Reliable and High-Performance Architecture for Nanoscale Integrated Systems

Shuo Wang, PhD

University of Connecticut, 2010

As technology scaling continues, computer systems are facing the challenge of reliability degradation. It is projected that the fault rates of nanoelectronic devices will be several orders of magnitude higher than that of conventional CMOS technology. How to build reliable systems from unreliable devices is becoming an increasingly challenging issue. Meanwhile, with abundant devices rendered by technology scaling, it is also important to convert massive computing horsepower to high performance. However, the requirements of reliability and performance are often competing for hardware resources. This compels us to find solutions to address the dual challenges in a unified manner.

This dissertation explores the architectural design of nanoscale integrated systems to address multiple challenges in reliability and performance. Several novel solutions are proposed for the design of memories and computing/signal processing systems. These solutions open up opportunities for design space exploration along many new dimensions to unfold the full potential of nanoscale integrated systems.

Reliable and High-Performance Architecture for Nanoscale Integrated Systems

Shuo Wang

B.S., Beihang University, 2002
M.S., Beihang University, 2005

A Dissertation
Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy
at the
University of Connecticut

2010

APPROVAL PAGE

Doctor of Philosophy Dissertation

Reliable and High-Performance Architecture for Nanoscale Integrated Systems

Presented by

Shuo Wang, B.S., M.S.

Major Advisor

Lei Wang

Associate Advisor

Faquir Jain

Associate Advisor

John Chandy

University of Connecticut
2010

Acknowledgements

I sincerely appreciate my advisor Lei Wang for his excellent guidance and tremendous support through out my PhD study. His sharp insight and research attitude set me an perfect example, from which I will continue to benefit in my future career. I would like to thank my associate advisors Faquir Jain and John Chandy for their advice and help. I also wish to thank professor Mohammad Tehranipoor, Zhijie Shi, and Yunsi Fei for their comments and suggestions on my research.

My thanks also go to my colleagues and friends: Weiguo Tang, Junxia Ma, Jianwei Dai, Niral Patel, Sumit Narayan, and Xin Tian. It has been joyful to work with them and each of them have been an inspiration.

Lastly, I am grateful to my parents for their unconditional love and constant sacrifice for so many years. Also, I want to thank my wife Ying for her support and inspiration. I feel extremely lucky to have her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Since the invention of CMOS-based integrated circuits (IC), computer system design has reaped a dramatic improvement in computational performance. The key enabling technologies are a combination of advances in semiconductor process and design methodology, and innovations of architecture/microarchitecture. As technology scaling continues, growing chip complexity introduces new challenges at both device and architectural levels. In particular, nanometer regime devices are approaching their physical limits, and precise control over design uniformity becomes extremely difficult [1].

Many novel nanoelectronic devices including carbon nanotubes [2], silicon nanowires [3], quantum-dot cellular automata [4] and resonant tunneling devices [5] have emerged as the potential computational substrates for nanoscale integration. However, the bottom-up stochastic assembly of nanoelectronic devices leads to substantial reliability degradation. Defect rates in nanoelectronic devices are projected to be in the range of $10^{-3} - 10^{-1}$, several orders of magnitude higher than conventional CMOS technology [51]. In addition, nanoelectronic devices are also vulnerable to soft (transient) errors caused by particle strikes and timing failures [123, 124], causing erroneous behaviors even more difficult to model and predict. Thus, building reliable nanoscale integrated systems is becoming a challenging problem that must be addressed at various levels of design hierarchy.

In addition, chip security is another important aspect of computer systems and has drawn more attentions recently. Well-designed cryptographic algorithms can provide strong protection on the secret information. However, cryptographic implementation always leaks certain information, such as erroneous behaviors, timing, power, and electromagnetic, etc., through side channels, which can be exploited to retrieve secret keys much more easily compared with algorithm weaknesses. This further exacerbates the challenges of building reliable computer systems.

Meanwhile, with abundant devices rendered by technology scaling, it is important to convert the massive computing horsepower effectively to high performance. However, this is difficult especially because the various requirements of reliability and performance are often conflicting. In particular, reliability requires certain forms of redundancy to detect possible errors/faults, whereas performance desires parallel execution. Thus, these two requirements are competing for hardware resources. This compels us to find efficient hardware resource management and architecture-level solutions to balance these requirements and to achieve careful tradeoffs. This dissertation explores the design space to address the above challenges.

1.2 Thesis Contributions

This dissertation contributes to the field of computer engineering by developing novel techniques to address the crucial design challenges of building reliable and high-performance nanaoscale systems. Different from existing solutions, these solutions seek to satisfy the different requirements in a unified manner. This promising research direction can lead to the advancement of nanoscale integrated systems and can unfold the full potential of these systems. The following published work has been done as the contributions in this direction:

- Proposed novel memory microarchitectures [22, 24, 25, 26, 27, 33] to address the complicated tradeoffs among requirements for error tolerance, performance, power, and bandwidth usage.

- Proposed architectural level solutions and resource management policies [34, 28] for nanoscale computing systems to efficiently manage abundant nanoelectronic devices for the enhanced reliability and performance.
- Developed algorithms for nanoscale signal processing [31, 32] to achieve self-enabled defect-tolerance at reduced complexity as compared with traditional defect-mapping based techniques.
- Proposed microarchitecture/circuit approaches [29] to enhance chip security against power analysis attacks in an algorithm-independent manner.

1.3 Outline

This dissertation is organized as follows. Chapter 2 briefly summaries the related work in the literature. In chapter 3 through 7, the proposed solutions will be elaborated. Chapter 8 outlines some topics for further research.

Chapter 2

Related Work

In this chapter, some related work on the general topics of enhancing system performance, reliability, and chip security is briefly reviewed. More specific techniques related to our work can be found in the following chapters.

2.1 Performance

To improve the performance of computer systems, research in computer architecture aims at exploiting higher levels of parallelism in instruction processing. There is a clear trend towards multithreading and multicore. Simultaneous multithreading (SMT) [11], [12] and chip multiprocessors (CMP) [13], [14] exploit instruction-level and thread-level parallelism jointly to improve instruction throughput. Many techniques have been proposed to further improve the performance of SMT-CMP based architecture.

It should be pointed out that memory system remains a bottleneck for the performance improvement of computer systems. To shorten the gap between processor and memory speeds, many hardware/software approaches [15, 16] are proposed to reduce miss rate and miss penalty by prefetching data to exploit parallelism. In addition, pipelined caches, multibanked caches, and nonblocking caches are proposed to increase cache bandwidth.

As single-threaded computing still struggles with the memory wall, SMT-CMP based architectures are further challenged by the incompatible memory hierarchy that is developed originally for single-threaded computing. As each processing core in SMT-CMP systems is extended with the capability of multithreaded execution, competition for memory resources is exacerbated. This not only hurts the overall performance but also affects the energy efficiency.

Some existing research studies inter-thread conflicts by modeling [17] and software optimization [18, 19]. It was shown that these techniques might be only affordable in large and less active L2 or L3 caches in high-performance processors. For embedded multicore systems-on-chip with limited memory resources, the effectiveness of these techniques becomes marginal. Some other work on the topic of improving memory performance can be found in [46, 47, 48, 49], where cache memory is reconfigured to reduce conflicts and hence miss rates. It should be pointed out that the above techniques only focus on memory performance.

2.2 Reliability

Most existing work addresses reliability issues by exploiting certain forms of redundancy. At the device/circuit level, defect mapping techniques [6] were proposed to identify and replace defective devices with spare defect-free devices (hardware redundancy). However, these per-chip based test-then-reconfigure approaches are expensive and time-consuming for ultra high-density nanoscale integrated systems.

In another approach, error correcting codes (ECC) exploit information redundancy by implementing error checking bits for the data under protection. ECC's are widely employed to provide fault tolerance mainly for memories and interconnects [7, 8]. However, it might be difficult, if not impossible, for practical ways of implementing ECC in logic operations and instruction processing.

At the system level, redundancy based techniques, such as modular redundancy and multiplexing logic, were proposed [9] to achieve fault-tolerant logic operations. Recently, NAND multiplexing is improved with a rather low degree of redundancy [84]. However, the existing redundancy-based techniques all introduce large overheads. In addition, most existing work uses fixed modular redundancy, which is inflexible to cope with different reliability requirements at runtime.

2.3 Chip Security

Chip security is another important aspect of building reliable computer systems. Well-designed cryptographic algorithms can provide strong protection on the secret information. However, cryptographic implementations always leak certain information, such as timing, power, and electromagnetic, etc., through side channels, which can be exploited to retrieve secret keys much more easily compared with algorithm weaknesses. This raises a challenge on building reliable and secure computer systems.

Furthermore, chip security and error/fault tolerance are related. Fault analysis attacks [37], which insert errors into cryptographic systems then observe and analyze the erroneous behaviors to break the secret key, can be effectively prevented by reliability enhancing techniques [38, 39].

Besides fault analysis attacks, there are many other types of side-channel attacks. Examples include timing attacks [40], cache-based attacks [41, 42], power attacks [44], and electromagnetic analysis, etc. Countermeasures [43, 45] try to address the side-channel attacks by targeting the source of information leakage (cache conflicts and data-dependent power etc.).

Chapter 3

Soft Redundancy for Memory Design

3.1 Overview

Among all on-chip functional units, memory is particularly exposed to the reliability degradation in nanoscale integration. This is because memory circuits are built on minimum-geometry devices that are very sensitive to variations in process parameters, supply voltage and temperature. In addition, nanoscale memory circuits are also likely to suffer from unpredictable soft errors caused by particle strikes and timing failures.

Existing techniques, such as double/triple memory redundancy [117, 125] and ECC-based techniques, incur large area and performance overhead due to integer factor redundancy and complex error detection and recovery circuits. New solutions are needed to achieve better tradeoffs between reliability and performance. In our research, we exploit a unique memory behavior referred to as *soft (transient) redundancy*, which is created at runtime due to dynamic access patterns of the workloads. Specifically, some memory spaces may store irrelevant data or remain idle over time. They can be released and utilized to store copies of important data. Soft redundancy was initially studied in our prior work [22], where we exploited soft redundancy at cache subline level. Later, we extended this study to cache line level [25] and thread level [27]. Different from existing work, soft redundancy does not increase memory size (e.g., no extra memory cells for redundancy as in N-modular redundancy). Rather, it exploits the under-utilized memory spaces. Furthermore, soft redundancy can be adaptively allocated to improve error tolerance and access performance in a unified manner. Our solutions at different memory levels are elaborated in the following sections.

3.2 Soft Redundancy at Subline Level

3.2.1 Introduction

Traditional cache design focuses primarily on access performance such as reducing miss rate. Large cache lines take advantage of the inherent spatial locality to reduce cache misses. However, many irrelevant data are likely to be fetched as well, thereby wasting memory space and bandwidth. Sub-blocked cache [118] is beneficial to bandwidth usage by dividing each cache line into several addressable sub-blocks. During a miss, only one sub-block will be fetched, thereby reducing bandwidth requirement. However, sub-blocked cache tends to increase cache misses. In most designs, the sizes of cache line and sub-block are fixed, but the spatial locality varies during runtime under different workloads. This makes even more difficult to manipulate the tradeoff between access performance and bandwidth usage. To adapt to the varying locality of the program, some existing techniques dynamically adjust the block size [126], [127], number of sub-blocks [128], cache size [46], set-associativity [47], or replacement policy [129] during the course of execution. However, it should be pointed out that memory robustness (i.e., error tolerance) is not the primary focus of these above techniques.

The dual challenges of device-level reliability degradation and architecture-level performance gap make memory design tradeoffs more complicated but in favor of techniques which address memory robustness, access performance, and bandwidth in a unified manner. While the increasing physical and architectural complexities impose significant challenges to memory design, they also create opportunities for design space exploration along many new dimensions. Different from the aforementioned techniques, this subsection studies a unique memory behavior referred to as the *soft (transient) redundancy* at cache subline level. Exploiting soft redundancy at subline level, we are able to achieve self-adaptive and selective error control while improving memory performance. In contrast to conventional error-correction codes (ECC) which provide static error-control coverage to all the cache lines, our technique deliberately enhances the tradeoff between error tolerance and performance improvement by exploiting the fact that memory usually contains many irrelevant or to be replaced data, and thus providing full error-control coverage to all memory data is unnecessary. In addition, the proposed technique does not increase memory size (e.g., no extra memory cells for redundancy as in N-modular redundancy) but exploits the inherent transient redundancy for error protection. Our technique is suitable to a large body of general purpose computing applications. Note that for specific applications, such as multimedia processing, some techniques different from traditional ECC or N-modular redundancy for general scenarios have also been developed. One example is reconstruction of the missing macroblocks for error concealment [130]. Different from these techniques, our approach focuses specifically on memory systems (e.g., on-chip cache) which are vulnerable to soft errors.

Soft redundancy was initially proposed for on-chip memory design in our past work [22], where some preliminary results demonstrated joint improvement of error tolerance and access efficiency. In [24], a new soft-redundancy allocation mechanism is developed to allow more effective utilization of soft-redundant memory resources. This is achieved by distinguishing two types of cache misses: tag mismatches and subline misses, thereby enabling more accurate judgement on the status of cache lines to detect soft redundancy. In addition, a new runtime optimization method is proposed to adaptively utilize the soft-redundant memory for time-varying workloads.

In section 3.2.2, we present the memory architecture to exploit the soft redundancy at cache subline level. A design optimization method supporting runtime reconfiguration is developed in section 3.2.3. Simulation results are provided in section 3.2.4 to evaluate the effectiveness of the proposed technique.

3.2.2 Memory Architecture Exploiting Soft Redundancy

Soft Redundancy

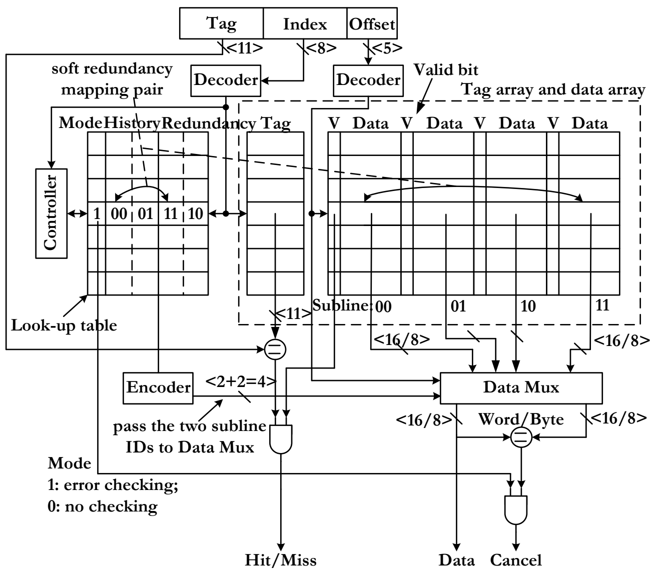


Figure 3.1: The generic architecture of the proposed soft-redundancy allocated memory. The numerical values of bit-widths are used in an example described in section 3.2.2 for the purpose of illustration.

Figure 3.1 shows the cache architecture allocating soft redundancy for memory access. The numerical values of bit-widths are used in an example described below for the purpose of illustration. In this example, the physical memory address consists of 24 bits. The total cache size is 32KB with set-associativity of four, and each cache line contains 32B. Only one way is shown in Fig. 3.1 as all the four ways are identical. In order to identify and utilize soft redundancy, each cache line is divided into multiple sublines (e.g., four sublines in Fig. 3.1 for the purpose of illustration). Note that the subline size is a key parameter that affects the distribution of soft redundancy, the coverage of error control, the miss rate, and the bandwidth usage of memory access. A general solution determining the subline size is given in section 3.2.3. A decoder (the top-left one in Fig. 3.1) uses the index to select a cache set and the cache lines in this set. Tag comparison will indicate whether it is a hit or a miss. Another decoder (the top-right one in Fig. 3.1) uses the lower bits of the offset to decode the word/byte address and the higher bits to determine which subline is accessed. This information will be used to determine the soft redundancy mapping pair. For example, in Fig. 3.1 subline 00 stores the original data and subline 11 stores the redundant copy. These two subline IDs are used by the data mux to select the original data and its redundant copy. The original data is read out directly. Meanwhile, the two copies are compared for error detection. If an error is detected, the read operation will be canceled and new data will be fetched from the lower level of memory.

A look-up table is introduced to monitor the subline status for soft redundancy detection and allocation. The look-up table is composed of three fields: (i) *mode field*, indicating the operation mode of each cache line, (ii) *history field*, storing the IDs of the sublines that experience misses most recently, and (iii) *redundancy field*, storing the IDs of the soft-redundant sublines. In general, if a cache line is divided into n sublines, both the history field and the redundancy field will contain the IDs of $n/2$ sublines. Any two sublines with IDs in the same positions of the two fields maintain a *soft-redundancy mapping pair*, and the mappings are updated at runtime based on access activities. In the example shown in Fig. 3.1, the history field of each cache line uses four bits to keep track of the two most recently missed sublines. For example, the value "00-01" shown in the history field indicates that misses occur most recently in the sublines "00" and "01". In the subsequent memory accesses, a miss in the subline "00" will not change the history field; whereas a miss in the subline "01" will update the history field to "01-00". If a miss occurs in any other sublines, e.g., subline "10" or "11", the history field will be updated to "10-00" or "11-00", respectively. Similar to the history field, the redundancy field uses four bits to store the IDs of soft-redundant sublines. For example, if the history and redundancy fields have "00-01" and "11-10", respectively, then subline pairs "00", "11" and "01", "10" are assigned as two soft-redundancy mapping pairs, i.e., the sublines "11" and "10" are utilized as soft redundancy to store the copies of sublines "00" and "01", respectively, for the purpose of error control.

Each cache line can be operated in two allocation modes, *error-checking* and *no-checking*, as indicated in the mode field of the look-up table. Cache lines in the no-checking mode are accessed as conventional cache, while those in the error-checking mode exploit soft redundancy for enhanced error control. The procedure of mode switching is explained through an illustration example as follows.

An Illustration Example

We use a sequence of memory accesses shown in Fig. 3.2 as an example to explain the operation of soft-redundancy allocation. Initially, this cache line is in the error-checking mode by default. The mode field is set to “1” indicating this mode. The history field stores “00-01” for the two default subline IDs “00” and “01”, while the redundancy field stores “10-11”, indicating that subline pairs “00”, “10” and “01”, “11” are two soft-redundancy mapping pairs. Note that these initial values can be chosen arbitrarily as soft-redundancy allocation will converge during the course of execution. The valid bits of all sublines are set to “0” before fetching any data into the cache.

We classify the cache misses into two types: one is due to tag mismatch, and the other occurs when the tag matches but the valid bit of the accessed subline is not set (referred to as subline miss). These two types of misses are treated differently in look-up table update and mode switching. Assume that the first miss in this cache line is a tag mismatch in the subline “01” (step 2 in Fig. 3.2). Since this is a tag miss, the current cache line is likely to undergo a change in access pattern. The operation mode is thus reset to error-checking so that the proposed technique can promptly detect soft redundancy once a new access pattern starts. The history and redundancy fields are updated together as a single field to “01-00-10-11”. Instead of fetching the entire cache line (the content of which would be, for example, A, B, C, and D in the four sublines, respectively), our technique only fetches data for sublines “01” and “00” listed in the history field. The sublines “10” and “11” listed in the redundancy field are utilized to store the copies of sublines “01” and “00”, respectively. The valid bits of subline “01” and “00” are set to “1”, while the other two sublines remain invalid (i.e., being used as soft redundancy).

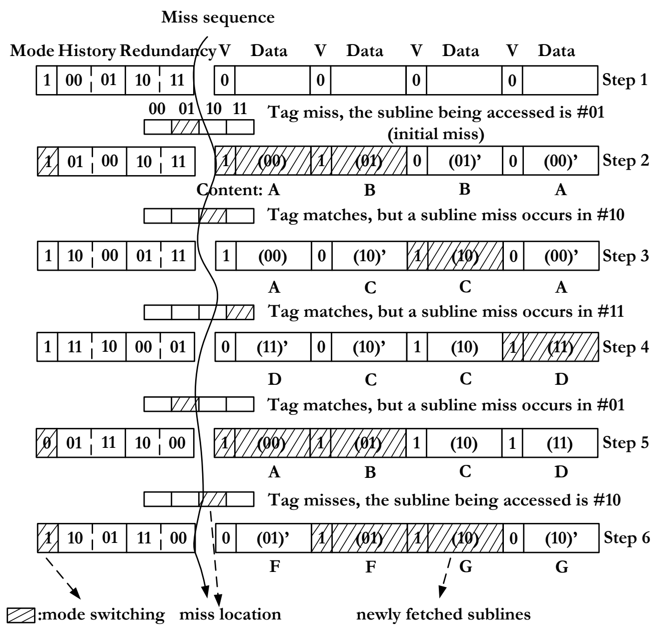

Miss sequence
Mode History Redundancy V Data V Data V Data V Data
Step 1
Tag miss, the subline being accessed is #01 (initial miss)
Step 2
Content: A B B A
Tag matches, but a subline miss occurs in #10
Step 3
Tag matches, but a subline miss occurs in #11
Step 4
Tag matches, but a subline miss occurs in #01
Step 5
Tag misses, the subline being accessed is #10
Step 6
▨:mode switching
miss location
newly fetched sublines

Figure 3.2: An example of soft-redundancy allocation with a specific miss sequence assumed.

Assume that the next miss occurs in subline “10”, which is not listed in the history field (step 3). Since only the history field in the error-checking mode contains the IDs of valid sublines, this miss is caused by subline invalidity, i.e., the valid bit of subline “10” is not set. When subline misses occur, the history field and redundancy field are updated separately. In this example, “10” is moved to the first position in the history field, so is “01” in the redundancy field to maintain the same soft-redundancy mapping pair. Other subline IDs will shift their positions within the corresponding field if necessary. Since the subline miss in step 3 is the first time that the look-up table fails to predict, the cache line stays at the error-checking mode. The sublines listed in the redundancy field remain as soft redundancy to store the copies of the sublines listed in the history field for enhanced error tolerance.

After three consecutive subline misses that the look-up table fails to predict (steps 3-5), the memory access pattern becomes unpredictable due to runtime variations in spatial locality. Hence, the allocation mode is switched to no-checking at step 5. The sublines “00” and “01” listed in the redundancy field at the end of step 4 are released.

After fetching new data into these two sublines, the entire cache line becomes valid. Note that in this example the confidence level of mode switching is set to three consecutive subline misses. A general method for determining the confidence level and other key parameters will be given in section 3.2.3.

