space to store parity or performance due to extra memory transactions. COP is able to distinguish between compressed data and uncompressed data by checking for the added ECC. μArch patching is used to ensure that application

data cannot masquerade as ECC code words. Several low-complexity compression schemes are also explored for use with COP with the goal of freeing a small amount of space per block (just enough to accommodate the ECC bits). A very effective hybrid scheme is proposed that achieves 94% compressibility over the set of evaluated workloads. An evaluation of COP shows a 93% reduction in the soft error rate over a baseline non-ECC DRAM system. To optionally extend ECC protection to incompressible blocks, a small region of main memory is used to store ECC for those blocks. This scheme (COP-ER) displaces a few bytes of from each incompressible block to add a pointer that indexes into the ECC region.

7.1 Future Work

This dissertation proposed two approaches using μarch patching to efficiently engineer reliable processors. There are also a number of potential avenues of future work that apply similar concepts in the context of reliability.

7.1.1 Patching the Memory Hierarchy

μArch patching could be extended to other parts of the memory hierarchy. For instance, a L1 cache could patch a lower-level cache. This approach would be particularly useful if the lower-level cache used a larger block size. An aggressive implementation could also use the same structures employed by iPatch to patch a

fault in a lower-level cache. Coordinating patching in this approach could be more complicated, however, especially if main memory is to be kept error-free.

7.1.2 Making COP More Robust

To tolerate multiple bit errors, COP could be modified to use a stronger error-correcting code. For instance, it could be updated to use a chipkill code that can correct any given chip failure. This modification would require more compression or a change in COP's compression-tracking approach. Given only four bytes for parity, only two single-symbol-correcting code words would fit in a block. To check for uncompressed data, an invalid code word could trigger an ECC region access and a backpointer in the ECC entry could be verified.

7.1.3 Tolerating Process Variation with COP-ER

Assuming a number of process variation-induced hard failures in DRAM, a variant of COP could be used to tolerate them. Because failures are likely to be spread out, it is more likely for memory blocks to contain a single failure than multiple. COP-ER could be used to correct the single failures with a special scheme to track and correct the less-common blocks with multiple failures.

BIBLIOGRAPHY

[1] J. Abella et al. "Low Vccmin Fault-tolerant cache with highly predictable performance". In: *Proc. 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*. 2009, pp. 111–121.

[2] J. Abella et al. "RVC: A mechanism for time-analyzable real-time processors with faulty caches". In: *Proc. 6th Int. Conf. on High Performance and Embedded Architectures and Compilers*. Heraklion, Greece, 2011, pp. 97–106.

[3] A. R. Alameldeen and D. A. Wood. "Frequent pattern compression: A significance-based compression scheme for L2 caches". In: *Technical Report 1500, Deptment of Computer Sciences, University of Wisconsin-Madison* (2004).

[4] A. Alameldeen and D. Wood. "Adaptive cache compression for high-performance processors". In: *Proc. 31st International Symposium on Computer Architecture*. 2004, pp. 212–223.

[5] A. Alameldeen et al. "Energy-efficient cache design using variable-strength error-correcting codes". In: *Proc. 38th Annual Int. Symp. on Computer Architecture*. 2011, pp. 461–471.

[6] American National Standard for Information Systems. *Coded character sets - 7-bit American national standard code for information interchange (7-bit ASCII)*. 1986.

[7] A. Ansari et al. "Archipelago: A polymorphic cache design for enabling robust near-threshold operation". In: *Proc. 17th Int. Symp. on High Performance Computer Architecture*. Feb. 2011, pp. 539–550.

[8] A. Ansari et al. "ZerehCache: Armoring cache architectures in high defect density technologies". In: *Proc. 42nd Annual IEEE/ACM Int. Symp. on Microarchitecture*. Dec. 2009, pp. 100–110.

[9] R. Baumann. "Soft errors in advanced computer systems". In: *Design Test of Computers, IEEE* 22.3 (2005), pp. 258–266.

[10] A. Bhavnagarwala, X. Tang, and J. Meindl. "The impact of intrinsic device fluctuations on CMOS SRAM cell stability". In: *IEEE Journal of Solid-State Circuits* 36.4 (Apr. 2001), pp. 658–665.

[11] N. Binkert et al. "The gem5 simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7.

[12] S. Borkar et al. "Parameter variations and impact on circuits and microarchitecture". In: *Proc. 40th Annual Design Automation Conference (DAC '03)*. Anaheim, CA, USA: ACM, 2003, pp. 338–342.