Assume that a tag miss occurs in subline “10” in the no-checking mode (step 6). This will reset the mode back to error-checking, and the history and redundancy fields are updated together to “10-01-11-00”. The sublines “10” and “01” are replaced with new data (the contents of which are G and F, respectively), while the sublines “11” and “00” are employed as soft redundancy to store the copies of sublines “10” and “01”, respectively.

General Algorithm for Soft-Redundancy Allocation

The above example demonstrates the underlying idea of soft-redundancy memory allocation. By monitoring runtime access activities, we can keep track of the locations (sublines) in a cache line that are accessed more frequently. If the spatial locality is stable, the history field can capture this behavior and predict future memory accesses. Accordingly, the sublines can be classified into two groups: frequently accessed and seldom used. For stable memory spatial locality, the frequently accessed sublines are likely to continue being accessed subsequently. The data in these sublines are presumably important and error protection is thus needed. On the other hand, the seldom-used sublines are most likely the ones containing irrelevant data. We can release these sublines and assign them as soft redundancy for error protection of frequently accessed sublines. Note that the access frequency alone is not effective enough for soft redundancy capturing, because the temporarily “inactive” sublines may still contain useful data while the frequently missed sublines may be occupied by the irrelevant data. Therefore, we take into account both the access history and the hit/miss information to direct the soft redundancy allocation. This is achieved by updating the history and redundancy fields based on the order of miss occurrence. As a result, the proposed technique is self-adaptive to runtime program variations. It also helps to reduce the bandwidth wasted on fetching the irrelevant data.

Algorithm 1 shows the general algorithm for identifying and allocating memory

Algorithm 1: Algorithm of soft-redundancy allocation. The allocation mode and look-up table update as well as data replacement policy are shown.

```
begin
    (part 1: allocation mode and look-up table update)
    while a miss occurs do
        if tag miss then
            $confidence\_counter = 0$
            update $history\_field$ and $redundancy\_field$ together
        end
        else if subline miss then
            if $confidence\_counter < 2c - 1$ then
                $confidence\_counter$ ++
            end
            update $history\_field$ and $redundancy\_field$ separately
        end
        if $confidence\_counter < c$ then
            $allocation\_mode$=error-checking
        else
            $allocation\_mode$=no-checking
        end
    end
    (part 2: data replacement policy)
    while replace do
        switch switching of allocation mode do
            (no-checking to no-checking transition will not happen)
            case error-checking to error-checking
                if tag miss then
                    fetch sublines listed in the $history\_field$ (1/2 line)
                    store copies to the soft redundancy mapping pairs
                end
                else if subline miss then
                    fetch the missed subline only (1 subline)
                    store a copy to the mapping pair
                end
            end
            case error-checking to no-checking
                if subline miss then
                    fetch all the invalid sublines (1/2 line)
                end
                (tag miss will not happen in this case, as tag miss always switchs the mode to
                error-checking)
            end
            case no-checking to error-checking
                if tag miss then
                    fetch sublines listed in the $history\_field$ (1/2 line)
                    store copies to the soft redundancy mapping pairs
                end
                (subline miss will not happen in this case, as all the subline is valid in no-checking mode)
            end
        end
    end
end
```

soft redundancy. There are two parts in the algorithm: one is regarding allocation mode and look-up table update, the other is regarding data replacement policy. The allocation mode is determined by the runtime access pattern predicted by the look-up table. The allocation mode switches according to the confidence level c on whether the access pattern could be sufficiently predicted or not. A counter is employed to direct the allocation mode switching. Two types of misses, i.e., tag mismatches and subline misses, lead to different operations on mode switching and data replacement. When a subline miss occurs, the counter value will be increased by one. Consecutive subline misses making counter value larger than the confidence level switch the mode from error-checking to no-checking. On the other hand, a tag miss indicates a possible change in access pattern. The counter is reset, switching the mode back to error-checking. The counter values within $[c+1, 2c-1]$ are reserved for runtime reconfiguration as discussed in section 3.2.3.

The data replacement policy is determined as follows. (i) In the error-checking mode, only the missed subline is replaced during a subline miss, whereas all the sublines listed in the history field are replaced during a tag miss. Meanwhile, copies of the newly fetched data are also stored in the sublines listed in the redundancy field according to the soft-redundancy mapping pairs. (ii) If the mode is switched to no-checking (caused by consecutive subline misses), only the sublines listed in the redundancy field are replaced making the entire cache line valid. Note that subline misses only occur in the error-checking mode because all the sublines are valid in the no-checking mode. (iii) If the mode is switched to error-checking (caused by tag misses only), only the sublines listed in the history field are replaced, thereby reducing the bandwidth usage by half. Since the bandwidth usage during a miss is either one subline or half of the cache line, the soft-redundancy allocation improves memory access efficiency. Note that the soft-redundancy allocation is performed simultaneously on a per-cache-line basis. Thus, the proposed technique can be directly applied to caches with different set-associative placements.

Error Control

The proposed technique relies upon a dynamic mapping strategy by utilizing the unused memory resources and thus enabling error control only when necessary, e.g., for the sublines being hit frequently where the data are most likely needed in subsequent accesses. Different from conventional cache operations, the proposed technique introduces an additional comparison between the soft-redundancy mapping pairs at the error-checking mode. Mismatches in comparison indicate data upsets and result in cancelation of the read-out data. Note that this additional comparison can be made off the critical path to minimize the performance overhead (see section 3.2.3). If errors are detected, a data refetch is performed to recover from any pattern of bit corruptions. There is a possibility that the soft-redundant copy is corrupted thereby introducing an additional miss. However, the performance overhead is negligible as shown in section 3.2.4.

In the past, soft errors are typically modeled as single-bit upsets (SBU). As semiconductor process being scaled into the nanometer domain, a single particle strike may potentially corrupt multiple memory bits, resulting in multiple-bit upsets (MBU). In addition, timing noise tends to cause MBU as well. Among the existing solutions, parity checking is considered as most effective for detecting SBU (and an odd number of errors), whereas widely-used Hamming code provides error detection for up to two bits of errors. ECC for more than two bits of errors is quite complicated and seldom used in memory design. The proposed technique could detect and recover multiple errors at any bits. Simulation results in section 3.2.4 demonstrate that the proposed technique achieves $10X$ improvement in error detection over the parity checking and Hamming code in both SBU- and MBU-dominant situations.

Access Efficiency

In addition to enhancing error control, the proposed technique also improves memory access efficiency. In general, a cache memory with low bandwidth usage and/or miss rate is considered as having high access efficiency. Conventional cache fetches entire cache line during a miss. Obviously, this wastes bandwidth and leads to high miss penalty. Sub-blocked cache fetches one sub-block each time to reduce bandwidth usage but may further increase the miss rate.

The proposed technique improves access performance by selectively fetching sublines according to history-based prediction. From Algorithm 1, if a cache line is in the error-checking mode, a miss will only result in the replacement of either one subline (if it is a subline miss) or half of sublines in the cache line (if it is a tag miss).

Also, only half of sublines in the cache line are fetched when the allocation mode switches between no-checking and error-checking. As a result, the bandwidth usage is reduced. Furthermore, in the no-checking mode when memory access pattern is not stable, the entire cache line will be valid. Hence, subline misses can be avoided thereby improving the access performance over the sub-blocked cache. Simulation results clearly demonstrate these benefits.

3.2.3 Design Optimization for Soft-Redundancy Allocated Memory

In this section, we present the design optimization for soft-redundancy allocated memory. We first discuss the hardware implementation and then develop a general method for parameter selection and runtime reconfiguration.

Implementation

In comparison with the traditional cache design, the soft-redundancy allocated memory introduces two additional components: a comparator for error detection and a look-up table with the control logic.

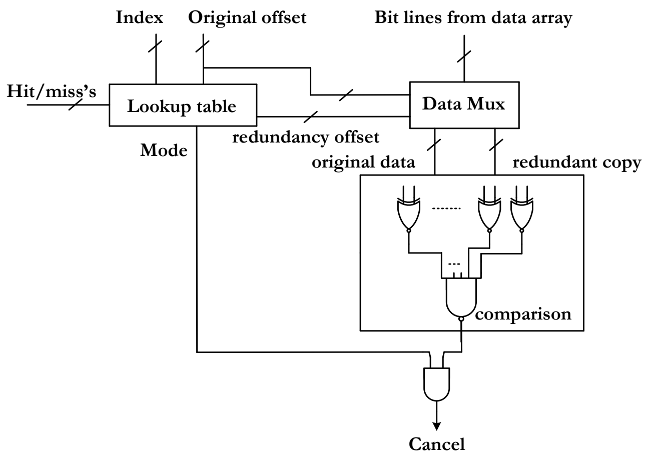
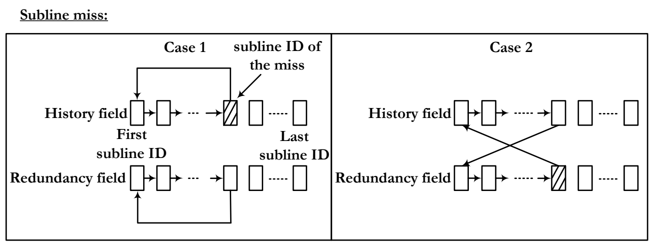


Figure 3.3: Logic schematic of error detection comparator (bit-widths can be found in Fig. 3.1).

The error detection comparator is shared by all cache lines at each read port. From the look-up table we can get the subline IDs of a soft-redundancy mapping pair. Possible soft errors can be detected through bitwise comparison between the two data copies from the corresponding sublines. The logic schematic of the error detection comparator is shown in Fig. 3.3, where the comparator is k-bit wide in general (k is the memory access data width, e.g., $k = 16$ for a word and $k = 8$ for a byte). Note that the error detection can be performed in parallel with the timing-critical read operations to avoid performance degradation. During a hit, the primary subline is read out directly without waiting for the result of comparison. If later the comparator reports data upsets, a cancelation signal will be asserted, notifying the execution unit to nullify the involved results and initiating a data replacement request.

The look-up table is addressed by the index address and the hit/miss information of each cache line. Each entry in the look-up table consists of one mode status bit plus a history field and a redundancy field storing the IDs of the primary sublines and soft-redundant sublines, respectively. These two fields consume $n\lceil \log_2 n \rceil$ bits in total, where n is the number of sublines in each cache line. The procedure of subline ID update in the look-up table is shown in Figs. 3.4(a) and (b) for four different cases. Note that we need to keep the order of subline IDs in the history field to follow their miss occurrence, i.e., the first ID must refer to the subline having the most recent miss (see the example in section 3.2.2). If a subline miss occurs (see Fig. 3.4(a)), the IDs of this subline and its soft-redundant pair will be moved to the first positions of the history field and redundancy field, respectively, while the other subline IDs will either shift to the right within the corresponding field or remain at the same location. This will maintain the same soft-redundancy mapping pairs between the two fields to avoid reshuffling the data in the sublines. On the other hand, if a tag miss occurs (see Fig. 3.4(b)), the history and redundancy fields will be updated together as a single field. Only the missed subline ID will be moved to the first position of the history field.

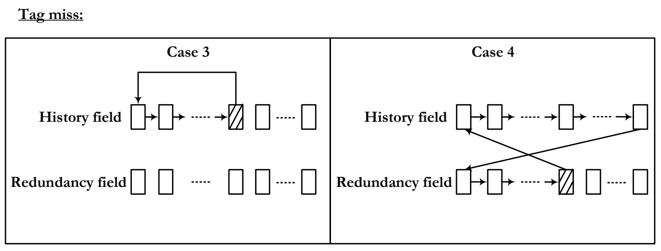
The control logic of the lookup table is responsible for mode transition. The counter in this control logic is designed with $2c$ states, where c is the confidence level.



Case 1: the subline ID of the miss is stored in the history field. In this case, the IDs of the missed subline and its soft-redundancy pair are moved to the first positions of each field. The other subline IDs either shift to the right or remain at the same positions as shown above.

Case 2: the subline ID of the miss is stored in the redundancy field. In this case, the IDs of the missed subline and its soft-redundancy pair are moved to the first positions of the other field. The other subline IDs either shift to the right or remain at the same positions as shown above.

(a)



Case 3: the subline ID of the tag miss is stored in the history field. In this case, the subline ID of the tag miss is moved to the first positions of the history field. The other subline IDs in the history field either shift to the right or remain at the same positions. IDs in the redundancy field stay at the same positions.

Case 4: the subline ID of the tag miss is stored in the redundancy field. In this case, the two fields are updated together. Specifically, the ID of the tag miss is moved to the first position. The other subline IDs either shift to the right or remain at the same position as shown above.

(b)

Figure 3.4: Illustration of look-up table update: (a) update procedure on a subline miss, (b) update procedure on a tag miss.

Thus, it is sufficient to express the counter by $\lceil \log_2 2c \rceil = 1 + \lceil \log_2 c \rceil$ bits.

Note that same as other error-tolerant techniques, the proposed technique relies upon the correctness in error detection and correction operations. The look-up table is much less complex and easy to be implemented with sufficient reliability margin. Thus, it is practically reasonable to assume that the error probability is much smaller than that of the cache lines. In addition, some simple error-control techniques such as parity checking may be able to further improve the robustness of look-up table. For these reasons, we will focus on the dominant errors in cache lines for the sake of demonstrating the essence of the proposed technique.

Parameter Selection

The confidence level c and the number of sublines n are the two key parameters that need to be optimized for soft-redundancy allocated memory. The selection of these two parameters must consider the hardware cost, error tolerance, and access efficiency in terms of miss rate and bandwidth usage.

Hardware Overhead

Consider a general case where each m-word cache line is divided into n sublines. The value of n needs to be an even number (usually power of two), which can be chosen between 2 and m.

To estimate the hardware overhead, we consider the logic complexity of error detection comparator and look-up table, the two new components in soft-redundancy allocated memory. The comparator is k-bit, where k is fixed by the memory data width. The look-up table contains $1 + n\lceil \log_2 n \rceil$ bits per cache line to store the mode status and the IDs of all sublines. The control logic has a counter with $1 + \lceil \log_2 c \rceil$ bits, where c is the confidence level. For a specific implementation, the hardware overhead is proportional to the logic complexity of these components. Let γ_1, γ_2 and γ_3 be the hardware cost per bit of the comparator, the history and redundancy fields, and the control logic, respectively, where the numerical values are determined by the specific physical implementation. The total hardware overhead of a memory with L cache lines can be expressed by

$$C(n, c) = k \cdot \gamma_1 + (1 + n\lceil \log_2 n \rceil) \cdot L \cdot \gamma_2 + (1 + \lceil \log_2 c \rceil) \cdot L \cdot \gamma_3. \tag{3.1}$$

Note that the overhead of the proposed technique is less than that of the traditional error-tolerant techniques such as ECC. The look-up table only needs to store the subline IDs, and the control logic and the comparator have a very simple structure. In comparison, many ECC implementations require additional memory cells for information redundancy and complicated logic such as finite field multipliers and division circuits.

Error Tolerance

The proposed technique enables error tolerance for those cache lines in the error-checking mode. The cache lines in the no-checking mode experience frequent misses. Thus, it is of low priority to provide error protection to those irrelevant and soon to-be-replaced data. The error-control coverage ratio, denoted as R, can be calculated as

$$R = \frac{MA_{error-checking}}{MA_{total}}, \tag{3.2}$$

where $MA_{error-checking}$ and MA_{total} are the number of memory accesses in the error-checking mode and the total number of memory accesses, respectively.

In the following analysis, we rewrite R as $R(n, c)$ because it is a function of parameters n and c. The value of $R(n, c)$ can be obtained by averaging over a group of benchmark programs, as discussed in section 3.2.4.

Access Efficiency

To evaluate the access efficiency, we consider both the miss rate and bandwidth usage. In the error-checking mode, the bandwidth usage can be reduced by exploiting history-based prediction on spatial locality, which enables fetching only the relevant portion of the cache line. In the no-checking mode where the memory does not show stable spatial locality, the cache lines work in the same way as conventional cache to avoid increase in subline misses. The self-adaptive mode transition enables better performance over the sub-blocked cache. We denote the miss rate and bandwidth usage as $M(n, c)$ and $B(n, c)$, respectively, to reflect the fact that they are functions of subline number n and confidence level c. The value of $M(n, c)$ can be calculated as $M(n, c) = \frac{miss\ count}{access\ count}$, and $B(n, c)$ can be estimated by the number of fetched words.

Joint Optimization for Hardware Cost, Error Tolerance, and Access Efficiency

The needs of enhancing error tolerance ($R(n,c)$), improving access efficiency ($M(n,c)$ and $B(n,c)$), and reducing hardware cost ($C(n,c)$) are usually competing with each other. For example, exploiting soft redundancy for high error tolerance may sacrifice access efficiency in terms of miss rate and bandwidth usage, while pursuing high access efficiency may increase hardware cost. Therefore, the optimal design should achieve the best tradeoff in a design space spanning over these correlated dimensions.

To enable a general approach, we first normalize these metrics by their maximum values, i.e.,

$$\widehat{C}(n,c) = \frac{C(n,c)}{C_{max}}, \tag{3.3}$$

$$\widehat{R}(n,c) = \frac{R(n,c)}{R_{max}}, \tag{3.4}$$

$$\widehat{M}(n,c) = \frac{M(n,c)}{M_{max}}, \tag{3.5}$$

$$\widehat{B}(n,c) = \frac{B(n,c)}{B_{max}}. \tag{3.6}$$

The optimal soft-redundancy allocation is the one that achieves the best tradeoff among error-control coverage ratio, access efficiency and hardware overhead. This can be expressed as

$$\exists(n_{opt}, c_{opt}) \in (n,c), s.t.,$$

$$\frac{(\widehat{R}(n_{opt},c_{opt}))^{W_r}}{(\widehat{C}(n_{opt},c_{opt}))^{W_c}(\widehat{M}(n_{opt},c_{opt}))^{W_m}(\widehat{B}(n_{opt},c_{opt}))^{W_b}}$$
$$= \max\left[\frac{(\widehat{R}(n,c))^{W_r}}{(\widehat{C}(n,c))^{W_c}(\widehat{M}(n_{opt},c_{opt}))^{W_m}(\widehat{B}(n_{opt},c_{opt}))^{W_b}}\right], \tag{3.7}$$

where W_r, W_c, W_m, and W_b are the weight factors that adjust the design priority. Specific values can be determined by the designers to best meet their goals. For example, if the four metrics are treated as equally important, the weight factors can be selected to have the same value. The solution to (3.7) can be obtained by iteration over all the possible combinations of (n, c)'s, where the results of error-control coverage ratio $R(n, c)$, miss rate $M(n, c)$, and bandwidth $B(n, c)$ are obtained along with the hardware cost $C(n, c)$ over a group of benchmark programs. Note that the optimization method is applicable to different memory systems.

Runtime Reconfiguration

The subline number n_{opt} and confidence level c_{opt} obtained from (3.7) are optimal with respect to the average effect of the benchmark programs. In practice, however, different programs may have different memory activities and performance requirements over time. Thus, it is necessary to reconfigure the soft-redundancy allocation at runtime. Note that changing the number of sublines is difficult because it is fixed by hardware after the n_{opt} is implemented. The confidence level, however, can be adjusted up to $2c - 1$ states.

As discussed in section 3.2.2, a high confidence level makes the memory stay in the error-checking mode longer. In general, the more accesses are in the error-checking mode, the higher error-control coverage ratio and less bandwidth usage are expected. This, however, may increase the cache misses, especially if the memory access pattern deviates away from that predicted by the current confidence level. Apparently, an optimal design would prefer the operation to be at the error-checking mode as much as possible but without increasing the miss rate. This can be achieved by runtime reconfiguration that monitors the miss count and adjusts the confidence level periodically. If during the most recent period the miss count drops by a predefined threshold, the dynamic confidence level c' is increased by one. Otherwise, if the miss count increase by a predefined threshold, the dynamic confidence level c' is decreased by one. Consequently, the reconfiguration dynamically adjusts the confidence level within the range of $[1, 2c - 1]$ through a direct correspondence with memory performance. The reconfiguration threshold is determined by the stability of memory access of the workload. If the access pattern is stable, a small threshold can help the confidence level quickly converge to an optimal level. Otherwise, a large threshold is needed to avoid the confidence level to be adjusted back and forth frequently, which may hurt the performance or the capability of error tolerance.

Note that the runtime reconfiguration is performed by software at a relatively long interval. In addition, it can be done after the data accesses or when there is no pending access. Thus, the performance overhead is negligible. When the confidence level is adjusted, it is transparent to the cache as a global setting. Therefore, no specific hardware is needed to support the runtime reconfiguration.

3.2.4 Evaluation and Discussion

In this section, we evaluate the proposed soft-redundancy allocated memory. We will compare the error tolerance, access efficiency and bandwidth usage with the conventional approaches.

The simulations were performed on a simulator based on the trace-driven simulator Dinero IV [113]. We use the SPEC CPU2000 [10] trace files collected from the Stream-Based Trace Compression (SBC) [114], where trace files of 23 benchmarks are available. The cache model is modified to support the proposed technique. The total cache size is $32KB$ with set-associativity of 4 and $32B$ each cache line. The replacement policy being used is the Least Recently Used (LRU). The conventional cache does not divide the cache line into sublines, whereas the sub-blocked cache and the proposed technique divide each cache line into multiple sublines as determined in section 3.2.4.

Design Optimization

We apply the optimization method as discussed in section 3.2.3 to determine the subline number n and confidence level c. Since each cache line contains 16 words (32 bytes), the possible subline number n is 2, 4, 8, or 16, and the confidence level c can be chosen between 1 and 8. Figure 3.5 shows the optimization results of different combinations of (n, c) with weight factors $W_r = 2, W_c = W_m = W_b = 1$, i.e., error tolerance is given the highest priority. The optimal parameters were found at $n_{opt} = 16$ and $c_{opt} = 2$.

Note that these optimal parameters account for the average effect of all the 23

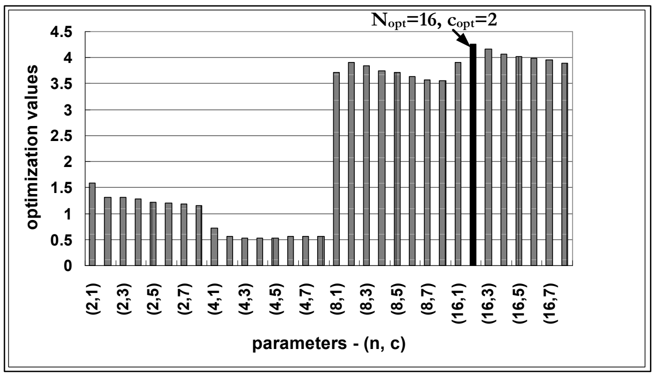


Figure 3.5: Parameter selection based on the average effect of 23 benchmarks.

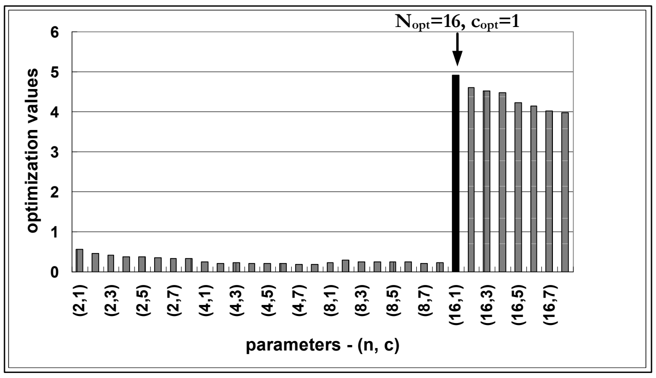


Figure 3.6: Parameter selection for the `vortex` benchmark.

benchmarks. If the design targets a specific class of applications, e.g., object-oriented database applications, the optimal parameters may be different. Figure 3.6 shows the example of running the `vortex` benchmark only. The configuration level is adjusted to $c_{opt} = 1$ for overall optimization of this application. This clearly shows the variations of memory access behavior across different applications. The runtime reconfiguration as discussed in section 3.2.3 can be quite effective to address this problem. We found that for the `vortex` benchmark the runtime reconfiguration leads to 12.3% and 4.0% reduction on miss rate and bandwidth usage, respectively, along with 8.6% increase on protection ratio as compared to the case without applying runtime reconfiguration. Similar results were observed in other benchmarks as well.

In the following simulations, we will start with the optimal parameters $n_{opt} = 16$

and $c_{opt} = 2$ and then employ runtime reconfiguration. The interval of runtime reconfiguration is every 1000 memory accesses with an adjustment threshold equal to 30%. This threshold indicates that the confidence level c' will be reduced (or increased) by 1 if the miss count in the current interval is 30% more (or less) than that in the previous interval (see section 3.2.3). It can be adjusted by software when necessary.

Error Tolerance

Existing work [120]−[122] on the soft error problem usually assumes certain operation conditions or target specific architectures. In this paper, instead of simulating soft errors directly, we evaluate the error-control ratio R as defined in (3.2), which is a system-level measure for evaluating error tolerance. Table 3.1 shows the results obtained from the 23 benchmarks. As discussed earlier, we would like to maximize R but a full coverage is not necessary. This is because many sublines may contain irrelevant data and thus do not need to be protected. The average error-control ratio is observed at 74.8%. In some benchmarks, e.g., `art`, `mcf`, and `swim`, nearly all the memory accesses are protected by the proposed technique.

Table 3.1: Error-control Coverage Ratio.

Workloads	**Coverage**	**Workloads**	**Coverage**
1.ammp	92.9 %	13.lucas	91.6 %
2.applu	94.4 %	14.mcf	100.0%
3.apsi	97.9 %	15.mesa	51.6 %
4.art	100.0%	16.mgrid	85.9 %
5.crafty	43.9 %	17.parser	75.4 %
6.eon	67.9 %	18.perlbmk	32.9 %
7.equake	65.2 %	19.sixtrack	33.4 %
8.fma3d	91.6 %	20.swim	99.9 %
9.galgel	98.6 %	21.twolf	70.2 %
10.gap	32.4 %	22.vortex	85.7 %
11.gcc	96.1 %	23.wupwise	49.6 %
12.gzip	64.3 %	**Average**	74.8 %

The proposed soft-redundancy allocation is self-adaptive to the variations in memory spatial locality, thereby enabling error protection only when necessary while releasing the unused memory resources for other critical tasks. This feature is demonstrated by the error detection capability measured in terms of the probability of undetected errors in the valid data. Tables 3.2 and 3.3 show the results under different soft-error rates in comparison with parity checking and Hamming code. In [119], the rates of single-bit upsets (SBU) and multiple-bit upsets (MBU) were found to be different by one to three orders of magnitude. Thus, we apply two orders of magnitude of difference between the SBU and MBU rates in these simulations. Clearly, our technique achieves $10X$ improvement in error detection over the conventional techniques without introducing large hardware overhead.

Table 3.2: Probability of Undetected SBU (PUS).

SER	**PUS parity checking**	**PUS proposed technique**
10^{-4}	1.20×10^{-6}	1.60×10^{-7}
10^{-5}	1.20×10^{-8}	1.60×10^{-9}
10^{-6}	1.20×10^{-10}	1.60×10^{-11}
10^{-7}	1.20×10^{-12}	1.60×10^{-13}
10^{-8}	1.20×10^{-14}	1.60×10^{-15}

Table 3.3: Probability of Undetected MBU (PUM).

SER	**PUM Hamming code**	**PUM proposed technique**
10^{-4}	1.20×10^{-10}	1.60×10^{-11}
10^{-5}	1.20×10^{-12}	1.60×10^{-13}
10^{-6}	1.20×10^{-14}	1.60×10^{-15}
10^{-7}	1.20×10^{-16}	1.60×10^{-17}
10^{-8}	1.20×10^{-18}	1.60×10^{-19}

Access Efficiency

Figure 3.7 shows the access efficiency as compared to the conventional cache and sub-blocked cache. By fetching only relevant sublines during memory accesses, our technique reduces the bandwidth usage by 41.3% over the conventional cache, which is comparable to the sub-blocked cache. Furthermore, the proposed technique achieves 59.5% reduction in miss rate on average over the sub-blocked cache. This is expected as our technique leverages soft redundancy for improving memory access efficiency. Thus, the proposed technique approaches the sub-blocked cache in bandwidth usage while further reduces the miss rate. These results demonstrate the advantages of soft-redundancy allocated memory for jointly improving error tolerance and access efficiency. Note that we have to admit some limitations in the proposed technique, as shown in the `perlbmk` benchmark, where the bandwidth is slightly increased as compared to the conventional cache. This is because the overhead induced by misjudgement on soft redundancy offsets the bandwidth reduction. A solution to this problem is to increase the weight factor for bandwidth during parameter selection or limit the reconfigurable confidence level at a cost of reduction in error-control coverage.

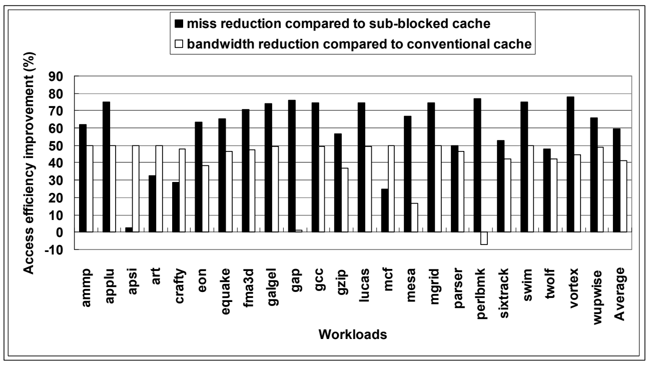


Figure 3.7: Improvement in access efficiency over the conventional cache and sub-blocked cache.

Scalability

To investigate the scalability of the proposed technique, we consider a range of memory associativities and block sizes.

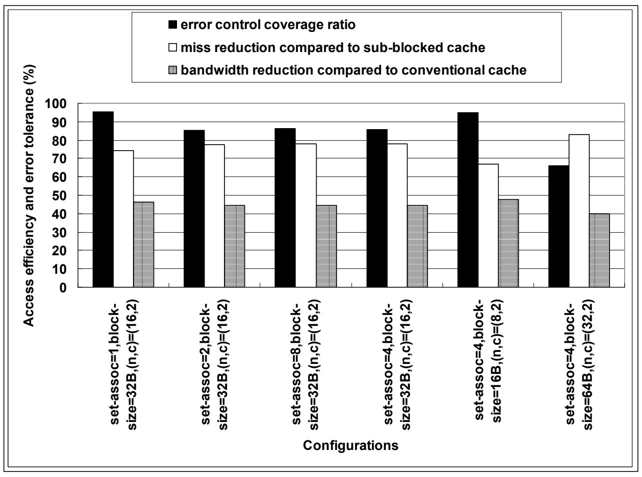


Figure 3.8: Performance of the proposed soft-redundancy allocated memory under different memory configurations (`vortex`).

In these simulations, we first choose the cache line size equal to $32B$ and vary the set-associativity among 1, 2, 4 and 8. Then, we fix the set-associativity at 4 and change the cache line size between $16B$ and $64B$. The optimal parameters are determined by using optimization method discussed in section 3.2.3. Figure 3.8 shows the results of error-control coverage ratio, miss rate reduction, and bandwidth usage reduction for the `vortex` benchmark under different memory configurations. The same trends were observed in other benchmarks as well. As indicated, our technique is robust to different memory configurations and thus supports good scalability for memory design.

3.3 Soft Redundancy at Cache Line Level

3.3.1 Introduction

Soft redundancy can be exploited at different level. In this section, we study a new approach referred to as the *soft indexing* that exploits the soft redundancy at cache line level. The idea is based on the observation that memory accesses are non-uniform across different cache lines, which results in many idle memory spaces (cache lines) and high conflict misses. The idle memory spaces create transient memory redundancy that can be exploited for both performance improvement and error control. The proposed soft indexing technique allocates memory resources in a self-adaptive manner in accordance with runtime memory behaviors, thereby achieving efficient memory access and effective error protection jointly. The benefits of the proposed technique are demonstrated by the SPEC CPU2000 benchmarks [10]. Simulation results show 94.9% average error-control coverage on the 23 benchmarks, with average of 23.2% reduction in memory miss rate as compared to the existing techniques.

In section 3.3.2, we develop the soft indexing memory microarchitecture for joint error protection and performance improvement. In section 3.3.3, we present a statistical analysis on error tolerance and provide the comparison to the existing techniques. In section 3.3.4, we evaluate the performance of the proposed technique.

3.3.2 Soft Indexing Memory Microarchitecture

The size of cache memory has a direct impact on the overall processor performance. In general, increasing cache size can reduce the miss rate by providing more memory resources for the workloads. However, runtime program statistics reveal some non-uniform memory access patterns, where many cache lines are accessed less frequently and could even remain idle or unused over time. This creates transient redundancy that can be exploited to jointly improve memory access performance and error tolerance. Since the distribution of idle cache lines varies during runtime, we need an adaptive (soft) indexing mechanism for dynamic allocation of memory resources.

In this section, we present the soft indexing memory microarchitecture that exploits the transient redundancy generated by non-uniform memory accesses for joint performance improvement and error tolerance.

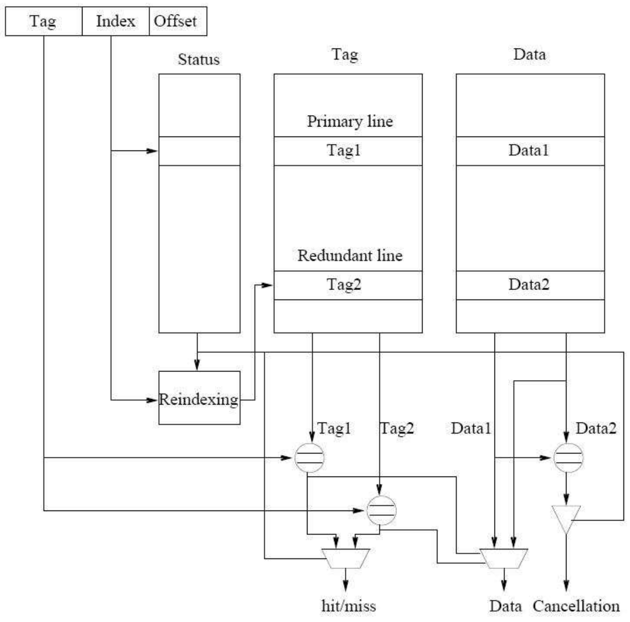


Figure 3.9: Microarchitecture of soft indexing memory.

Soft Indexing

Fig. 3.9 shows the proposed memory microarchitecture. A re-indexing function is introduced to assign an idle cache line to the currently accessed cache line for different purposes that will be explained later. Each cache line is extended with some status bits that keep track of the history of access patterns. These status bits are stored in a status table for the control of memory allocation. As shown in the following discussion, memory resources are dynamically allocated according to the availability of transient redundancy and the statistics of memory access patterns.

An example of memory access sequence using soft indexing is shown in Fig. 3.10. Each cache line can be operated in one of the three allocation modes: *idle*, *no-checking*, and *error-checking*, which represent different memory access patterns. The

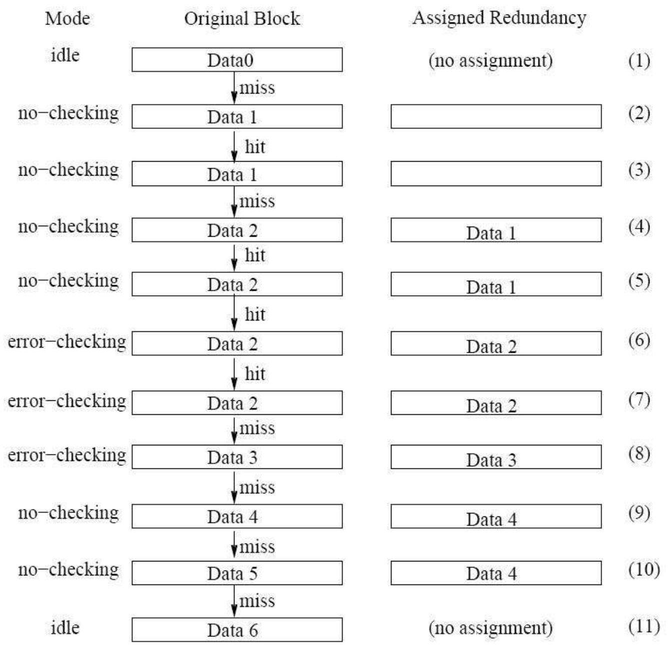


Figure 3.10: An example of soft indexing.

idle mode indicates that a cache line is currently idle or unused. Thus, this cache line can be used for transient redundancy. The *no-checking* mode represents the situation where a cache line is accessed frequently but the access pattern is less predictable. The program lacks the confidence in the outcome (hit or miss) of subsequent accesses to this cache line. Since the performance of this cache line is unstable, we can assign a redundant cache line (i.e., one of those in the *idle* mode) to this cache line to improve the access performance. On the other hand, the *error-checking* mode indicates the confidence in hit occurrence for a frequently accessed cache line. Since this cache line is experiencing access hit in a stable pattern, we can assign a redundant cache line to this cache line to improve the error tolerance.

The mode switching is based on the access history stored in the status table. The detailed operation of memory access is explained below in reference to the steps in Fig. 3.10.

Initially, all cache lines are set to the *idle* mode by default, and the re-indexing function is disabled. In the subsequent operations, assume that a miss occurs in a cache line. The status of this cache line is thus changed from the *idle* mode to the *no-checking* mode (step 2 in Fig. 3.10). Meanwhile, a redundant cache line will be assigned to this cache line by re-indexing the requested address. Ideally, the redundant cache line should be the one currently in the idle or unused status. In the proposed technique, we locate the redundant cache line using a re-indexing function as discussed later.

A single, non-consecutive hit in the cache line that is in the *no-checking* mode will not cause any data replacement (steps 3 and 5), whereas a single, non-consecutive miss will result in a replacement of both the primary and the redundant cache lines (step 4). The current data in the primary cache line will be transferred to the redundant cache line, and the new data will be filled into the primary cache line. Note that in the *no-checking* mode, the redundant cache line keeps the previously accessed data (data 1), which in general is different from the new data (data 2) in the primary cache line. This improves memory access performance by saving the previously accessed data nearby for possible future use.

Two consecutive hits in the *no-checking* mode establish the confidence in memory hit occurrence in this cache line. Thus, this cache line is switched to the *error-checking* mode (step 6). The redundant cache line is thereafter updated as a redundant copy of the primary cache line. Note that the switch condition (e.g., two consecutive hits or misses in this example) can be configured for different requirements on memory performance and error control. In the *error-checking* mode, a hit will get both copies in the primary cache line and the redundant cache line. The two copies are then compared to detect any possible data errors. If an error is detected, the hit is canceled and a miss is generated instead (step 7, also see Fig. 3.9). On the other hand, a single miss will replace both cache lines with the new data (step 8).

Two consecutive misses in the *error-checking* mode will change the status of the primary cache line to the *no-checking* mode (step 9) due to the loss of confidence in hit occurrence. Furthermore, two consecutive misses in the *no-checking* mode will send the cache line back to the *idle* mode (step 11). When the status returns to *idle* mode, this cache line is no longer associated with any redundant cache line. Subsequently, this cache line can either return to the *no-checking* mode if an access occurs, or remain in the *idle* mode if no access occurs. For the latter case, this cache line can be used as a transient redundancy for other cache lines.

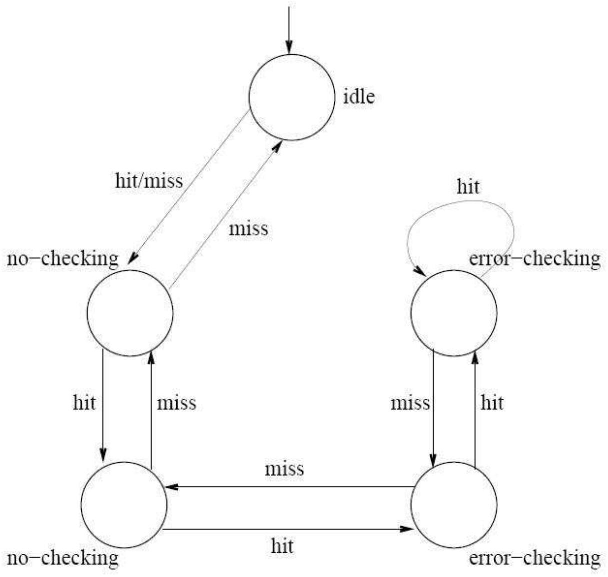


Figure 3.11: Allocation mode switch control.

In the proposed soft indexing microarchitecture, the allocation mode switches when enough confidence in hit or miss occurrence has been established. In the above example, this confidence is measured by two consecutive hits or misses. The switch control diagram is shown in Fig. 3.11. In total, five states are needed to control the mode switch, which requires only three status bits each cache line. This incurs very small hardware overheads.

Ideally, we would like to find the idle or unused cache lines for redundancy. However, this is a really difficult task due to the complexity of memory runtime behaviors. Here, we exploit memory spatial locality to locate redundant cache lines. Specifically, we utilize a XOR-based re-indexing function, where the primary cache index are masked by an XOR code to generate the address of the redundant cache line that is guaranteed to be far away from the primary cache line. Due to memory spatial locality, it is unlikely that the program will access these two memory locations simultaneously. Simulation results in section 3.3.4 demonstrate this scheme is quite effective.

Joint Performance Improvement and Error Tolerance

The proposed soft indexing technique exploits runtime memory behaviors to improve access performance and error tolerance. For cache lines in the *no-checking* mode, the program does not have enough confidence in hit or miss occurrence. This implies that the access performance is unstable for these cache lines. Instead of trashing the previously accessed data during a miss, we save these data in the redundant cache lines for possible future use (see steps 4, 5 and 10 in Fig. 3.10). In response to the subsequent memory accesses, if the redundant cache lines have the requested data, the data can be retrieved instead of fetching from the lower level memory. Indeed, by holding the previously accessed data in the redundant cache lines, conflict misses are reduced due to an equivalent increase in set associativity.

For cache lines in the *error-checking* mode, the program provides enough confidence in hit occurrence. These cache lines are hit frequently and presumably contain important data. Therefore, the redundant cache lines are assigned to these cache lines for error control (see steps 6−8 in Fig. 3.10). During data read-out, there is an additional comparison between the two data copies in the cache line pair. This comparison can be performed in parallel with the tag address comparison. Mismatches between the two copies indicate errors and hence call for cancelation of the data read-out. A memory miss is generated as a result and the new data will be fetched from the lower level memory. The redundant cache lines are generated dynamically and provide an effective means for error tolerance.

Design Considerations

The overheads of the proposed microarchitecture are manageable. Our technique does not require extra ports for the cache. Actually, only a small write buffer is needed. Other hardware overheads include a few bits per cache line for the status table and the XOR re-indexing function. Also, the latency of the additional data comparison is masked by the tag address comparison, thereby not involving any timing penalty on the critical paths.

The operations in the *idle* mode remain the same as those in traditional cache.

On the other hand, if the accessed cache line is in the *no-checking* mode, the original tag and data in the primary cache line are forwarded to the write buffer, while the tags from both the primary cache line and the redundant cache line are compared at the same time. If either cache line contains the requested address, a hit is generated and the requested data is delivered to the execution unit (for a read) or written to the primary cache line (for a write). Otherwise, a miss is generated and the requested data are fetched from the lower level memory (for a read) or written to the primary cache line (for a write). Thereafter, the previous data in the primary cache line stored in the write buffer will be written to the redundant cache line. This write operation is performed off-line and thus does not affect other memory operations that might be timing-critical.

Similarly, if the accessed cache line is in the *error-checking* mode, an additional data comparison between the cache line pair is needed during a read access for error detection. As mentioned before, this comparison can be conducted in parallel with tag comparison, thereby not affecting memory timing. For a write access, the write buffer is able to hide the write latency by scheduling the access to the redundant cache line at a later time.

3.3.3 Statistical Analysis of Error Tolerance

In this section, we perform a statistical analysis to quantify the error tolerance achieved by the proposed soft indexing technique.

In traditional memory systems, soft errors are typically modeled as single-bit upsets (SBU). As the feature size of semiconductor process being scaled into the nanometer domain, a single partial strike may potentially corrupt multiple memory bits, resulting in multiple-bit upsets (MBU). In addition, timing noise tends to cause MBU as well. Among the existing solutions, parity checking code is considered as the most effective for detecting SBU, whereas Hamming code provides error detection for up to two bits of errors. Error-control codes for more than two bits of errors are quite complicated and thus are seldom used in memory systems.