[13] K. Bowman, S. Duvall, and J. Meindl. "Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration". In: *IEEE J. Solid-State Circuits* 37.2 (Feb. 2002), pp. 183–190.

[14] T. E. Carlson, W. Heirman, and L. Eeckhout. "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: ACM, 2011, 52:1–52:12.

[15] L. Chang et al. "An 8T-SRAM for variability tolerance and low-voltage operation in high-performance caches". In: *IEEE Jounal of Solid-State Circuits* 43.4 (2008), pp. 956–963.

[16] G. Chen et al. "Yield-driven near-threshold SRAM design". In: *Proc. Int. Conf. on Computer-Aided Design*. Nov. 2007, pp. 660–666.

[17] L. Chen, Y. Cao, and Z. Zhang. "Free ECC: An efficient error protection for compressed last-level caches". In: *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. 2013, pp. 278–285.

[18] Y. Choi et al. "MAEPER: Matching access and error patterns With error-free resource for low Vcc L1 cache". In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 21.6 (2013), pp. 1013–1026.

[19] E. Chun, Z. Chishti, and T. N. Vijaykumar. "Shapeshifter: Dynamically changing pipeline width and speed to address process variations". In: *Proc. 41st Annual IEEE/ACM Int. Symp. on Microarchitecture*. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 411–422.

[20] T. J. Dell. "A white paper on the benefits of chipkill-correct ECC for PC server main memory". In: *IBM Microelectronics Division* (1997), pp. 1–23.

[21] J. Dusser, T. Piquet, and A. Seznec. "Zero-content augmented caches". In: *Proc. 23rd International Conference on Supercomputing*. ICS '09. Yorktown Heights, NY, USA: ACM, 2009, pp. 46–55.

[22] A. Dutta and N. A. Touba. "Synthesis of nonintrusive concurrent error detection using an even error detecting function". In: *Proc. Int. Test Conf.* 2005, pp. 1059–1066.

[23] P. Elakkumanan, K. Prasad, and R. Sridhar. "Time redundancy based scan flip-flop reuse to reduce SER of combinational logic". In: *Proc. Int. Symp. Quality Electronic Design*. 2006, pp. 619–624.

[24] D. Ernst et al. "Razor: a low-power pipeline based on circuit-level timing speculation". In: *Proc. Int. Symp. on Microarchitecture*. 2003, pp. 7–18.

[25] E. Fujiwara. *Code Design for Dependable Systems: Theory and Practical Application*. Wiley-Interscience, 2006. ISBN: 0471756180.

[26] D. Genbrugge, S. Eyerman, and L. Eeckhout. "Interval simulation: Raising the level of abstraction in architectural simulation". In: *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. 2010, pp. 1–12.

[27] P. Greenhalgh. *big.LITTLE processing with ARM Cortex-A15 & Cortex-A7*. 2011.

[28] E. Gunadi and M. H. Lipasti. "A position-insensitive finished store buffer". In: *Proc. 25th Int. Conf. on Computer Design*. Oct. 2007, pp. 105–112.

[29] M. Hsiao. "A class of optimal minimum odd-weight-column SEC-DED codes". In: *IBM Journal of Research and Development* 14.4 (1970), pp. 395–401.

[30] Z. Hu et al. "Microarchitectural techniques for power gating of execution units". In: *Proc. Int. Symp. on Low Power Electronics and Design*. Newport Beach, California, USA: ACM, 2004, pp. 32–37.

[31] "International technology roadmap for semiconductors, 2009 edition". In: *Semiconductor Industry Association* (2009).

[32] S. Jahinuzzaman, D. Rennie, and M. Sachdev. "A soft error tolerant 10T SRAM bit-cell with differential read capability". In: *IEEE Trans. Nuclear Science* 56.6 (2009), pp. 3768–3773.

[33] A. Jas and N. Touba. "Test vector decompression via cyclical scan chains and its application to testing core-based designs". In: *International Test Conference*. 1998, pp. 458–464.

[34] I. Kim et al. "Built in self repair for embedded high density SRAM". In: *Proc. Int. Test Conf.* Oct. 1998, pp. 1112 –1119.

[35] J. Kim, M. Sullivan, and M. Erez. "Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory". In: *Proc. 21st International Symposium on High Performance Computer Architecture*. 2015.

[36] J. Kulkarni, K. Kim, and K. Roy. "A 160 mV robust Schmitt trigger based subthreshold SRAM". In: *IEEE Journal of Solid-State Circuits* 42.10 (Oct. 2007), pp. 2303–2313.