Consider a cache line with n bits each entry, i.e., each access can obtain n bits in total. A single soft error corruption may lead to either a SBU or a MBU. In [119], the rates of SBU and MBU are different from one order to three orders of magnitude based on the operating conditions such as supply voltages. According to this observation, we use two orders of magnitude of difference between the SBU and MBU rates, i.e.,

$$P_s = \beta, \tag{3.8}$$
$$P_m = \beta \cdot 10^{-2}, \tag{3.9}$$

where β is the soft error rate (SER). The soft errors are assumed to be independently and identically distributed (i.i.d.) events.

For SBU dominant cases, we denote $P_{ue\_s\_sr}$ and $P_{ue\_s\_par}$ as the probabilities of undetected errors (PUE) in the proposed technique and that in the parity checking code, respectively. We can derive

$$P_{ue\_s\_sr} = \sum_{i=1}^{n} C_n^i P_{s\_r}^i (1 - P_{s\_r})^{n-i}, \tag{3.10}$$
$$P_{ue\_s\_par} = \sum_{i=1}^{n/2} C_n^{2i} P_s^{2i} (1 - P_s)^{n-2i}, \tag{3.11}$$

where $C_n^i = \frac{n!}{(n-i)!i!}$, and $P_{s\_r} = P_s^2$ is the probability of a single-bit error that cannot be detected by the proposed technique. This occurs rarely only when the same bits of the original data and the redundant copy are both corrupted. On the other hand, the undetectable errors in parity checking schemes occur when the number of corrupted bits are even. Numerical results from (3.10) and (3.11) demonstrate $10X$ improvement in error detection capability over the parity checking code, as shown in Table 3.4 where the number of bits is $n = 16$, i.e., each memory access fetches a 16-bit word.

The proposed technique is also able to detect multiple errors occurred in any bits. This is a significant improvement over the existing error-control techniques such as parity checking code and single-error-correction double-error-detection Hamming code. The probability of undetected errors (PUE) in the MBU cases can be derived

Table 3.4: Probability of Undetected SBU (PUS).

SER	**PUS parity checking**	**PUS proposed technique**
10^{-4}	1.20×10^{-6}	1.60×10^{-7}
10^{-5}	1.20×10^{-8}	1.60×10^{-9}
10^{-6}	1.20×10^{-10}	1.60×10^{-11}
10^{-7}	1.20×10^{-12}	1.60×10^{-13}
10^{-8}	1.20×10^{-14}	1.60×10^{-15}

as

$$P_{ue\_m\_sr} = \sum_{i=1}^{n} C_n^i P_{m\_r}^i (1 - P_{m\_r})^{n-i}, \tag{3.12}$$

$$P_{ue\_m\_ham} = \sum_{i=2}^{n} C_n^i P_m^i (1 - P_m)^{n-i}, \tag{3.13}$$

where $P_{m\_r} = P_m^2$ denotes the probability of a double-bit error (the dominant MBU) that cannot be detected by the proposed technique. Similar to the SBU cases, this happens rarely only when the same two bits are corrupted in both the original data and the redundant copy. On the other hand, the undetectable errors in Hamming code occur when more than two memory bits are corrupted. Again, numerical results from (3.12) and (3.13) demonstrate $10X$ improvement in error detection capability over the Hamming code, as shown in Table 3.5 where the number of bits is $n = 16$.

Table 3.5: Probability of Undetected MBU (PUM).

SER	**PUM Hamming code**	**PUM proposed technique**
10^{-4}	1.20×10^{-10}	1.60×10^{-11}
10^{-5}	1.20×10^{-12}	1.60×10^{-13}
10^{-6}	1.20×10^{-14}	1.60×10^{-15}
10^{-7}	1.20×10^{-16}	1.60×10^{-17}
10^{-8}	1.20×10^{-18}	1.60×10^{-19}

Unlike parity checking and Hamming code that provide static error-control coverage to all the cache lines, the proposed technique relies upon a dynamic mapping strategy that enables error protection only when necessary while releasing the unused memory resources for other critical tasks such as improving access performance. In fact, this approach leads to a joint optimization for efficient memory access and effective error tolerance. Since memory operation mode is dynamically switching in accordance with runtime memory requirements, the cache lines are not always under the error protection. Specifically, when a cache line is in the *no-checking* mode, the data is not protected as this cache line undergoes an unstable access pattern. The error-control coverage ratio, denoted as R_p, can be calculated by

$$R_p = \frac{MA_{error-checking}}{MA_{total}}, \tag{3.14}$$

where $MA_{error-checking}$ and MA_{total} are the number of memory accesses when the cache line is in the *error-checking* mode and the total number of memory accesses, respectively. The error-control coverage ratio R_p reflects the effectiveness of the proposed technique in dealing with soft errors. Obviously, the switching frequency of memory allocation modes affects the error-control coverage ratio.

3.3.4 Simulation Results

Table 3.6: Configuration of Simulation Environment.

Parameter	**Value**
Cache Size	32KB
Line Size	32B
Associativity	Direct Mapped
States number of initial mode	1
States number of no-checking mode	2
States number of error-checking mode	2

In this section, we study the performance of memory access and error tolerance

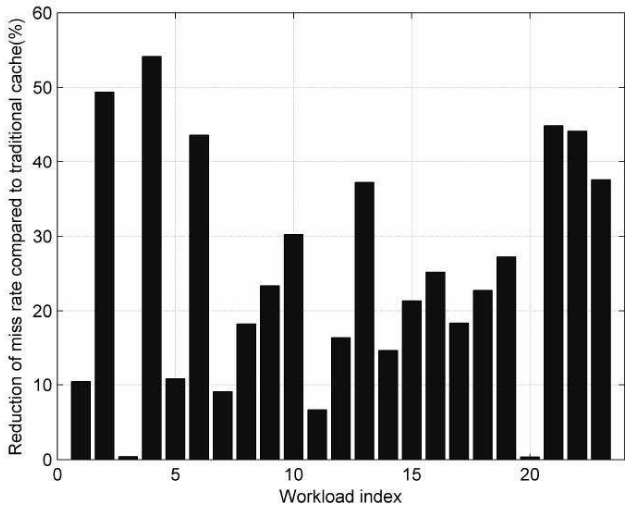


Figure 3.12: Reduction of miss rate compared to conventional direct mapped cache.

achieved by the proposed technique.

Our simulation results were obtained from a trace-driven simulator based on Dinero IV [113], which is a uniprocessor cache simulator for memory reference. The cache model in this simulator is modified to support the proposed soft indexing microarchitecture. Table 3.6 shows the configuration of the simulation environment. All the simulations were running on the SPEC CPU2000 [10] trace files collected from the Stream-Based Trace Compression (SBC) [114], where traces of 23 benchmarks are available. In these simulations, we use direct mapped cache for the purpose of demonstration. We expect to extend the proposed memory microarchitecture to set-associative cache in our future work.

As shown in Fig. 3.12, our technique achieves an average of 23.2% reduction in miss rate for the 23 benchmarks as compared to the conventional cache design. These results demonstrate that our technique is very effective in improving memory access performance. Note that the XOR code for the re-indexing function is a pre-determined value for all the 23 benchmarks. This XOR code introduces three-bit inversion from the index of the primary cache line to that of the redundant cache line. Future work needs to exploit dynamic code generation for re-indexing function to further improve the adaptability.

Table 3.7: Error-Control Coverage Ratio.

Workloads	**Coverage**	**Workloads**	**Coverage**
1.ammp	97.6 %	13.lucas	99.8 %
2.applu	99.8 %	14.mcf	26.6 %
3.apsi	91.9 %	15.mesa	99.9 %
4.art	84.2 %	16.mgrid	93.4 %
5.crafty	97.2 %	17.parser	98.7 %
6.eon	99.9 %	18.perlbmk	99.7 %
7.equake	99.9 %	19.sixtrack	99.7 %
8.fma3d	99.8 %	20.swim	99.8 %
9.galgel	99.6 %	21.twolf	98.9 %
10.gap	99.9 %	22.vortex	99.8 %
11.gcc	96.1 %	23.wupwise	99.7 %
12.gzip	97.7 %	**Average**	94.9 %

Evaluating error tolerance using architecture simulators requires realistic error models and error injection mechanisms. Many existing works [120]− [122] on soft errors usually assume certain conditions or target specific architectures. Instead of simulating errors directly, we evaluate the error-control coverage ratio as defined in (3.14). A higher error protection coverage ratio along with the improved error detection capability implies better tolerance to memory errors. Table 3.7 shows the results of error-control coverage ratio of the 23 workloads. These results are obtained from (3.14) using statistical results reported by the simulator. The average error-control coverage ratio of all the 23 benchmarks is measured at 94.9%. These results along with the theoretical analysis on error detection capability in section 3.3.3 demonstrate the significant advantage of the proposed technique in error tolerance. Moreover, the proposed technique induces very small design overheads as described in section 3.3.2. Future work to improve the proposed error-control technique could be a combination of the soft indexing and error checking codes, thereby providing error checking to cover all the data and meanwhile improve error detection.

3.4 Soft Redundancy at Thread Level

3.4.1 Introduction

Semiconductor process scaling leads to explosive silicon capacity for higher chip density, faster speed, and better opportunities of reducing power dissipation. Fueled by these technological advances, research in computer architecture has been successful to boost the microprocessor performance beyond that achievable by process scaling alone. There is a clear trend towards chip multithreaded processing [11], [14] to exploit the performance benefits rendered by future nanometer billion-transistor integration. Chip multithreading imposes new challenges to reliable computing, but it also creates opportunities for design space exploration across many not well-investigated dimensions. Existing reliability-enhancing techniques in general induce large performance and hardware overheads, and are increasingly difficult to support the architecture trend towards future multithreaded computing.

Similar to soft redundancy at cache subline level and cache line level, non-overlapped access patterns that generate transient (soft) redundancy can also be observed and exploited at thread level. This approach, referred to as the *inter-thread redundancy*, is built upon the observation that cache sets in multithreaded microprocessors exhibit non-uniform access activities from different threads. This generates inter-thread transient redundancy in memory spaces concurrently accessed by multiple threads. By releasing soft-redundant spaces to store the copy of more frequently accessed data, effective error tolerance can be achieved. In contrast to conventional redundancy-based solutions, inter-thread redundancy does not introduce extra hardware redundancy. Rather, it naturally resides in the multithreaded computing process and thus can be utilized deliberately to compensate for unpredictable performance variations and low-level physical effects. Different from the past work [116] on register files, we target unique inter-thread redundancy for cache memory in multithreaded systems. In addition, the proposed error detection mechanism is able to detect and recover multiple-bit upsets (MBU). Trace driven simulations on the SPEC CPU2000 benchmarks demonstrate the proposed technique in improving error tolerance for multithreaded computing.

In section 3.4.2, we review our past work on thread-associative cache. In section 3.4.3, we propose an adaptive memory microarchitecture exploiting inter-thread redundancy for error tolerance. Simulation results are presented in section 3.4.4 for evaluation of the proposed technique.

3.4.2 Thread-Associative Memory Microarchitecture

Most of the present-day multithreaded microprocessors employ conventional memory microarchitecture developed originally for single-threaded architectures. This memory microarchitecture does not distinguish the access requests from concurrent threads, and thus is vulnerable to inter-thread interferences that inevitably hurt the overall performance.

Thread-aware memory microarchitecture effectively addresses this issue by managing memory resources in a different way to compensate for inter-thread contentions. In [26], we proposed a thread-associative cache by introducing a new concept, referred to as the *rail*, to regulate memory accesses by different threads. In a thread-associative cache, each way within a set is grouped into several rails sharing the same address. If there are m rails in a set, it is called m-rail thread-associative cache. The number of rails within a set is equal to the number of concurrent threads supported by the microprocessor. Memory access requests will be directed to multiple memory rails with the same address. The thread information will then determine which rail to access. In contrast to traditional memory design, cache replacements for concurrent threads are decoupled, thereby removing inter-thread conflicts. Furthermore, intra-thread conflicts could be managed as well by interleaving the thread associativity with set associativity.

Figure 3.13 shows an example of thread-associative memory microarchitecture. The cache is two-rail thread-associative supporting two concurrent threads, and each set is four-way set-associative. Different from conventional caches, the cache address is composed of four components: tag, index, offset, and thread ID. When the instructions from different threads are scheduled, the thread ID of each thread can be attached to the original memory request in order to determine which rail in the

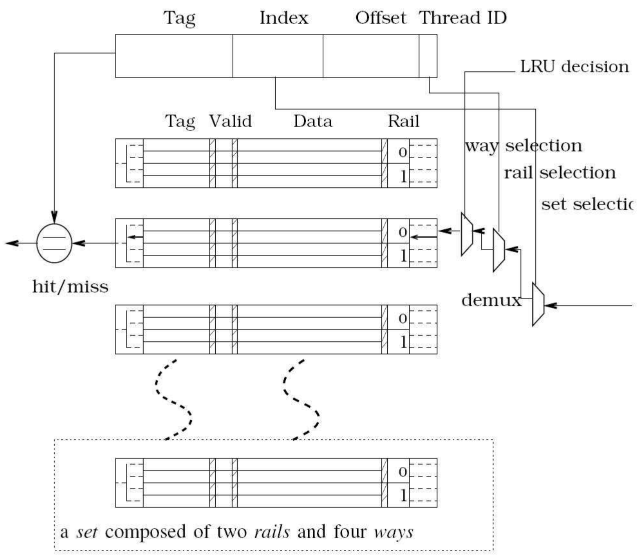


Figure 3.13: The microarchitecture of thread-associative cache.

desired set should be actually accessed. Note that the proposed microarchitecture does not increase the memory size but instead allocates the same amount of memory resources in a different way as compared to conventional cache.

3.4.3 Exploiting Inter-Thread Redundancy for Error Tolerance

Thread-associative mapping achieves significant improvement in both memory access performance and energy efficiency as compared to conventional cache [26]. Since memory accesses from concurrent threads are confined within the designated rails, non-uniform access activities from different threads create inter-thread transient (soft) redundancy that can be exploited for improving error tolerance in multithreaded microprocessors. In this section, we will introduce a microarchitecture technique for identifying and allocating inter-thread redundancy and then discuss some design issues related to the implementation.

Adaptive Microarchitecture for Inter-Thread Redundancy

The soft redundancy among simultaneously executed threads is generated dynamically by the multithreaded computing process. This is evident in the thread-associative memory, where memory rails assigned to the concurrent threads exhibit non-uniform access activities due to runtime program variations. Within a certain time period, one of the cache rails might be accessed more frequently by the corresponding thread, while the other rails are accessed less frequently and may even remain idle over time. We denote this phenomenon as *inter-thread redundancy* to distinguish it from other soft redundancy phenomena [22] due to different mechanisms in conventional single-threaded microprocessors.

To utilize the inter-thread redundancy for error tolerance, we develop an adaptive microarchitecture technique to identify and allocate the soft-redundant memory resources across the thread boundary. In general, each thread is expected to exhibit certain locality characteristics, but these characteristics are most likely different for the concurrent threads. By keeping track of the access history of thread-related rails within a set, we can predict inter-thread access patterns and allocate soft-redundant memory resources accordingly for error-control purpose.

In the proposed microarchitecture, each cache set incorporates counters to monitor the hit/miss in the rails. The counters are employed to detect whether a rail is experiencing hits more frequently than the other rails. If the hit count exceeds a certain level, we can make the following observations to the corresponding rail: (1) the data in this rail have been unreplaced for a relatively long time, thus error protection is needed (necessary to protect); (2) the data in this rail are very likely to be accessed again in the near future, i.e., the data are presumably important (worthy to protect); and (3) other rails accessed less frequently provide inter-thread redundancy because of their low activities (feasible to protect). Based on these observations, an arbitrator will then select one way (according to certain policy, e.g., LRU) in the most hit-intensive (MHI) rail and store a redundant copy of it to a least hit-intensive (LHI) rail. When one rail has a hit, its counter will be increased by one while all the other counters are decreased by one. This allows the counters to keep the relative hit counts. If any rail has a miss, it indicates that either the data in the MHI rail would be replaced in near future by another miss, or the data in the LHI rail should be maintained for a while in case the program calls the data again. Thus, all the counters will be decreased by one, preventing from allocating inter-thread redundancy prematurely. Updating the counters in each set according to the relative hit/miss results will automatically determine the MHI and LHI rails. Note that the MHI and LHI rails are not fixed but dynamically changing during the course of multithreaded execution. Once the counter value of the MHI rail exceeds a threshold, the arbitrator will make a copy of one way from the MHI rail and store it to another way in the LHI rail. Note that more aggressive strategies such as copying the whole MHI rail can also be enforced according to the error-control requirements and access behaviors of the concurrent threads. After inter-thread redundancy allocation, the counters of those involved rails are reset to the initial value for future allocation.

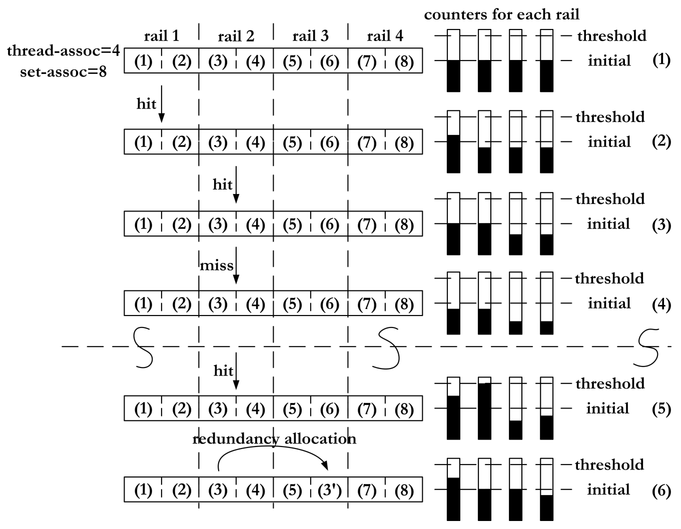


Figure 3.14: An example of inter-thread redundancy allocation.

Figure 3.14 gives an example of this microarchitecture implementation. Here, the cache is four-rail thread-associative (supporting four concurrent threads) and eight-way set-associative for the purpose of demonstration. All the four counters are set to an initial value (shown in stage 1). If rail 1 (assigned to thread 1) has a hit, the counter 1 will be increased by one, while the other three counters are decreased by one (stage 2). Likewise, a hit in rail 2 increases counter 2 and decreases other counters (stage 3), but a miss in rail 2 results in decrease of all the four counters (stage 4). As this process continues, counter 2 reaches the threshold value after a certain period (stage 5), implying that thread 2 is the most hit-intensive. The arbitrator detects this event and initiates the inter-thread redundancy allocation within the set. Since at this time rail 3 is the LHI rail, the arbitrator decides to copy data from one way in rail 2 to one way in rail 3. The selection of these two ways can be based on the LRU replacement policy, i.e., the most recently accessed way in rail 2 and the least recently accessed way in rail 3. After this operation, rail 3 will keep a copy of the data originally in rail 2 for error tolerance, and counters 2 and 3 will be reset to the initial values (stage 6).

Implementation

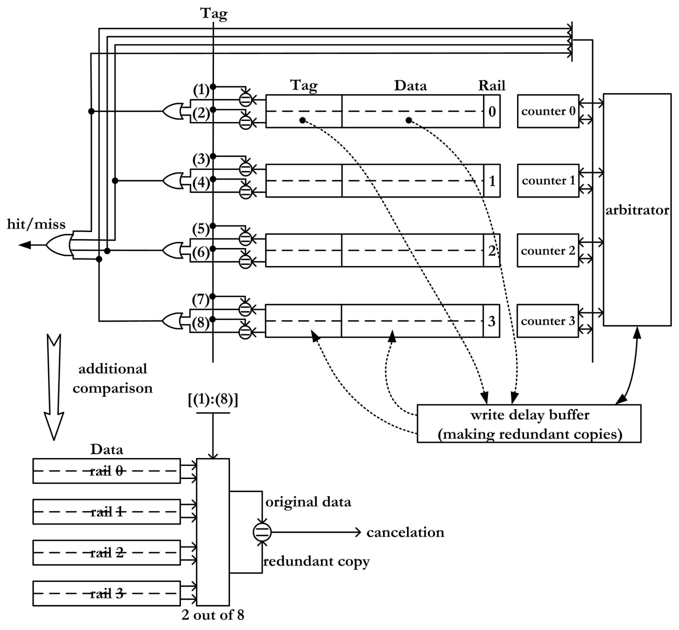


Figure 3.15: Implementation of the proposed technique.

Figure 3.15 shows the block diagram of the proposed memory microarchitecture for a cache set with four-rail thread-associative and eight-way set-associative. Hit/miss information is gathered by the four counters for access updating. Based on the values of these counters, the arbitrator will determine the status of inter-thread redundancy and assign soft-redundant memory resources to the MHI rails for error tolerance. A dedicated memory buffer, namely write delay buffer as shown in Fig. 3.15, is notified to store the tags and data of the selected ways in the MHI rails. A delayed write is performed to write the tags and data to the assigned soft-redundant space. These write operations are offloaded from the critical timing path, i.e., they will be performed when there is no other pending write operations.

During read accesses, redundant data copies are detected if the desired tag address matches tags of multiple cache lines. An additional comparison is performed between the original data and its redundant copy. If soft errors are detected, the corrupted read-out data will be canceled. Since data can be read out in parallel with this additional comparison, there is no performance degradation on timing-critical read operations.

The hardware overheads of this technique include: (1) additional storage for the rail indication bits, as shown in Fig. 3.13, which are $\log_2 m$ bits per cache block for an m-rail cache, (2) an additional comparator shared by all the data accesses for bit-wise comparison to detect possible errors, which needs k XOR-gates if the width of each access is k bits (usually smaller than the cache block size), and (3) some other components, such as m counters per set for an m-rail cache (each of which only needs a small number of bits to predict inter-thread redundancy) and the arbitrators (including pointers and control logic for determining the MHI and LHI rails, and necessary circuitry for directing the write delay buffer to make redundant copies). Note that by bit-wise comparison between two data copies, a variety of error patterns can be detected. In contrast, commonly used error correcting codes (ECC) can only detect certain error patterns (e.g., single error correction and double error detection). To provide equivalent capability of detecting multiple-bit upsets (MBU), very complicated ECC circuits such as finite field multipliers need to be implemented. This would lead to more hardware overheads than the proposed technique. Furthermore, the proposed technique does not aim at providing static error protection as conventional ECC, as explained in the next section.

3.4.4 Simulation Results

In this section, we evaluate the proposed error-tolerant memory microarchitecture. Specifically, we determine the error-control coverage ratio, defined as the ratio of protected accesses over all accesses. This measure reflects the effectiveness of the proposed technique in dealing with soft errors. A higher error-control coverage ratio indicates better tolerance to soft errors.