[37] N. Ladas, Y. Sazeides, and V. Desmet. "Performance-effective operation below Vcc-min". In: *Proc. 2010 IEEE Int. Symp. on Performance Analysis of Systems Software*. Mar. 2010, pp. 223–234.

[38] H. Li et al. "Deterministic clock gating for microprocessor power reduction". In: *Proc. 9th Int. Sypm. on High-Performance Computer Architecture*. Feb. 2003, pp. 113–122.

[39] S. Li et al. “McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures”. In: *Proc. 42nd Annual IEEE/ACM Int. Symp. on Microarchitecture*. Dec. 2009, pp. 469–480.

[40] S. Li et al. “System implications of memory reliability in exascale computing”. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’11. Seattle, Washington: ACM, 2011, 46:1–46:12.

[41] X. Liang and D. Brooks. “Mitigating the impact of process variations on processor register files and execution units”. In: *Proc. 39th Annual IEEE/ACM Int. Symp. on Microarchitecture*. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 504–514.

[42] X. Liang, G.-Y. Wei, and D. Brooks. “ReVIVaL: A variation-tolerant architecture using voltage interpolation and variable latency”. In: *Proc. 35th Annual Int. Symp. on Computer Architecture*. ISCA ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 191–202.

[43] T. Mahmood and S. Kim. “Realizing near-true voltage scaling in variation-sensitive L1 caches via fault buffers”. In: *Proc. 14th Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*. Oct. 2011, pp. 85–94.

[44] H. Mahmoodi, S. Mukhopadhyay, and K. Roy. “Estimation of delay variations due to random-dopant fluctuations in nanoscale CMOS circuits”. In: *IEEE J. Solid-State Circuits* 40.9 (Sept. 2005), pp. 1787–1796.

[45] M. Manoochehri, M. Annavaram, and M. Dubois. “CPPC: Correctable parity protected cache”. In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA ’11. San Jose, California, USA: ACM, 2011, pp. 223–234.

[46] B. Maric, J. Abella, and M. Valero. “ADAM: An efficient data management mechanism for hybrid high and ultra-low voltage operation caches”. In: *Proc. Great Lakes Symp. on VLSI*. Salt Lake City, Utah, USA, 2012, pp. 245–250.

[47] B. Maric, J. Abella, and M. Valero. “APPLE: Adaptive performance-predictable low-energy caches for reliable hybrid voltage operation”. In: *Proc. 50th Annual Design Automation Conference*. Austin, Texas, 2013, 84:1–84:8.

[48] B. Maric, J. Abella, and M. Valero. “Efficient cache architectures for reliable hybrid voltage operation using EDC codes”. In: *Proc. Conf. on Design, Automation and Test in Europe*. Grenoble, France, 2013, pp. 917–920.

[49] S. Mitra et al. “Combinational logic soft error correction”. In: *Proc. IEEE Int. Test Conf.* 2006, pp. 1–9.

[50] K. Mohanram and N. Touba. “Cost-effective approach for reducing soft error failure rate in logic circuits”. In: *Test Conference, 2003. Proceedings. ITC 2003. International*. Vol. 1. Oct. 2003, pp. 893–901.

[51] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. “The soft error problem: an architectural perspective”. In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. HPCA ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 243–247.

[52] M. Nicolaidis. “Time redundancy based soft-error tolerance to rescue nanometer technologies”. In: *Proc. VLSI Test Symp.* 1999, pp. 86–94.

[53] A. Nieuwland, S. Jasarevic, and G. Jerin. “Combinational logic soft error analysis and protection”. In: *Int. On-Line Testing Symp.* 2006.

[54] M. Orshansky et al. “Impact of spatial intrachip gate length variability on the performance of high-speed digital circuits”. In: *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 21.5 (May 2002), pp. 544–553.

[55] D. J. Palframan, N. S. Kim, and M. H. Lipasti. “Mitigating random variation with spare RIBs: redundant intermediate bitslices”. In: *Proc. IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*. 2012, pp. 1–11.

[56] D. J. Palframan, N. S. Kim, and M. H. Lipasti. “Time redundant parity for low-cost transient error detection”. In: *Proc. Design, Automation and Test in Europe (DATE)*. 2011, pp. 1–6.

[57] D. J. Palframan, N. S. Kim, and M. H. Lipasti. “Resilient high-performance processors with spare RIBs”. In: *IEEE Micro* 33.4 (2013), pp. 26–34.