Our simulation results were obtained from a trace-driven simulator based on the Dinero IV [113] for memory reference. The cache model is modified to support the proposed error-tolerant and thread-associative cache microarchitecture. In these simulations we only evaluate the L1 data cache, but the trends observed are likely to repeat in other memory systems. The total memory size and block size of the cache are set to 32KB and 64B, respectively. The L1 data cache is configured as two-rail thread-associative and four-way set-associative cache in a dual-threaded processor, and four-rail thread-associative and eight-way set-associative in a four-threaded processor. The initial and threshold values of allocation counters are 0 and 5, respectively. All the simulations were running on the SPEC CPU2000 [10] trace files collected from the Stream-Based Trace Compression (SBC) [114].

Table 3.8: Error-control coverage ratio in dual-threaded simulation.

Index	**Workloads**	**Error-Control Coverage**
1	ammp & crafty	88.9%
2	apsi & twolf	90.3%
3	eon & gzip	95.3%
4	equake & gap	96.7%
5	fma3d & swim	81.9%
6	galgel & gcc	85.8%
7	lucas & mgrid	76.0%
8	mesa & parser	93.7%
9	perlbmk & sixtrack	99.1%
10	vortex & wupwise	94.5%
	Average	90.2%

Table 3.9: Error-control coverage ratio in four-threaded simulation.

Index	**Workloads**	**Coverage**
1	eon & gzip & equake & gap	87.8%
2	eon & gzip & mesa & parser	86.7%
3	eon & equake & mesa & perlbmk	89.5%
4	equake & gap & perlbmk & sixtrack	88.5%
5	mesa & parser & perlbmk & sixtrack	89.2%
6	apsi & twolf & eon & gzip	84.7%
7	perlbmk & sixtrack & vortex & wupwise	89.3%
8	gap & gzip & parser & sixtrack	85.3%
9	equake & parser & perlbmk & wupwise	90.1%
10	gzip & mesa & sixtrack & vortex	85.1%
	Average	87.6%

Tables 3.8 and 3.9 show the results of error-control coverage ratio from dual-threaded and four-threaded simulations, respectively. Combinations of workloads are randomly selected from the available benchmarks. The average error-control coverage ratio is measured at 90.2% in the dual-threaded simulation and 87.6% in the four-threaded simulation. Interestingly, our technique does not provide 100% coverage to all the memory data. As a matter of fact, this exactly reflects the unique feature of the proposed technique. Traditional solutions (e.g., error correcting coding) provide a full but static coverage where many irrelative or to be trashed data are also under the protection. In contrast, our technique exploits inter-thread redundancy for error tolerance only when necessary (e.g., stable hits indicating that the corresponding data are important and necessary to protect), thereby allowing a balanced approach for effectiveness and efficiency of error tolerance.

Chapter 4

Hybrid Redundancy for Nanomemory

4.1 Introduction

The design of integrated circuits has witnessed a dramatic improvement in integration density and performance due to process scaling as well as advances in design methodologies. However, conventional CMOS is approaching the end of roadmap, making Moore's law difficult to continue. This compels researchers to investigate nanoelectronic devices including carbon nanotubes [2], silicon nanowires [3], quantum-dot cellular automata [4, 98], resonant tunneling devices [5], and single electron transistors [99]. Many nanoelectronic-based systems have been proposed in some work such as programmable logic arrays [100], application-specific integrated circuits [101], and memories [102, 103, 104]. Although these systems hold the promise to orders of magnitude improvement in performance, the underlying nanosubstrates also introduce some new challenges that may have a profound impact on system design and optimization. One of the fundamental obstacles for robust computing at nanoscale is the likely wide range of variations and uncertainties in a bottom-up fabrication method such as self-assembly. It is expected that defect/fault rates in nanoelectronic devices will reach several orders of magnitude higher than conventional CMOS. Memory systems, which are the primary application targeted by the emerging nanotechnologies, are particularly exposed to this problem due to the ultra-high integration density and elevated error sensitivity.

To deal with defects/faults in nanoscale systems, most existing techniques rely on certain types of redundancy at various levels of design hierarchy. At the device/circuit level, defect mapping techniques [105, 6] are proposed to identify defective devices and utilize spare defect-free devices. A similar idea is also adopted in nanoelectronic crossbar memory [103, 104], where redundant rows and columns are introduced as hardware redundancy to replace defective cells. However, these per-chip based test-then-reconfigure approaches are usually expensive and time-consuming. When defect rates are high, the amount of hardware redundancy necessary for reconfiguration may become unaffordable. As shown in [104] for example, when the bit defect rate is 5%, it is difficult to construct even a small 32×32 nanoelectronic crossbar memory unless a significant amount of hardware redundancy (i.e., $15\times$ overhead) is introduced. Furthermore, even if defects are diagnosed and mapped out prior to the normal operation, new defects and transient faults may still sneak into the system over the life time. To address this problem, error-control coding (ECC) has been employed to provide fault tolerance for nanoelectronic crossbar memory [106]. However, simple ECC schemes may not be adequate to correct all errors and faults, while complex ECC schemes in general lead to large overhead in encoding and decoding circuits, which could be exposed to similar reliability problems. At a higher system level, redundancy-based techniques such as N-modular redundancy [9, 84] can effectively detect and correct logic errors, but they also involve large integer factor overhead.

Existing techniques relying on hardware redundancy for defect/fault tolerance, while adequate in the past, would be difficult to scale at high defect/fault rates. In this chapter, we propose a new approach where defect/fault tolerance is achieved by exploiting *hybrid redundancy*, which is a combination of hardware redundancy (hereinafter referred to as hard redundancy) and transient (soft) redundancy generated by runtime utilization of spatial/temporal locality in memory accesses. Soft redundancy exists because of the mismatches between fixed memory size and varying workload activities. Certain memory locations may become idle or to-be-replaced during the course of computation, and thus can be released as soft redundancy. In chapter 3, we have demonstrated the use of soft redundancy in CMOS-based memory systems. This approach does not increase memory size (e.g., no extra memory cells for redundancy as in N-modular redundancy) but exploits the inherent transient redundancy for improvement in memory error tolerance and access efficiency. Note that soft redundancy is a common phenomenon in computing applications including those performed by nanoelectronic-based systems. However, it was shown that soft redundancy may be less predicable in certain applications. In this chapter, we exploit a seamless integration of soft redundancy and hard redundancy. The proposed technique leads to new performance-reliability tradeoffs that are critical to effective and scalable error management for nanoelectronic memory systems.

The preliminary idea of hybrid redundancy allocation was reported in [33]. Here, we extend our past work by developing a design framework to address the critical issues such as nanoelectronic memory organization and overhead management. We also conduct a statistical analysis on the error correction capability of the proposed technique. This analysis shows that hybrid redundancy allocation can achieve better error tolerance than many complicated ECC codes at a reduced cost. A comprehensive evaluation is performed with new results covering the probability of uncorrectable errors, overhead, and scalability under various system settings.

The rest of the chapter is organized as follows. In section 4.2, we review the basic structures of nanoelectronic memory systems built from nanowire crossbars. In section 4.3, we present the proposed hybrid redundancy allocation for tolerance of memory errors caused by defects and faults. Simulation results are discussed in section 4.4.

4.2 Nanoelectronic Crossbar Memory

Most nanoelectronic memory systems employ an architecture as shown in Fig. 4.1 [102]. The crossbar array consists of a number of vertical and horizontal nanowires on two parallel planes. Each crosspoint is a Pt/rotaxane/Ti junction, the resistance of which can be configured to a low resistance (between 10^6 and $5 \times 10^8 \Omega$) or a high resistance ($\geq 4 \times 10^9 \Omega$) to represent '0' and '1' states, respectively. Thus, not only logic functions but also memory cells can be realized. The column and row selection circuits address the specific memory cells (crosspoints) in the memory array. Upon a memory access, the selected memory cells are applied with bias voltages to configure the resistances on a write or to measure the currents flowing through the wires on a read.

Various defects may occur during bottom-up (self-assembly) fabrication. It is projected that defect rates in the range of $5\% - 10\%$ will pose a significant challenge for constructing complex logic functions and memory systems. Most defects in nanoelectronic crossbars can be grouped into: (1) crosspoint stuck-open, (2) crosspoint stuck-close, (3) nanowire open, and (4) nanowire short. The two short defects (types (2) and (4)) are equivalent in the way that both are detrimental to all the crosspoints located on the two shorted orthogonal wires. Hence, the involved nanowires along with the affected crosspoints should be mapped out through a reconfiguration step. Indeed, short defects can be easily detected because the resistances are as low as those of nanowires (e.g., $\leq 10^5\Omega$), which stand out clearly from valid logic states.

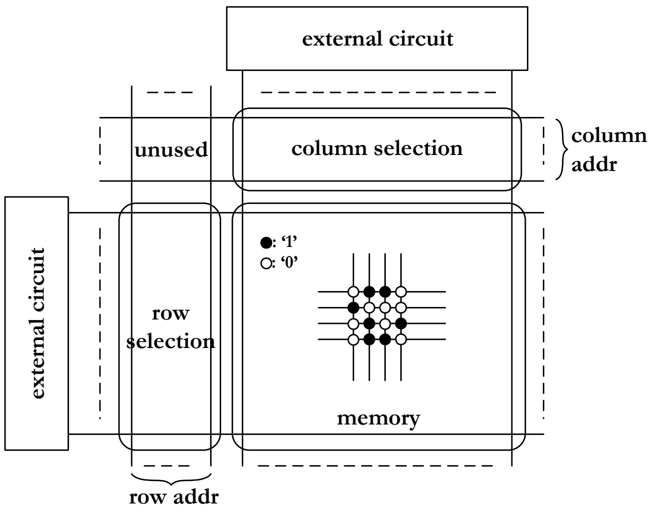


Figure 4.1: A generic organization of nanoelectronic crossbar memory.

Open defects (types (1) and (3)), on the other hand, keep the resistances high (e.g., $\geq 10^9\Omega$) at the affected crosspoints. Therefore, the memory bits are read as '1' regardless of their stored values. This leads to an interesting observation, i.e., whether an open defect can develop into bit errors is data-dependent. Specifically, when the defective memory cell stores a '1', the bit can still be read out correctly. An error occurs only when the memory cell stores a '0'. This fact has been used in [107] to reduce the complexity of ECC for nanoelectronic crossbar memory. In [33], we also employ this fact to directly correct memory bit errors using bitwise AND logic.

In addition to defects, transient faults will continue to be a problem in nanoelectronic memory, causing erroneous behaviors that are difficult to model and predict. Note that defects and faults may also occur in other components (e.g., decoders and encoders, microscale-to-nanoscale interfaces), as studied in [108, 109, 110, 111]. In the following sections, we will focus our study on crossbar arrays.

4.3 Hybrid Redundancy Allocation for Defect/Fault Tolerance

In this section, we present a nanoelectronic memory system exploiting hybrid redundancy for tolerance of memory errors caused by defects and faults. We first discuss the memory organization and then develop a general method for hybrid redundancy allocation. The effectiveness of the proposed approach and the incurred overhead are also evaluated.

4.3.1 Memory Organization

To build nanoelectronic memory, many design concepts applied to CMOS memory are still viable. For example, it is generally easy to ensure functional correctness in small memory. Hence, nanoelectronic memory systems, especially the large ones, need to be organized into banks [103]. Each bank is a small sub-memory that avoids excessively large shared rows/columns and variations. Furthermore, banks can be divided into blocks/sub-blocks to accelerate data access. Due to the spatial locality of memory access, when one memory cell is accessed, its neighboring cells are likely to be accessed as well. Operating at the granularity of blocks/sub-blocks, multiple bits can be read/written in parallel to achieve high access efficiency. In the proposed technique, we apply both bank organization and block/sub-block partitioning as shown in Fig. 4.2. We denote the total size of the nanoelectronic crossbar memory as $N \times N$ (N columns and N rows). The number of banks in this memory and the number of blocks in each bank are N_{bnk} and N_{blk}, respectively. Thus, the size of each block is $B = \frac{N^2}{N_{bnk}N_{blk}}$ bits. The memory physical address is divided into three fields: bank ID, block ID, and offset, referring to a specific bank, block and bit location, respectively. Conventional CMOS circuitry may be utilized to implement the multiplexing and decoding logic for interbank connectivity. As this chapter focuses on the reliability issues in crossbar arrays, we assume the same nano/CMOS interface as studied in [103], [111].

To deal with a large number of defects, spare cells (i.e., hardware redundancy) are introduced to most nanoelectronic memory systems. Different from existing defect/fault tolerance techniques, we exploit the hardware redundancy in combination with soft redundancy generated at runtime for new ways of managing memory errors caused by defects and faults. To exploit hybrid redundancy, we divide each user block into N_{sub} sub-blocks, each containing B/N_{sub} bits. Spare cells are grouped into N_{spare} sub-blocks (of the same sub-block size) as hardware redundancy. As shown in Fig. 4.2 for example, each block has $N_{sub} = 4$ sub-blocks (the IDs are "000", "001", "010", and "011"). In addition, $N_{spare} = 2$ sub-blocks (i.e., sub-block "100" and "101") are built-in hardware redundancy. As the size of memory block is fixed, runtime variations in memory access locality may cause some sub-blocks to be under-utilized or occupied by irrelevant data. It will be shown in the next subsection that these sub-blocks can be identified as soft redundancy by tracking memory access history at the granularity of sub-blocks.

4.3.2 Hybrid Redundancy Allocation

Soft redundancy inherent in memory access can be released to store a copy of useful data for error tolerance. However, depending on the access patterns of workloads at runtime, the availability of soft redundancy might not be stable enough to provide adequate error-control coverage. On the other hand, techniques relying solely

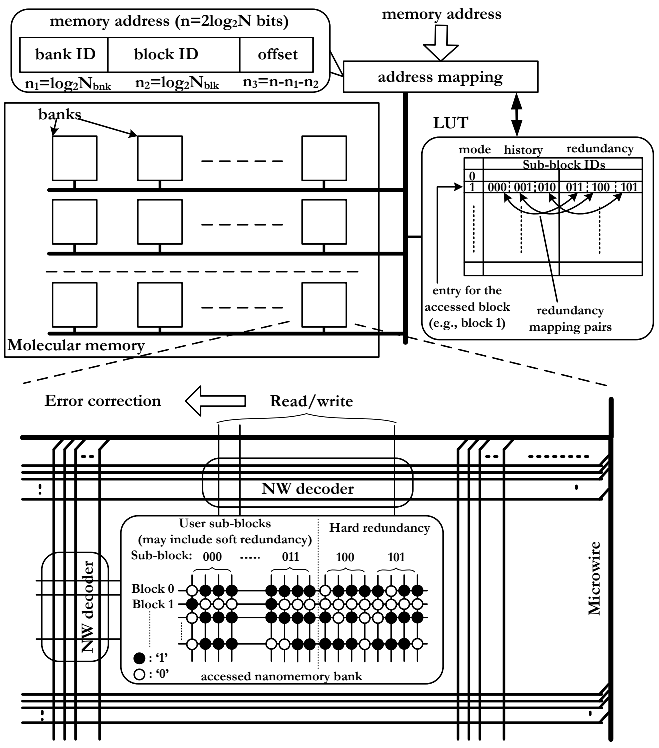


Figure 4.2: Memory architecture for hybrid redundancy allocation.

on hardware redundancy for error tolerance induce large overhead and thus are difficult to scale at high defect/fault rates. Our approach exploits hybrid redundancy to complement and enhance each other for significant improvement of effectiveness, efficiency, and scalability of error tolerance for nanoelectronic memory systems.

To facilitate hybrid redundancy allocation, a look-up table (LUT) is introduced to monitor sub-block access activities. Each entry in the LUT maintains the redundancy mapping information of one block (containing $N_{sub} + N_{spare}$ sub-blocks) in the nanoelectronic memory. As shown in Fig. 4.2, a LUT entry is composed of three fields: *mode field*, indicating the operation mode of each block; *history field*, storing the IDs of the sub-blocks that are considered useful for computation and hence need to be protected (e.g., the most recently replaced sub-blocks are useful as they are likely to be accessed again according to access locality); and *redundancy field*, storing the IDs of the redundant sub-blocks, including those considered as soft redundancy as well as built-in hardware redundancy. Note that this does not necessarily lead to a large LUT; in fact, the LUT can be organized to serve for the active blocks only. When some blocks become inactive during the course of computation, the assigned LUT entries can be released for other active blocks. As the LUT is critical to hybrid redundancy allocation, it may be implemented in conventional CMOS to improve its robustness. The involved overhead is discussed in section 4.3-D.

According to the value of the mode field, a block can operate in either hard-redundancy allocation mode (mode=0) or hybrid-redundancy allocation mode (mode=1). To assess the difference between these two modes, we define the error-control coverage ratio R as the possibility that a sub-block is protected by the proposed technique when it is accessed. During the hard-redundancy allocation mode, only hardware redundancy will be allocated to store data copies for error tolerance. Thus, the N_{spare} sub-blocks grouped by the spare crosspoints will store the redundant copies for N_{spare} of the N_{sub} user sub-blocks. The error-control coverage ratio is thus equal to $R = \frac{N_{spare}}{N_{sub}}$, which is less than 100% for small hardware overhead. During the hybrid-redundancy allocation mode, all the useful sub-blocks listed in the history field will be protected by a combination of soft redundancy and hardware redundancy, i.e., $R = 100\%$. Apparently, if the memory block can frequently stay in the hybrid-redundancy allocation mode, the average error-control coverage ratio would be closer to 100%, even when a relatively small N_{spare} is available (i.e., a small amount of hardware redundancy). The decision on which allocation mode to operate in is based on whether the access pattern is stable enough to predict soft redundancy from the access history. Hard-redundancy allocation mode is taken when the access pattern is not stable, whereas hybrid-redundancy allocation mode is activated when the LUT can sufficiently identify soft redundancy. A confidence level c is used to direct mode switching; if the LUT correctly (or incorrectly) predicts the access pattern for consecutively c times, the allocation mode will switch to the hybrid-redundancy (or hard-redundancy) allocation mode. The rationale for using this metric is that memory access history can be used as a viable indicator to predict future accesses because of the inherent temporal/spatial locality. It is shown in section 4.4 that by utilizing the proposed technique, the nanoelectronic memory can achieve a very high average error-control coverage even with a small hardware redundancy. Detailed procedure on allocation mode assignment will be discussed with an illustration example after we describe the history field and redundancy field in the LUT.

The history and redundancy fields store the IDs of useful and inactive sub-blocks, respectively, where the latter ones are released as soft redundancy. Each of the two fields holds the IDs of half of the $N_{sub}+N_{spare}$ sub-blocks. Depending on the allocation mode of the block, a sub-block may keep a redundant copy in another sub-block within the same block. These two sub-blocks maintain a *redundancy mapping pair*. During the hybrid-redundancy allocation mode, each sub-block with the ID listed in the history field is assigned a redundancy mapping pair, as specified by the order of the sub-block IDs in the history and redundancy fields. Specifically, the sub-block with the ID listed in the i^{th} location of the history field will be assigned a redundant copy on the sub-block with the ID listed in the same location of the redundancy field. During the hard-redundancy allocation mode, not all the sub-blocks listed in the history field are assigned the redundancy mapping pairs. Due to the limited hardware redundancy, only the first N_{spare} sub-blocks listed in the history field will be protected by hardware redundancy.

The redundancy mapping pairs are not predetermined or fixed but are adjusted at runtime based on data replacement in sub-blocks. In addition, sub-blocks allocated as soft redundancy can be released for functional usage if necessary; i.e., all crosspoints are accessible and there is no increase of memory size when using soft redundancy. We use an example as shown in Fig. 4.3 to illustrate the procedure of hybrid redundancy allocation. Only one memory block is shown where $N_{sub} = 4$ and $N_{spare} = 2$. The four user sub-blocks have IDs "000", "001", "010", and "011", and the hardware redundancy are indexed as sub-blocks "100" and "101". The confidence level is set to $c = 2$ for the purpose of demonstration. Initially, this memory block is in the hybrid-redundancy allocation mode by default (mode=1). Each sub-block listed in the history field has a redundancy mapping pair as indicated by the sub-block ID in the redundancy field. For example, sub-block IDs "000" and "011" are the first in the history field and the redundancy field, respectively. Therefore, they form a redundancy mapping pair and both store the same data. The content of this data is denoted as A, and A' is the redundant copy of A. Here, sub-block "011" is utilized

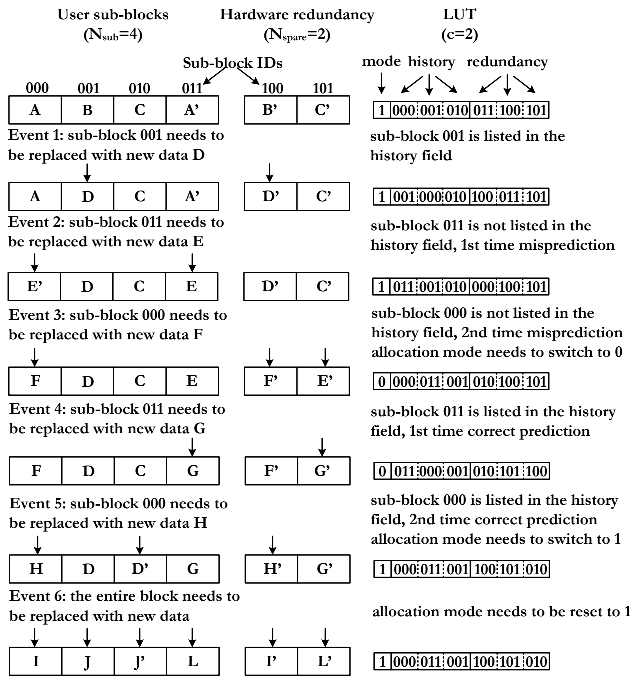

User sub-blocks (N_{sub}=4)
Hardware redundancy (N_{spare}=2)
LUT (c=2)
Sub-block IDs
mode history redundancy
000 001 010 011 100 101
A B C A' B' C'
1 000 001 010 011 100 101
Event 1: sub-block 001 needs to be replaced with new data D
sub-block 001 is listed in the history field
A D C A' D' C'
1 001 000 010 100 011 101
Event 2: sub-block 011 needs to be replaced with new data E
sub-block 011 is not listed in the history field, 1st time misprediction
E' D C E D' C'
1 011 001 010 000 100 101
Event 3: sub-block 000 needs to be replaced with new data F
sub-block 000 is not listed in the history field, 2nd time misprediction allocation mode needs to switch to 0
F D C E F' E'
0 000 011 001 010 100 101
Event 4: sub-block 011 needs to be replaced with new data G
sub-block 011 is listed in the history field, 1st time correct prediction
F D C G F' G'
0 011 000 001 010 101 100
Event 5: sub-block 000 needs to be replaced with new data H
sub-block 000 is listed in the history field, 2nd time correct prediction allocation mode needs to switch to 1
H D D' G H' G'
1 000 011 001 100 101 010
Event 6: the entire block needs to be replaced with new data
allocation mode needs to be reset to 1
I J J' L I' L'
1 000 011 001 100 101 010

Figure 4.3: An illustration example of allocating hybrid redundancy.

as soft redundancy. Likewise, sub-blocks “001” and “100”, “010” and “101”, are redundancy mapping pairs as well, where sub-blocks “100” and “101” are actually hardware redundancy.

The redundancy allocation is explained using several access events for the purpose of demonstration. In access event 1, sub-block “001” is replaced with new data (denoted as D in Fig. 4.3). As sub-block “001” is already listed in the history field, the LUT predicts this access pattern correctly. The orders of sub-block IDs in the history and redundancy fields need to be updated in response to this access. Sub-block ID “001” is moved to the first position in the history field because it is the most recently accessed sub-block. According to memory spatial/temporal locality, sub-block “001” stores the data that is most likely to be accessed again in the near future. This predication, however, could be wrong as discussed in access event 2. The ID of sub-block “100”, which is the redundancy mapping pair of sub-block “001”, is also moved to the first position in the redundancy field. The content of sub-block “100” is updated to D' to store a redundant copy of sub-block “001”. The IDs of other sub-block will either shift to the right within the corresponding field or remain at the same location. Hence, the history and redundancy fields are updated to "001-000-010" and "100-011-101", respectively. This will maintain the same redundancy mapping pairs between the two fields without reshuffling the data in the sub-blocks.

Assume that in access event 2, sub-block "011" is updated with new data E. Note that sub-block "011" is not listed in the history field. This indicates that the LUT fails to predict this access. As the confidence level is set at $c = 2$, this misprediction does not enable mode switching. The history and redundancy fields are updated to "011-001-010" and "000-100-101", respectively, to maintain the current redundancy mapping pairs. Next, access event 3 needs to replace sub-block "000" with new data F. This sub-block is again not listed in the history field; i.e., the LUT fails twice consecutively. With the confidence level $c = 2$, the access pattern is considered to become unstable. The allocation mode is switched to the hard-redundancy allocation mode (mode=0). Because it is now impossible to predict soft redundancy and maintain the redundancy mapping pairs, the history and redundancy fields need to be updated in a different way. Sub-block "000" is replaced with new data F, and sub-block "100", which is first hardware redundancy sub-block listed in the redundancy field, will store a copy of sub-block "000". Meanwhile, sub-block "101", which is the second hardware redundancy sub-block (and also the last one as $N_{spare} = 2$), will store a copy of sub-block "011".

During the next two access events 4 and 5, the LUT correctly predicts the accesses as the IDs of the accessed sub-lines are all found in the history field. This indicates that the access pattern might have returned to stable (with confidence level $c = 2$). Thus, the allocation mode is switched back to the hybrid-redundancy allocation mode (mode=1). Finally, if the next access needs to replace the entire block as shown in event 6, the allocation mode will be reset to the hybrid-redundancy allocation mode (in this example it is already in this mode and thus no operation is needed). Since the access pattern might be changed completely after the replacement of the entire block, hybrid redundancy is utilized to exploit this effect for error tolerance.

4.3.3 Analysis of Error Tolerance

The effectiveness of the proposed technique can be measured by the probability of uncorrectable errors. Denote the bit error rate as f and assume a uniform error distribution. Exploiting redundancy allocation can detect and correct errors in any bits unless the same bits in both the user sub-block and its redundancy mapping pair are corrupted (the possibility of this case is f^2). The probability of uncorrectable errors, $P_{protected}$, can be calculated for the hybrid-redundancy allocation mode as

$$P_{protected} = \sum_{i=1}^{k} \mathbf{C}_k^i (f^2)^i (1-f^2)^{k-i}, \tag{4.1}$$

where $k = \frac{B}{N_{sub}}$ is the bit-width of the sub-block and $\mathbf{C}_k^i = \frac{k!}{i!(k-i)!}$. Note that in the hybrid-redundancy allocation mode, the error-control coverage ratio is always 1 and thus $P_{protected}$ is achievable. However, a memory block may not always operate in the hybrid-redundancy allocation mode. When it is in the hard-redundancy allocation mode, the error-control coverage ratio is $\frac{N_{spare}}{N_{sub}} < 1$. Thus, some sub-blocks are not protected and the corresponding probability of uncorrectable errors, P_{no}, can be expressed as

$$P_{no} = \sum_{i=1}^{k} \mathbf{C}_k^i f^i (1-f)^{k-i}. \tag{4.2}$$

From (4.1) and (4.2), we obtain the overall probability of uncorrectable errors P_{hybrid} as

$$P_{hybrid} = R P_{protected} + (1-R) P_{no}, \tag{4.3}$$

where R is the effective error-control coverage ratio by averaging over the hybrid- and hard-redundancy allocation modes.

In comparison, ECC-based error tolerance needs $n-k$ extra memory bits to build the error checking bits for a sub-block of length k bits. This ECC code, if exists, will be a (n,k) code that can correct up to t bit errors. According to the Sphere-Packing bound [112], the following condition should hold

$$\sum_{i=0}^{t} \mathbf{C}_n^i \leq 2^{n-k}, \tag{4.4}$$

where (4.4) is a necessary but not sufficient condition. The probability of uncorrectable errors for ECC, P_{ecc}, can be calculated as

$$P_{ecc} = \sum_{i=t+1}^{n} \mathbf{C}_n^i f^i (1-f)^{n-i}. \tag{4.5}$$

Simulation results in section 4.4 show that the proposed hybrid redundancy allocation achieves better error tolerance than the ECC-based techniques under a range of error rates.

4.3.4 Managing Overhead

The overhead of the proposed technique comes from two parts: one is the spare memory cells used as hardware redundancy (same as most nanoelectronic memory systems), and the other is the LUT. Hardware redundancy in each block can be quantified by $N_{spare} \times k$ crosspoints, where k is the bit-width of each sub-block. As shown in section 4.4, this overhead is much lower than ECC-based techniques. Therefore, we will focus on the LUT overhead next.

As discussed before, each LUT entry stores one bit of mode field and the IDs of $N_{sub}+N_{spare}$ sub-blocks, each ID requiring $\lceil \log_2(N_{sub}+N_{spare}) \rceil$ bits. Thus, each LUT entry would need $(1+(N_{sub}+N_{spare}) \cdot \lceil \log_2(N_{sub}+N_{spare}) \rceil)$ bits in total. This overhead needs to be managed carefully when a large number of sub-blocks are implemented in each block.

Usually, not all the memory blocks/banks are active at the same time. This implies that a rather small number of LUT entries need to be maintained for the active blocks only. When these blocks become inactive, the corresponding LUT entries can be released to store the information of the new active blocks. Some existing policies, such as LRU or FIFO, can be employed to manage LUT context switch. Thus, the overhead of the LUT can be made significantly lower than the ECC encoding/decoding circuits, which typically require complicated finite field operations. Furthermore, the tradeoffs between error-control overhead and access performance can be adjusted based on the activities of specific workloads and system requirements. This enhances the flexibility and scalability of the proposed technique.

4.4 Evaluation and Discussion

In this section, we apply the proposed hybrid-redundancy allocation for error management in nanoelectronic crossbar memory. We will compare the error tolerance performance and induced overhead with existing techniques. The scalability of the proposed technique is also evaluated under different system settings.

4.4.1 System Configurations

For evaluation, we consider a specific nanoelectronic crossbar memory with total size $N \times N$ bits. The memory is divided into N_{bnk} banks. Within each bank, there are N_{blk} blocks and the size of each block is $B = \frac{N^2}{N_{bnk}N_{blk}}$. To exploit hybrid redundancy, we further divide each block into N_{sub} user sub-blocks and assume N_{spare} sub-blocks built by the spare crosspoints as hardware redundancy. All the sub-blocks have the same size of $k = \frac{B}{N_{sub}}$ bits. An LUT is introduced to monitor memory access activities and allocate hybrid redundancy. The mode transition is directed by the confidence level c on access pattern prediction. Based on our experiments, $c = 2$ is a reasonable setting that allows good tradeoffs between error tolerance and control complexity. The values of other parameters are listed in Table 4.1. We will initially use this system for comparison and then evaluate different settings and their impacts on error tolerance.

Without loss of generality, we consider the memory system for general purpose applications such as SPEC CPU2000 benchmarks [10]. Note that the proposed memory technique can be used in other applications with little or no modification. The simulations are performed on a simulator based on the trace-driven Dinero IV [113].

The memory access activities are captured by the memory trace files, which are collected from the Stream-Based Trace Compression (SBC) [114], where trace files of 23 benchmarks are available.

Table 4.1: Simulation setup.

N	N_{bnk}	N_{blk}	B	N_{sub}	N_{spare}	k	c
512	64	64	64	16	12	4	2

4.4.2 Error Tolerance

We first determine the error-control coverage ratio R, which is a system-level measure for evaluating error tolerance. Table 4.2 shows the results obtained from the 23 benchmarks. The average error-control ratio is observed at 98.5%. In some benchmarks, such as `mcf`, `lucas`, `art` and `applu`, nearly all memory accesses are protected by the proposed technique. As discussed before, we would like to maximize R but a full coverage is not necessary. This is because many sub-blocks may contain irrelevant data and thus do not need error protection.

To perform a comparison with ECC-based techniques, we first determine the ECC codeword width n using (4.4) for the sub-block width $k = 4$ with an error correction capability t. Table 4.3 shows the resulted ECC codes. Note that (4.4) is only a necessary condition. In other words, the ECC codes shown in Table 4.3 may not exist in practice. Thus, the results obtained for the ECC-based error tolerance (see Fig. 4.4) are optimistic, which makes our comparison conservative.

Substituting the average error-control coverage ratio R into (4.3), we can determine the probability of uncorrectable errors for the proposed technique. The probability of uncorrectable errors for ECCs with different values of t can also be determined using (4.5). We compare these results under a range of bit error rates $f \in [0.01, 0.25]$ and different ECC capabilities $t \in [1, 4]$. Figure 4.4 shows the results of error tolerance measured by the probability of uncorrectable errors. It is observed that the proposed technique achieves comparable error tolerance performance when the bit error rate

Table 4.2: Error-control coverage ratio R.

Workloads	**Coverage**	**Workloads**	**Coverage**
1.ammp	98.9 %	13.lucas	100.0 %
2.applu	100.0 %	14.mcf	100.0 %
3.apsi	99.7 %	15.mesa	90.8 %
4.art	100.0 %	16.mgrid	99.9 %
5.crafty	97.8 %	17.parser	99.6 %
6.eon	96.3 %	18.perlbmk	93.2 %
7.equake	99.6 %	19.sixtrack	97.5 %
8.fma3d	99.5 %	20.swim	99.9 %
9.galgel	99.9 %	21.twolf	99.9 %
10.gap	97.9 %	22.vortex	97.3 %
11.gcc	99.9 %	23.wupwise	99.8 %
12.gzip	98.5 %	**Average**	98.5 %

Table 4.3: Possible ECC codes at different t's with information bits $k = 4$)

t	n	ECC code (if exists)
1	7	(7, 4)
2	10	(10, 4)
3	13	(13, 4)
4	15	(15, 4)

f is relatively low ($f < 0.08$). When f is higher than 0.08, the proposed technique achieves better error tolerance than the ECC code with $t = 2$. For bit error rates $0.15 < f < 0.2$, the proposed technique shows even better performance than the ECC code with $t = 3$, indicating our technique is more effective to correct 3 bit errors in $k = 4$ memory bits. These trends clearly favor the proposed technique in dealing with a large number of errors caused by high defect/fault rates in nanoelectronic crossbar memory.

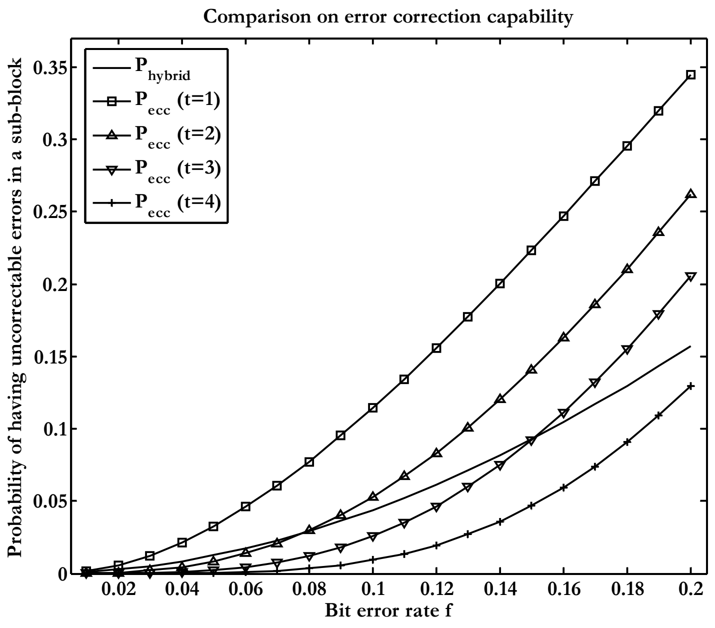


Figure 4.4: Comparison of the probability of uncorrectable errors.

4.4.3 Overhead

Table 4.4 compares the overhead of the proposed technique with ECCs. In our technique, the number of extra memory bits required for hard-redundancy allocation is $k \times N_{spare} = 48$ bits per block, whereas the error checking bits needed for implementing ECC codes at $t = 2$ and $t = 3$ are 96 and 144 bits per block, respectively. Furthermore, the proposed technique employs bit-wise XOR/AND logic to perform error correction between the sub-blocks in a redundancy mapping pair. The LUT overhead can be minimized by applying the methods as discussed in section 4.3-D. For example, when LUT entries maintain the most active bank in the $N_{bnk} = 64$ banks, the hardware overhead can be reduced to less than 3 bits per block on average (i.e., $[1 + (16 + 12) \times \lceil \log_2(16 + 12) \rceil])/64 = 2.2$ bits). In contrast, ECC codes need complicated encoding and decoding circuits and even finite field operations, which in general incur large hardware overhead. Hence, the overhead of the proposed technique is much smaller than that of the existing techniques.

4.4.4 Scalability

To assess the scalability of the proposed technique, we use different settings for N_{spare}, N_{sub}, N_{blk}, and N_{bnk}, and measure the corresponding error-control coverage ratio R. As shown in Table 4.2, benchmark `mesa` has the lowest error-control coverage ratio, indicating its access pattern is relatively difficult to predict. We will use this benchmark as a worst case study. On the other hand, benchmark `mcf` shows 100% error-control coverage ratio. This benchmark is studied as well for comparison. In addition, the average coverage ratio of all the 23 available benchmarks is also evaluated.

Table 4.4: Comparison of hardware overhead

ECC	
error correction capability t	no. of parity check bits per block: $(n-k) \times N_{sub}$
1	$3 \times 16 = 48$
2	$6 \times 16 = 96$
3	$9 \times 16 = 144$
4	$11 \times 16 = 176$

hybrid redundancy architecture	
hardware redundant bits per block	$N_{spare} \times k$ $12 \times 4 = 48$

Figure 4.5 shows the results obtained from the different values of N_{spare}. When $N_{spare} = 0$, no hardware redundancy is available and only soft redundancy is exploited. The proposed technique still achieves 99.2% error-control coverage for benchmark `mcf` and 86.7% on average for all the 23 benchmarks. However, benchmark `mesa` sees a rather low error-control coverage at 41.9%. Fortunately, the error-control coverage ratio improves significantly with the increase in N_{spare}. This indicates that hardware redundancy and soft redundancy indeed complement and enhance each other. Exploiting these observations, we can effectively manage hardware overhead for specific applications. For example, with $N_{spare} = 2$, 17 benchmarks achieve error-control coverage higher than 90%, among which 9 benchmarks achieve coverage higher than

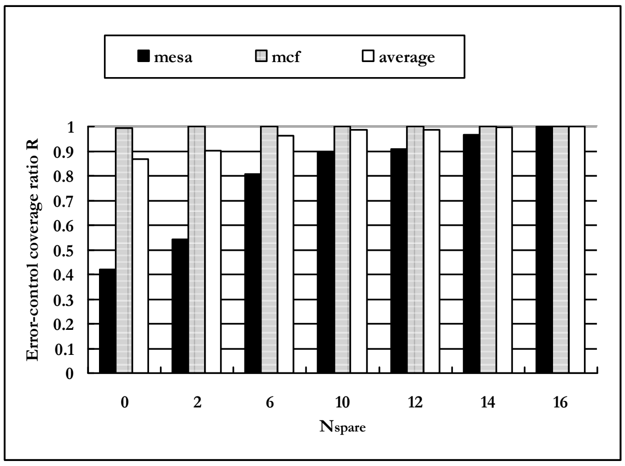


Figure 4.5: Error-control coverage ratio measured at different N_{spare}'s. Other settings are: $N = 512$, $N_{bnk} = N_{blk} = 64$, $N_{sub} = 16$.

95%. Thus, we can select a small N_{spare} for these benchmarks to achieve satisfactory error tolerance.

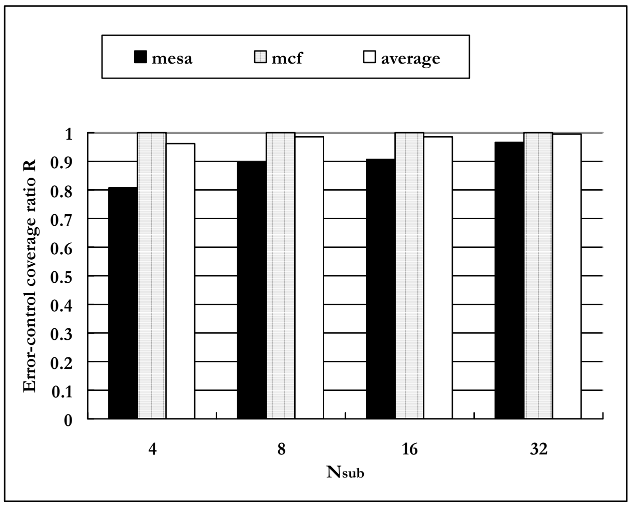


Figure 4.6: Error-control coverage ratio measured at different N_{sub}'s. Other settings are: $N = 512$, $N_{bnk} = N_{blk} = 64$, $N_{spare} = N_{sub} \times 3/4$ (if $N_{sub} = 4$, $N_{spare} = 2$ is selected).

As soft redundancy manifests at the sub-block level, we vary N_{sub} from 4 to 32 to evaluate the significance of this parameter on error tolerance. To allow a fair comparison with the baseline system, $N_{spare} = (3/4)N_{sub}$ is employed for all the simulations except for $N_{sub} = 4$, where $N_{spare} = 2$ is used. From the results in Fig. 4.6, we observe that the error-control coverage increases with N_{sub}. This is because a large N_{sub} leads to fine-grained partitioning of memory blocks. Hence, the proposed hybrid redundancy allocation can be more effective in predicting memory access patterns. However, increasing N_{sub} also leads to a large LUT and hence a tradeoff must be considered for this parameter. In this case study, both $N_{sub} = 8$ and 16 can achieve approximately 90% error-control coverage ratio for the worst case benchmark. However, $N_{sub} = 8$ might be more favorable due to its smaller LUT overhead.

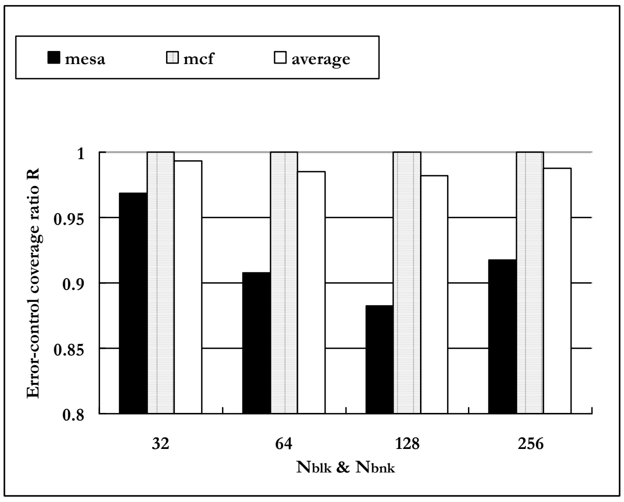


Figure 4.7: Error-control coverage ratio measured at different N_{blk}'s and N_{bnk}'s. Other settings are: $N = 512$, $N_{bnk} = N_{blk}$, $N_{sub} = 16$, $N_{spare} = 12$.

Figure 4.7 shows the results of using different vales of N_{blk} and N_{bnk} in the nanoelectronic memory. In general, when smaller N_{blk} and N_{bnk} (e.g., smaller sub-memory) are used, the nanoelectronic memory will see improved robustness. Again, this will involve a careful design tradeoff as smaller banks/blocks may incur larger overhead in interbank connectivity.

Chapter 5

Dynamic Redundancy Allocation for Nanocomputing

5.1 Introduction

Since the invention of CMOS-based integrated circuits (IC), computer system design has reaped a dramatic improvement in computational performance. The key enabling technologies are a combination of advances in semiconductor process and design methodology, and innovations in computer architecture. As conventional CMOS technology quickly approaches the end of roadmap, many novel nanoelectronic devices have emerged as the potential computational substrate for nanoscale integration. However, the emerging nanoelectronic technology is accompanied by some new challenges that may have a profound impact on architecture-level design and optimization.

It is widely acknowledged that nanoscale integration will no longer enjoy high reliability as the conventional CMOS technology. At the system level, redundancy-based fault-tolerance techniques, such as N-modular redundancy and multiplexing logic [9], were proposed to use hardware redundancy for recovering from the faults. Recently, NAND multiplexing is extended to a rather low degree of redundancy [84] and new models are presented in [85, 86] to study the multiplexing systems. Redundancy based fault tolerance is considered very effective for nanocomputing systems due to the abundant device resources offered by nanoscale integration. Most existing work uses fixed modular redundancy, which lacks adaptivity to different reliability requirements at runtime. An improved strategy [87, 88] was developed to frugally allocate redundancy so as to avoid the resource usage from exponentially increasing.