[58] G. Pekhimenko et al. “Base-delta-immediate compression: practical data compression for on-chip caches”. In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT ’12. Minneapolis, Minnesota, USA: ACM, 2012, pp. 377–388.

[59] G. Pekhimenko et al. “Linearly compressed pages: a low-complexity, low-latency main memory compression framework”. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 172–184.

[60] E. Perelman et al. “Using SimPoint for accurate and efficient simulation”. In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 31. 1. ACM. 2003, pp. 318–319.

[61] A. Phansalkar, A. Joshi, and L. John. “Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite”. In: *ACM SIGARCH Computer Architecture News*. Vol. 35. 2. ACM. 2007, pp. 412–423.

[62] R. R. Rao, D. Blaauw, and D. Sylvester. “Soft error reduction in combinational logic using gate resizing and flipflop selection”. In: *Proc. Int. Conf. on Computer-aided design*. San Jose, California, 2006, pp. 502–509.

[63] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. "DRAMSim2: A cycle accurate memory system simulator". In: *Computer Architecture Letters* 10.1 (2011), pp. 16–19.

[64] S. Sarangi et al. "EVAL: Utilizing processors with variation-induced timing errors". In: *Proc. 41st Annual IEEE/ACM Int. Symp. on Microarchitecture*. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 423–434.

[65] S. A. Seshia, W. Li, and S. Mitra. "Verification-guided soft error resilience". In: *Design Automation and Test in Europe*. 2007, pp. 1–6.

[66] S. Sethumadhavan et al. "Late-binding: Enabling unordered load-store queues". In: *Proc. 34th Annual Int. Symp. on Computer Architecture*. San Diego, California, USA, 2007, pp. 347–357.

[67] A. Shafiee et al. "MemZip: Exploring unconventional benefits from memory compression". In: *Proc. 20th International Symposium on High Performance Computer Architecture*. 2014, pp. 638–649.

[68] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. First edition. McGraw-Hill, 2005.

[69] P. Shivakumar et al. "Modeling the effect of technology trends on the soft error rate of combinational logic". In: *Proc. Dependable Systems and Networks*. 2002, pp. 389–398.

[70] A. Sinkar and N. S. Kim. "Analyzing and minimizing effects of temperature variation and BTI on active leakage power of power-gated circuits". In: *Proc. IEEE Int. Symp. on Quality Electronic Design*. 2010, pp. 491–796.

[71] B. Solomon et al. "Micro-operation cache: a power aware frontend for variable instruction length ISA". In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 11.5 (2003), pp. 801–811.

[72] J. Suh et al. "Soft error benchmarking of l2 caches with PARMA". In: *SIGMETRICS Perform. Eval. Rev.* 39.1 (2011), pp. 85–96.

[73] R. Teodorescu et al. "Mitigating parameter variation with dynamic fine-grain body biasing". In: *Proc. 40th Annual IEEE/ACM Int. Symp. on Microarchitecture*. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 27–42.

[74] A. Tiwari, S. R. Sarangi, and J. Torrellas. "ReCycle: pipeline adaptation to tolerate process variation". In: *Proc. 34th Annual Int. Symp. on Computer Architecture*. ISCA '07. San Diego, California, USA: ACM, 2007, pp. 323–334.

[75] J. Tschanz et al. "Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage". In: *Proc. IEEE Int. Solid-State Circuits Conf.* Vol. 1. ISSCC 2002. 2002, pp. 422–478.

[76] O. Unsal et al. “Impact of parameter variations on circuits and microarchitecture”. In: *IEEE Micro* 26.6 (2006), pp. 30–39.

[77] C. Wilkerson et al. “Trading off cache capacity for reliability to enable low voltage operation”. In: *Proc. 35th Annual Int. Symp. on Computer Architecture*. 2008, pp. 203–214.

[78] J. Yang, Y. Zhang, and R. Gupta. “Frequent value compression in data caches”. In: *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 33. Monterey, California, USA: ACM, 2000, pp. 258–265.

[79] D. H. Yoon and M. Erez. “Virtualized and flexible ECC for main memory”. In: *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XV. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 397–408.

[80] G. Zhang et al. “A novel single event upset hardened SRAM cell”. In: *IEICE Electronics Express* 9.3 (2012), pp. 140–145.

[81] H. Zheng et al. “Mini-rank: Adaptive DRAM architecture for improving memory power efficiency”. In: *Proc. 41st IEEE/ACM International Symposium on Microarchitecture*. 2008, pp. 210–221.