In nanocomputing systems, the problem of fault tolerance is closely related to high-performance parallel execution. Traditionally, architectural level research for performance improvement has been directed toward exploiting various levels of parallelism. This trend is clearly reflected in simultaneous multithreading (SMT) [11, 12] and chip multiprocessors (CMP) [13, 14]. Nanocomputing systems show great potential to support this trend of parallelism exploitation due to the ultra-high integration density. However, with high defect/error rates, the benefit of parallelism may be limited by the redundant computation necessary for defect/error detection and recovery. Massive parallelism does not necessarily lead to high performance unless the system can effectively recover from unpredictable upsets. Furthermore, the unpredictable upsets also lead to performance unpredictability. This is especially a problem for real-time applications that require stable and predictable execution performance, e.g., multimedia communications that have tight protocol timing specifications. Thus, it is critical to balance the often conflicting requirements on performance and reliability.

The interplay between performance and reliability gives rise to a challenging problem on resource management that is critical to nanocomputing systems. This compels us to explore nanoarchitecture solutions in order to unfold the full potential of nanocomputing paradigm. In [34, 28], we propose a dynamic redundancy allocation technique that enables reliable and scalable parallelism. The proposed technique manages the parallelism at an optimal level so that both fault tolerance and performance enhancement can be achieved in a coherent manner. Different from the existing redundancy management strategies [87, 88, 9], the proposed technique explicitly considers the availability of the computational resources and the varying requirements on reliability and performance, and therefore is more flexible in redundancy allocation across the instructions at runtime. As the proposed approach can be applied to address both permanent defects and transient errors, we do not differentiate between them and refer the general term *fault* to both.

The rest of the chapter is organized as follows. In section 5.2, we present the nanoarchitecture model and control mechanism as the platform for dynamic redundancy allocation. In section 5.3, we provide the details of the proposed dynamic redundancy allocation. Simulation results are discussed in section 5.4 to evaluate the effectiveness of the proposed technique.

5.2 Nanoarchitecture Model

Nanocomputing systems feature highly regular, locally connected, and data parallel architectures that match well to the ultra-high speed and integration density offered by nanoelectronics [91, 92]. As shown in Fig. 5.1, the underlying nanoarchitecture may be multiclustered [93, 94, 20], where each cluster consists of multiple functional units that can be tuned specifically for a certain category of applications. The workload is dispatched to the individual clusters using predefined metrics (e.g., slack [95], deadline [96], and mean response time [97]) to achieve overall performance optimization.

While specific nanocomputing systems may be constructed with various configurations, in this chapter we consider a generic nanoarchitecture model where each cluster contains a pool of three basic functional units: computation elements (CE), memory units (MEM), and voters, as shown in Fig. 5.1(b). The CEs and voters can be implemented by programmable nanoelectronic logic arrays (e.g., nanowire crossbars) according to the defined functions, as shown in [100, 20]. Memory units can also be implemented in a similar way [102]. As the computation is inherently fault-prone, it is necessary to execute multiple copies by exploiting the spatial and temporal redundancy in the CEs. The results are then compared by the voters to detect and correct errors based on majority voting. The voters also provide localized control within the cluster by allocating and managing CEs and MEMs according to certain allocation strategies such as the one proposed in section 3. As the control and redundancy allocation are performed distributively, the implementation can be simplified to LUT, as shown in section 3.4. Same as most redundancy-based architectures, the above nanoarchitecture model relies upon the correct voter operation. A complete characterization of reliability issues relevant to voter design needs to be done before

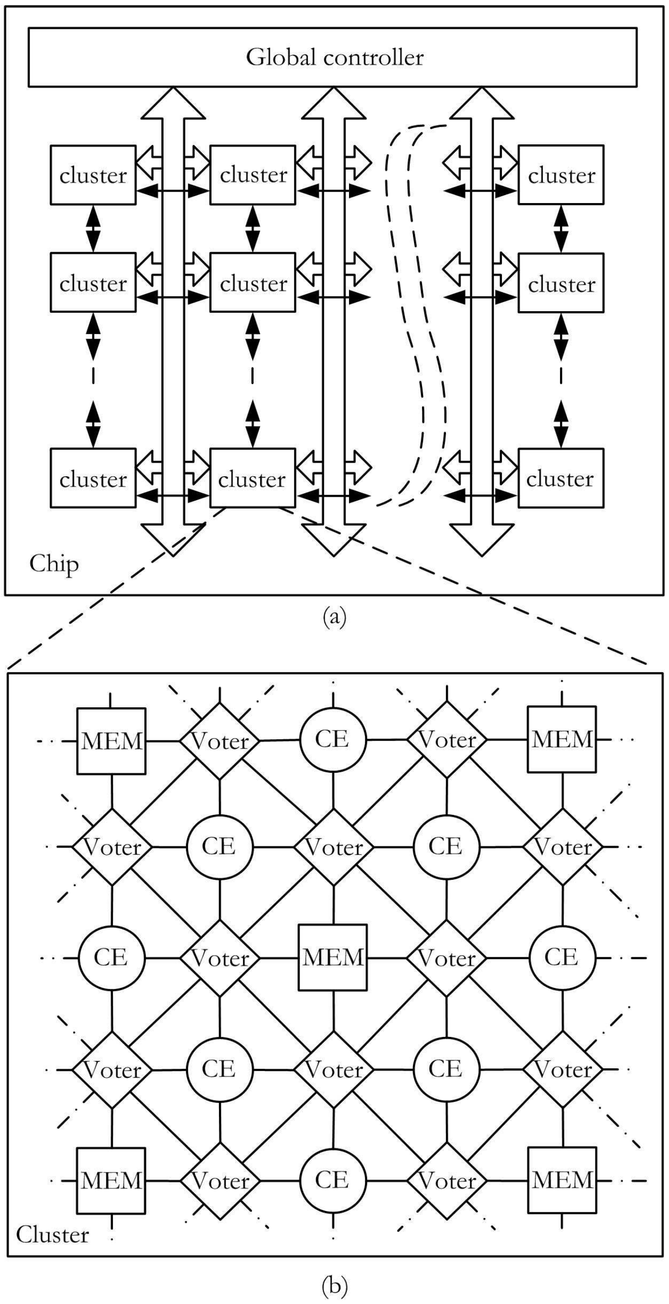


Figure 5.1: A conceptual illustration of nanoarchitecture model: (a) top-level view and (b) cluster configuration.

the application of the proposed technique. We assume the operation of voters to be correct by using reliability enhancing design techniques. Thus, faults occurred in the voters have a much lower rate than that in the CEs and hence are ignored in the following discussion. In addition, other techniques, e.g., error correcting codes, are effective to deal with faults in interconnect and memory.

To exploit parallel processing, pipelined architecture is implemented in the clusters. For the purpose of demonstration, we consider a prototype pipeline including four stages: *issue*, *execute*, *compare*, and *complete*. Note that we do not lose any generality with this architecture as further split of stages into more complex pipeline is possible depending on the requirement of specific implementations. In the issue stage, voters will be invoked to manage the instruction processing. To facilitate fault tolerance, each voter will initially allocate a number of CEs (spatial redundancy) to perform redundant computation. In the execute stage, the selected CEs execute the instruction and return the results to the voters. After the execution is finished, the CEs will be released to standby. In the compare stage, the voters evaluate the returned results and determine the correctness by majority voting. If majority voting is unable to resolve the disputed results, the initially allocated CEs fail to achieve fault tolerance. The voters will store the unconfirmed results to MEMs and then allocate additional CEs to execute the same instruction (temporal and spatial redundancy) until the results eventually get confirmed. In the complete stage, incorrect results and related speculations are pruned leaving the correct results for future use. Note that to avoid resources being depleted by redundant computing, a dedicated resource management is needed to balance redundant computing and parallel processing. We will elaborate this point in section 3.

Figure 5.2 shows an example of the localized control in the four-stage pipeline. In the issue stage, a voter issues the instruction to its neighboring CEs. To provide fault tolerance for instruction execution, the voter will first determine the amount of redundancy (i.e., CEs) as denoted by R. Figure 5.2(a) shows two cases where $R = 3$ and $R = 4$, respectively. The details of the redundancy allocation policy will be discussed in section 3. To improve the efficiency, the voter will first try to allocate redundancy using the nearest CEs, as shown in the first case of Fig. 5.2(a). If failed, the voter will collaborate with other voters to collect enough CEs, as shown in the second case of Fig. 5.2(a). The latency of voter bypass is considered as a

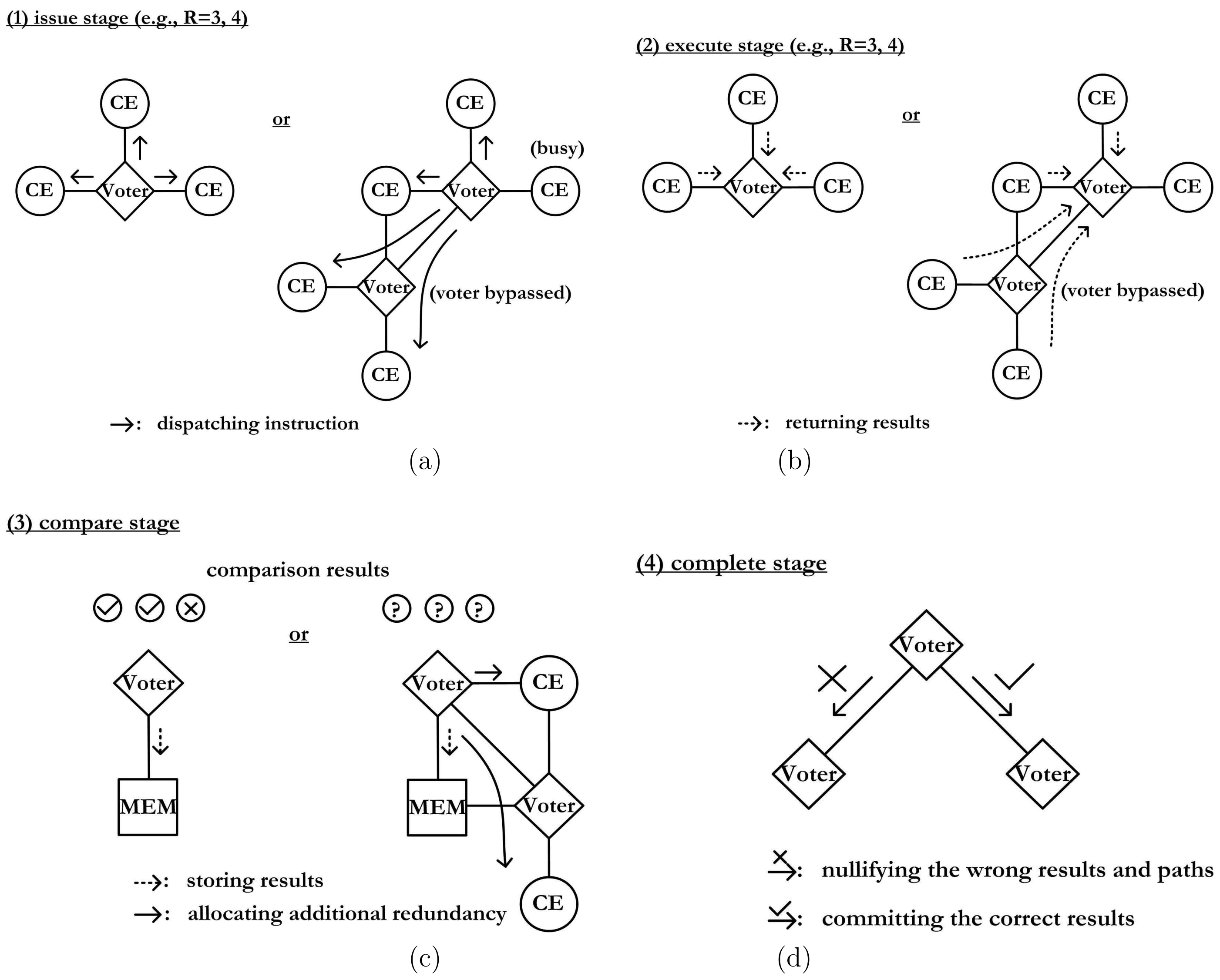

(1) issue stage (e.g., R=3, 4)
CE
Voter
or
(busy)
(voter bypassed)
→: dispatching instruction
(a)
(2) execute stage (e.g., R=3, 4)
or
(voter bypassed)
- - →: returning results
(b)
(3) compare stage
comparison results
or
MEM
- - →: storing results
→: allocating additional redundancy
(c)
(4) complete stage
nullifying the wrong results and paths
committing the correct results
(d)

Figure 5.2: An example of localized pipeline control: (a) issue stage, (b) execute stage, (c) compare stage, and (d) complete stage.

timing component in the issue stage. If more CEs need to be collected that might cause long and nondeterministic latency, the voter will wait for the neighboring CEs to become available after they finish the execute stage (which may take additional 1–2 cycles). This is a type of structural hazard. In the execute stage, CEs finish the execution and return the results to the voter in charge of this instruction (see Fig. 5.2(b)). Next, in the compare stage, the comparison logic determines whether the results are correct or not. Meanwhile, the results are stored temporarily in the MEM for future reference. It is possible that the results cannot achieve agreement due to the faults in the execution (as the second case shown in Fig. 5.2(c)). Thus, the completion for this instruction has to be postponed and the voter will need to allocate additional redundancy to recover the faults. Eventually, when the execution results are confirmed, the voter will process in the complete stage (as shown in Fig. 5.2(d)). The incorrect results and the associated speculative execution paths are nullified and pruned by the voter, while the correct results are propagated by the voter. For other types of faults with relatively deterministic nature (e.g., the clustered faults), the affected voters may be masked out to minimize the performance degradation.

As the instruction execution is fault-prone, subsequent instructions are issued using unconfirmed results of the precedent instructions to solve data dependencies for high performance. Thus, the execution of instructions is inherently speculative. As shown in Fig. 5.3, the unconfirmed results of the precedent instructions propagate to provide the operands for speculative execution of the subsequent instructions (Fig. 5.3(a)). Again, if the voter is unable to allocate enough redundancy using the nearest CEs, it will cooperate with neighboring voters to issue speculative executions to other CEs, as shown in the second case in Fig. 5.3(a). After these results get confirmed, the wrong paths of the execution will be pruned while the right paths remain, thereby forming a tree (Fig. 5.3(b)) to complete the instruction execution.

5.3 Dynamic Redundancy for Reliable and Scalable Parallelism

In this section, we will apply the above nanoarchitecture model to investigate the intrinsic relationship between redundant computation and massive parallelism in nanocomputing systems. We will then develop a dynamic redundancy allocation technique that enables reliable and scalable performance in a coherent manner.

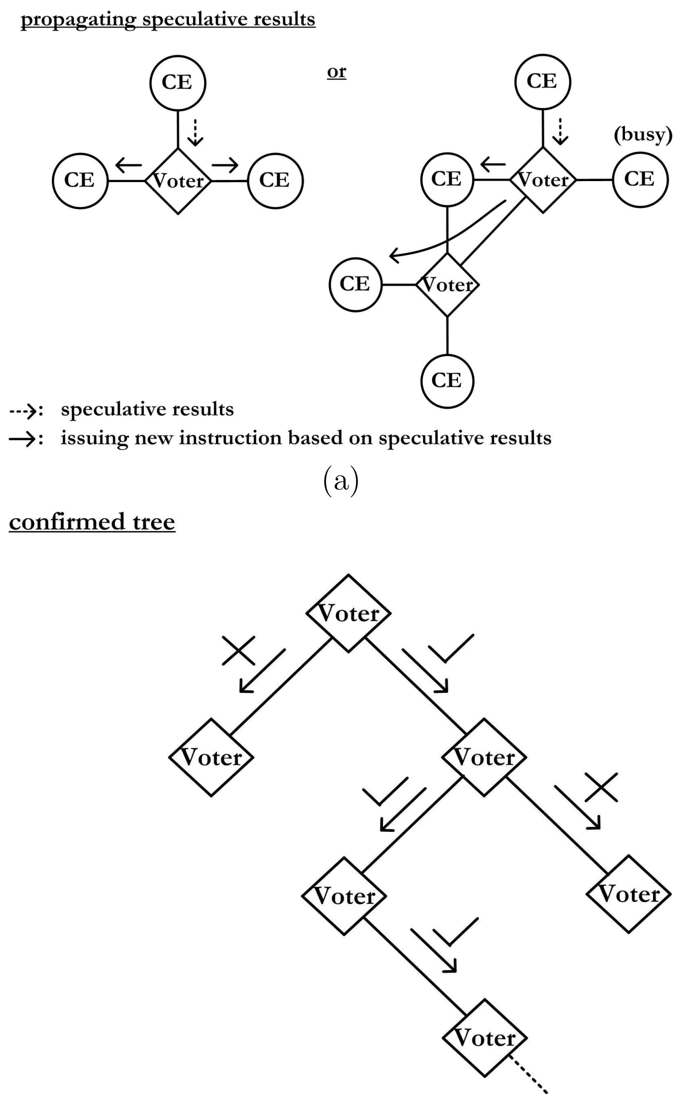

propagating speculative results
or
CE
Voter
(busy)
--->: speculative results
—>: issuing new instruction based on speculative results
(a)
confirmed tree
(b)

Figure 5.3: Speculative execution for high performance: (a) propagating speculative results and (b) a tree formed after result confirmation.

5.3.1 The Underlying Problem

Nanocomputing systems feature massive parallelism that can be exploited for high performance. However, massive parallelism does not necessarily lead to high performance given the high fault rates and fault recovery penalties. Although in theory fault tolerance can be achieved by redundant computation, this approach inevitably competes with parallel processing for hardware resources. How to cooperatively manage and utilize massive parallelism for reliable and scalable performance is critical for realizing the potential of nanoscale integration.

We illustrate this problem in Fig. 5.4. At a given time, all the pending instructions will fall into two categories according to the dependency relations: some instructions have to be executed in series, as they closely depend on the results of their precedent instructions; and some instructions can be executed in parallel, as they are loosely coupled (e.g., they may come from different threads or become relatively independent after dependency is safely removed). Therefore, based upon these dependency relations, the instructions can be partitioned into multiple chains, where in each chain the instructions are sequentially executed (e.g., instructions 1 through L in chain 1, as shown in Fig. 5.4, have to be executed one by one). Meanwhile, instructions in the different chains can be executed in parallel as a group (e.g., the instruction 2 in all the chains as shown in Fig. 5.4 can be organized as group 2 and executed simultaneously). On one hand, it is generally desired to execute as many instructions in parallel as possible for maximal throughput. On the other hand, in nanocomputing systems each instruction requires multiple execution copies for fault tolerance. Thus, the need of providing redundancy for fault tolerance conflicts with the motivation for high performance. When the subsequent instructions in a chain are issued to avoid performance slowdown, their precedent instructions might not have got their results confirmed yet. Thus, the subsequent instructions have to be speculatively issued with multiple unconfirmed results. Consequently, how to optimally allocate the available hardware resources (the number N of CEs as shown in Fig. 5.4) is complex. Specifically, the hardware resources need to be allocated to each parallel executed group (e.g, R_j of CEs allocated to the j^{th} instruction group). Meanwhile, within each group the hardware resources need to be further distributed to the individual instructions belonging to multiple chains and the redundant execution of each instruction as well.

It can be observed that the requirement for parallel processing and the need for reliable computation are competing against each other for hardware resources. When the fault rate is high, all the instructions including the precedent instructions are difficult to complete with confirmed results. To maintain speculative execution for

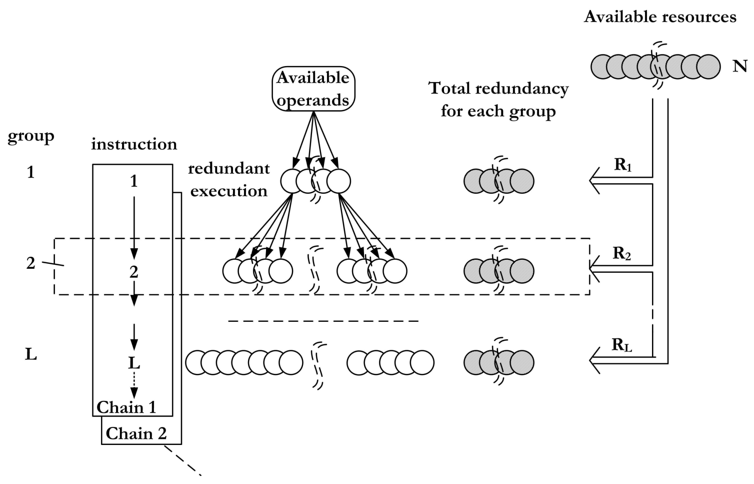


Figure 5.4: The underlying problem of redundancy allocation.

high parallelism, the utilization of redundancy will exponentially increase and may eventually deplete all the available resources, causing frequent structural hazards. Consequently, the course of execution will be stalled frequently because the unconfirmed instructions cannot get enough redundancy to complete the processing, and speculative execution of the subsequent instructions cannot have resources either to continue exploiting parallelism. This problem will be exacerbated if there are multiple simultaneous threads, where each group will contain many instructions due to the relatively weak dependency. Therefore, how to optimally allocate resources for reliable and scalable performance is a fundamental problem that must be addressed in nanocomputing systems. Furthermore, it is also critical to adjust the redundancy allocation at runtime according to different requirements on reliability and performance, which may vary during the course of program execution.

5.3.2 Parallelism Level and the Implications to Reliable Performance

As shown in Fig. 5.4, there are two ways to improve performance: one is to break the instructions into more chains so that more instructions can be simultaneously processed in each group; the other is to execute more groups concurrently while they are possibly in different pipeline stages. The first approach is largely determined by the nature of dependency inherent in specific programs, which is beyond control of nanocomputing architecture design. On the other hand, the number of groups executed in parallel can be increased by performing speculative execution. In this chapter, we define the *parallelism level* as the number of instruction groups that can be processed concurrently. Note that the parallelism level is defined for an entire cluster, not just for a single voter or CE. As an example, a conservative design may execute an instruction only when all the dependent instructions are completed and confirmed with no faults. The instruction level in this case is 1. Another example is an aggressive design, where the system may speculatively issue and execute instructions even when the operands from precedent instructions are still not confirmed yet. As a result, there are more instruction groups being processed at the same time (possibly in different pipeline stages), which increases the parallelism level (denoted as L as shown in Fig. 5.4). In fault-free systems, the relationship between parallelism level and performance is straightforward: a higher parallelism level leads to better performance. In fault-prone systems, the complete stage in the pipeline may take an unpredictable number of cycles, depending on how quickly the faulty results can be recovered. This complicates the relationship between parallelism level and performance. It is likely that, while many instructions can be executed in parallel, they cannot obtain enough hardware resources to confirm their results quickly via redundant execution. As a result, instructions are frequently stalled in the pipeline, leading to a superficially "high" parallelism level which does not deliver high performance. Hence, the actual parallelism level needs to be determined carefully.

We now examine the relationship between the parallelism level and reliable performance. We assume that there are totally M instructions in the program, which can be partitioned into multiple chains and organized into totally K groups. Assume that the parallelism level is always L as shown in Fig. 5.4, which implies that the cluster can always process L groups of instructions simultaneously. The pipeline depth is assumed to be D and the pipelined execution may be performed distributively on multiple CEs. The total time T for processing the M instructions specifies the execution performance. It consists of two components: regular execution time T_e (the time spent on the regular processing in all the pipeline stages, excluding stalls due to faults) and fault recovery penalty T_f (the time spent on the re-execution when uncorrectable faults are detected), expressed as

$$T = T_e + T_f. \tag{5.1}$$

In the regular execution time T_e, D cycles are needed for the very first group of instructions to go through the pipeline and complete. As the parallelism level is assumed to be L, the pipeline can only filled to the extent of $\frac{L}{D}$. After the pipeline is initially filled by the first L instructions, ideally the cluster can start processing the following groups one after another. This procedure only needs $K - 1$ cycles if the parallelism level is equal to the pipeline depth (i.e., $L = D$). However, if $L < D$, there are always some "bubbles" in the pipeline (which can be considered as having $(\frac{D}{L} - 1)$ NOP groups inserted immediately after each instruction group without delivering any performance). Thus, in fault-free systems, the parallelism level is upper-bounded by the pipeline depth, i.e., $L \leq D$. The fault-free systems can achieve the optimal performance when $L = D$. On the other hand, in fault-prone systems, the parallelism level L is no longer bounded by the pipeline depth D. If the faulty results cannot be quickly recovered, the compare stage may take an unpredictable number of cycles to finish. As a result, there might exist $L > D$ instruction groups being processed concurrently. However, this does not deliver high performance as the regular execution time T_e will not be reduced. In fact, this case indicates the existence of stalls, and hence the non-zero fault recovery penalties T_f, as will be discussed later. The regular execution time T_e can be derived as

$$T_e = \begin{cases} D + (K-1)\frac{D}{L}, & 0 < L \leq D, \\ D + K - 1, & L > D. \end{cases} \tag{5.2}$$

From (5.2), the optimal parallelism level should be $L^{opt} = D$ for both fault-free and fault-prone systems. Note that in ideal fault-free systems the actual parallelism level is always smaller than L^{opt}. This is because not all the opportunities of parallel execution can be detected and exploited in practice, especially considering pipeline bubbles due to control and structural hazards. When L is much less than L^{opt}, the utilization of parallelism tends to be over-conservative. As a result, execution takes a longer time according to (5.2), leading to performance slowdown even when there is no penalty T_f. On the other hand, faults in real systems may drive the parallelism level away from the optimal L^{opt}. Once uncorrectable faults occur, the comparison fails to resolve the disputed results, thereby the system has to allocate additional redundancy (temporal and/or spatial) for re-execution. Hence, stalls occur frequently that hold more instructions than expected. This may increase L beyond L^{opt}, especially if the parallelism is over-aggressively utilized. However, this increase in L only leads to performance degradation. Thus, both over-conservative and over-aggressive utilization of parallelism will cause performance slowdown. An optimal nanoarchitecture solution should aim at closing the gap between L and L^{opt} as much as possible.

5.3.3 Scalable Parallelism via Dynamic Redundancy Allocation

We now present the dynamic redundancy allocation technique that optimizes the resource management under high fault rates, thereby enabling scalable parallelism for jointly achieving reliable and high-performance nanocomputing.

Assume that the redundancy allocated for parallel execution of the instructions in group j is R_j of CEs and the total number of available CEs is N. The central idea of our approach is to adjust R_j dynamically so that the parallelism level L is maintained at L^{opt}. There is a constraint regarding the availability of redundancy, expressed as follows

$$\sum_{j=1}^{L} R_j \leq N. \tag{5.3}$$

If the requests for redundancy are more than that available, unsatisfied requests combined with new requests during the following cycles will be accumulated quickly and eventually push the parallelism away from the optimal level. Simulation results in section 4 clearly show that techniques unaware of (5.3) lead to performance degradation.

Consider the fact that the available redundancy N may vary over time due to the runtime variations of instruction processing. Instead of trying to find the specific amount of redundancy allocated to each instruction, we focus on the relative ratios that are more stable for allocating the available redundancy. Thus, the constraint (5.3) can be recast as

$$\sum_{j=1}^{L} \alpha_j \leq 1, \tag{5.4}$$

where $\alpha_j = \frac{R_j}{N}$ defines the ratio between R_j and N, both of which are time-varying in general.

As discussed before, when the actual parallelism level is greater than the optimal L^{opt}, stalls due to uncorrectable faults occur, which hurts the overall performance. Since speculative execution based on the faulty results of the precedent instructions has already consumed too many resources at this time, it does not make sense to continue allocating redundancy for further speculation. Therefore, we only need to allocate redundancy for the instructions in groups from 1 up to L^{opt} at runtime, i.e.,

$$\alpha_j = \begin{cases} \alpha_j^{opt}, & \text{if } 1 \leq j \leq L^{opt}, \\ 0, & \text{if } j > L^{opt}. \end{cases} \tag{5.5}$$

where $\{\alpha_j^{opt}\}_{j=1}^{L^{opt}}$ are the optimal ratios of assigning R_j of CEs to the instructions in group j and $\sum_{j=1}^{L^{opt}} \alpha_j^{opt} = 1$ so that (5.4) can be always satisfied. Doing so avoids over-aggressive speculative processing that may potentially deplete the resources.

In order to determine the optimal ratios α_j^{opt}'s, we consider the fault recovery penalty T_f in (5.1). It is determined by the number of re-executions as well as the penalty associated with these re-executions. Note that the instructions have different probabilities of being re-executed. Thus, we consider the mean value t_f, which is the statistical measure of T_f, expressed for group 1 through L as

$$t_f = \sum_{j=1}^{L} P_j \overline{\delta}, \tag{5.6}$$

where P_j is the probability of re-execution of instructions in group j and $\overline{\delta}$ is the average penalty of each re-execution.

To calculate the re-execution probability P_j, we need to consider the probability of having uncorrectable faults. Assume that due to faults a CE will generate wrong results with a probability f, referred to as the fault rate of the CEs. If an instruction is executed with redundancy R, faults cannot be resolved when the number of correct results is less than two, as the majority voting cannot select the correct results out of the incorrect ones. Note that the majority voting may make wrong decisions under certain fault patterns. This is a limitation of majority voting, which will affect all fault-tolerant techniques using majority voting or similar schemes for fault detection. However, fault tolerance can be achieved at a confident level when assigning enough amount of redundancy R for a given fault rate f. The probability of having uncorrectable faults due to lack of correct results is given by $q(f, R)$ as follows

$$q(f, R) = f^R + R(1-f)f^{R-1}. \tag{5.7}$$

If the redundancy for the instructions in group 1 is R_1, these instructions will have a re-execution probability $P_1 = q(f, R_1)$. Since the instructions in group 2 with redundancy R_2 are executed speculatively when their precedent instructions in group 1 have not been confirmed yet, each of the R_1 copies of instructions in group 1 is assigned with $\frac{R_2}{R_1}$ redundancy on average (assuming R_1 is a factor of R_2) when executing the corresponding instructions in group 2. The re-execution probability of the instructions in group 2 is thus expressed as $P_2 = P_1 + (1 - P_1)q(f, \frac{R_2}{R_1})$. This is obtained based on the following observations. If the instructions in group 1 need to be re-executed (with probability P_1), the corresponding instructions in group 2, which are speculatively executed based on the instructions in group 1, should definitely be re-executed as well. On the other hand, even if the instructions in group 1 do not need re-execution (with probability $1 - P_1$), the instructions in group 2 may still have to be re-executed (with probability $q(f, \frac{R_2}{R_1})$) due to their faulty CEs. Proceeding in the same fashion, we obtain the re-execution probability for the instructions in group j as

$$P_j = P_{j-1} + (1 - P_{j-1})q(f, \frac{R_j}{R_{j-1}}), \tag{5.8}$$

where $j > 1$ and $P_1 = q(f, R_1)$.

From (5.7) and (5.8), the re-execution probability of all the instructions can be calculated in a recursive way. Thus, the fault recovery penalty t_f can be evaluated according to (5.6). Since α_j's determine the redundancy allocation, the optimal ratios α_j^{opt}'s should satisfy

$$\text{minimize:} \sum_{j=1}^{L^{opt}} P_j, \tag{5.9}$$

$$\text{subject to:} \sum_{j=1}^{L^{opt}} \alpha_j = 1. \tag{5.10}$$

It is in general very difficult to analytically determine the values of α_j^{opt}'s. However, the following guidelines can be applied to estimate α_j^{opt}'s.

(i) From (5.7), fault tolerance is possible if $R > 1$, which is very much in line with the well-prevalent notion of redundancy-based execution. Thus, $\frac{R_j}{R_{j-1}}$ in (5.8) should be greater than 1, implying $\alpha_j > \alpha_{j-1}$. Thus, we should increase the values of α_j^{opt}'s from the precedent instructions to the subsequent instructions.

(ii) From (5.8), the re-execution probability (and hence the recovery penalty) are accumulated from the precedent instructions to the subsequent instructions. This implies that quick confirmation of the precedent instructions is important. Given this observation, more resources should be made available for fault tolerance in each copy of the precedent instructions. According to (5.8), a practical way to achieve this is to make $\frac{R_{j+1}}{R_j} < \frac{R_j}{R_{j-1}}$. This requires $\frac{\alpha_{j+1}^{opt}}{\alpha_j^{opt}} < \frac{\alpha_j^{opt}}{\alpha_{j-1}^{opt}}$, where the increase in the ratios α_j^{opt}'s needs to be slowed down from the precedent instructions to the subsequent instructions.

By minimizing the fault recovery penalty, the proposed dynamic redundancy allocation technique is able to improve not only the performance but also the predictability of performance. This will be shown in section 4.

Note that the above analysis can be easily extended to the general case where $\frac{R_j}{R_{j-1}}$ may not be an integer. Assume that $\frac{R_j}{R_{j-1}} = r + \Delta r$, where we define $r = \lfloor \frac{R_j}{R_{j-1}} \rfloor$ and $\Delta r = \frac{R_j}{R_{j-1}} - r$. When the average redundancy for each speculative execution is $\frac{R_j}{R_{j-1}}$ (where R_{j-1} is the total redundancy of the precedent group of instructions), we should allocate r and $r+1$ redundancy for speculative executions based on the $R_{j-1}(1-\Delta r)$ and $R_{j-1}\Delta r$ unconfirmed results, respectively, of the precedent instructions. Therefore, (5.7) can be extended to

$$q(f,R) = \begin{cases} f^R + R(1-f)f^{R-1}, & R \in \mathbf{Z}, \\ (1-\Delta r)q(f,r) + \Delta r\ q(f,r+1), & \text{otherwise}, \end{cases} \tag{5.11}$$

where $r = \lfloor R \rfloor$ and $\Delta r = R - r$.

Table 5.1: An example of selecting optimal redundancy allocation ratios.

N	**f**	$\alpha_{\mathbf{1}}^{\mathbf{opt}}$	$\alpha_{\mathbf{2}}^{\mathbf{opt}}$	$\alpha_{\mathbf{3}}^{\mathbf{opt}}$	$\alpha_{\mathbf{4}}^{\mathbf{opt}}$
20	0.1	0.10	0.20	0.30	0.40
20	0.2	0.15	0.25	0.25	0.35
20	0.3	0.15	0.25	0.25	0.35
20	0.4	0.20	0.25	0.25	0.30
20	0.5	0.20	0.25	0.25	0.30
40	0.1	0.075	0.150	0.300	0.475
40	0.2	0.075	0.200	0.350	0.375
40	0.3	0.100	0.275	0.300	0.325
40	0.4	0.100	0.275	0.300	0.325
40	0.5	0.125	0.275	0.275	0.325

We use an example to show the selection of the optimal allocation ratios. In this example, $L^{opt} = D = 4$ is assumed. The total number of available CEs is $N = 20$ and $N = 40$ for demonstration purpose. We vary the fault rate in the range of

$f \in [0.1, 0.5]$ and search for α_j^{opt}'s to minimize the fault recovery penalty. The search is done in the space where α_j^{opt}'s are rounded to the closest $1/N$ so that $N\alpha_j^{opt}$'s are integers. As shown in Table 5.1, when N increases, for the same f we should allocate relatively more redundancy for the subsequent instructions. This is because the precedent instructions can obtain enough redundancy for quick confirmation even with the smaller ratios. If f increases but N is fixed, more redundancy should be allocated to the precedent instructions to minimize the re-execution probabilities.

Note that the α_j^{opt}'s can be determined in the design phase based on availability of N and estimated f from the physical systems. These ratios can be employed thereafter to guide the dynamic redundancy allocation at runtime.

5.3.4 Dynamic Redundancy Allocation Algorithm

The general procedure of dynamic redundancy allocation is summarized in Algorithm 2. Assume that the instruction being processed is in group j, and there are totally k voters in this group. The goal of this algorithm is to determine when to allocate redundancy and how much redundancy should be allocated.

If the parent voters notify the current voter that the operands the current voter is using have been proved to be invalid, the current voter should nullify all the involved results and propagate the information to the involved voters. Otherwise, the current voter needs to check whether the execution has achieved majority agreement. If the results are not confirmed yet, the current voter will try to allocate more redundancy.

Next, the amount of redundancy is determined according to the proposed dynamic redundancy allocation strategy. This function is generalized in Algorithm 3. It is essentially performing $\frac{N\alpha_j^{opt}}{k}$. By doing so, the redundancy can be distributed evenly to each instructions in this group for the sake of fairness. Note that prioritized allocation in favor of specific instructions is also feasible. The complexity of redundancy allocation consists mainly of LUT, as the voters just perform LUT-based operations to allocate redundancy as shown in Algorithm 2 and Algorithm 3. Furthermore, some operations shown in Algorithm 3 do not need to be performed every cycle. For example, the operation for getting α_j^{opt} is not necessary if the group number j remains

Algorithm 2: The general procedure of dynamic redundancy allocation.

```
Input:
$status\_parent$ (status of the parent voter),
$confirmation$ (confirmation status of the current voter),
$N$ (total availability),
$j$ (group index),
$k$ (number of voters simultaneously processing in the same group)
begin
    while the voter is active do
        if $status\_parent$ is $Invalid$ then
            nullify all the execution under control of this voter
            propagate info. to the subsequent voters
        end
        else if $confirmation$ is $False$ then
            $R = calculate\_redundancy(N, j, k)$
            allocate $R$ redundant computation from neighboring CEs
        end
    end
end
```

Algorithm 3: *calculate_redundancy*(N, j, k).

```
Input: $N$, $j$, $k$
Output: $R$
begin
    if $j$ is changed then
        read $\alpha_j^{opt}$ from LUT indexed by $j$
    if $N$ is changed then
        LUT-based calculation for $N\alpha_j^{opt}$
    LUT-based calculation for $R = N\alpha_j^{opt}\frac{1}{k}$
    return $R$
end
```

the same. Thus, the algorithm can come up with a timely decision. In addition, the timing efficiency can be further improved by separating the issue stage into two stages.

5.4 Evaluation and Discussion

In this section, we evaluate the proposed dynamic redundancy allocation technique. We first explain the evaluation methodology and then discuss the simulation results.

5.4.1 Methodology

The simulations are implemented in C program based on the proposed dynamic redundancy allocation technique. Here we are interested in architecture-level abstracted behaviors instead of implementation or program details. To evaluate the effectiveness of the proposed technique, we compare our work with two existing techniques: one uses fixed dual-modular redundancy, where speculative execution always takes two computation units; and the other is an improved technique [87, 88], which assigns either one processing unit if the previous instruction can still be confirmed, or two if the previous instruction is not confirmable. In order to ensure a fair comparison, all the settings are equivalent to the two existing techniques except for the redundancy allocation algorithm. In particular, we compare the performance of our technique with the two existing techniques under the same number of CEs. We also make the same assumption on the utilization of MEMs. The focus of this work is on how to efficiently utilize available computational hardware resources (CEs) to jointly achieve fault tolerance and high performance under a range of fault rates. In the simulations, all the instructions are dependent on their immediate precedent instructions. In such an example, instructions are speculatively executed based on unconfirmed results. Other programs of different dependencies can be considered as a combination of multiple copies of this specific example but with different numbers of instructions and appearing at different phases in the program. There are totally 40 CE's in the cluster and $L^{opt} = D = 4$. The optimal ratios α_j^{opt}'s for dynamic redundancy allocation are selected according to the method described in section 3.

In these simulations, we deliberately introduce faults into the execution results of CEs and evaluate the performance in terms of cycle per instruction (CPI). These faults represent the possible upsets during CE execution as well as in the associated interconnects and MEM units. We evaluate our technique using abstracted instructions, which do not depend on specific applications but do represent many key computational activities in real tasks. Note that only the correctness of execution, rather than the actual execution results, matters to this evaluation. Thus, abstracted instructions not targeting program details are sufficient for this study. The voters are assumed to be reliable as explained in section 2. Each time when a CE is processing an instruction, the result may go wrong at a certain rate, which is contributed by both permanent defects and transient errors. Without a general fault model for nanoscale systems, we may assume for the purpose of demonstration that faults occur independently. Note that the fault rate of CEs is generally higher than the fault rate of individual nanoelectronic devices. Thus, a large range of fault rates are evaluated for the proposed technique to account for the effect of increase in failure probability at the coarse granularity.

5.4.2 Average Performance

Figure 5.5 compares the average CPI for fault rates ranging from 0 to 0.5. The fault rates here are the possibility that the execution results of CEs go wrong. As we consider in-order execution, the minimum CPI is close to 1. Employing the proposed technique, the average CPI can approach this bound even at rather high fault rates. In addition, our technique achieves approximately $20\% - 30\%$ improvement on the average CPI as compared to the two fixed redundancy allocation techniques.

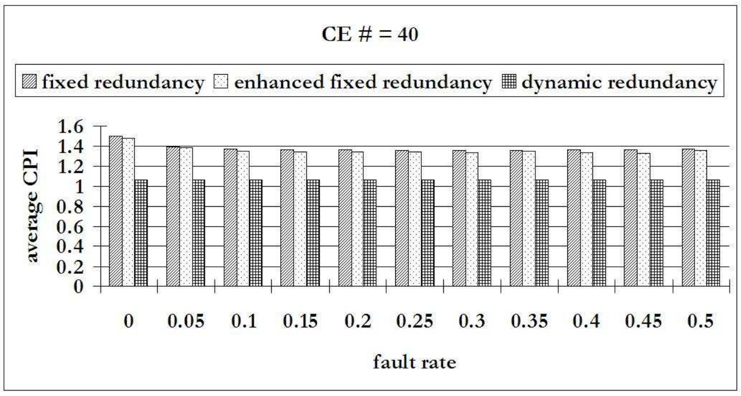


Figure 5.5: Comparison of average CPI.

An interesting result is that when the fault rate increases, the proposed dynamic redundancy technique maintains a very stable performance as compared with the other two techniques. Using the two fixed redundancy techniques, the average CPI first reduces then increases as the fault rate increases. This is because the fixed redundancy techniques utilize redundancy rigidly neglecting the changing demands of fault tolerance and high performance. When the fault rate is low, the execution results are easily confirmed. However, the fixed redundancy schemes still utilize the same amount of redundancy for fault tolerance, thereby wasting resources that could otherwise be used for enhancing the performance. As the fault rate starts to increase, some speculative executions are corrupted and the incorrect paths are pruned. This effectively releases some resources for performance improvement. As a result, we observe a reduction in average CPI when the fault rate increases from 0 to around 0.25 in fixed redundancy allocation schemes. When the fault rate continues to increase, the fixed redundancy techniques show disadvantages manifested as the performance slowdown. While the executions are corrupted at a higher rate, these schemes only provide a fixed amount of redundancy for each allocation, which is unlikely to be sufficient to confirm the instructions quickly. As a result, although some efforts can be saved from the pruned incorrect branches, the overall performance can hardly be satisfactory due to the lagging instruction confirmation and frequent re-execution for fault recovery.

5.4.3 Performance Predictability

Performance predictability is an important metric, especially for real-time applications which typically require stable and predictable system performance. Similar to the simulations described above, we study the standard deviation of CPI under different fault rates. The results are shown in Fig. 5.6. As indicated, the performance of the fixed redundancy techniques shows large variations. When the fault rate is around 0.5, the CPI deviation is more than 12%. In contrast, the proposed dynamic redundancy technique significantly reduces the CPI deviation, e.g., only at 1.5% when the fault rate is 0.5, and almost zero when the fault rate is smaller.

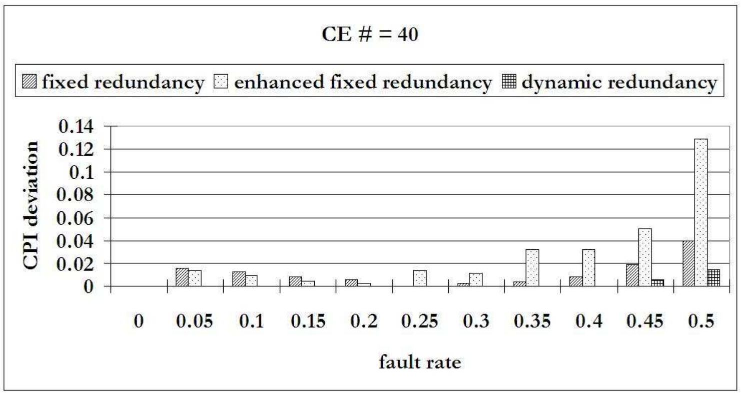


Figure 5.6: Comparison of CPI deviation.

It can be observed that the proposed technique is able to deliver stable performance over a range of fault rates. The reason is that dynamic redundancy allocation is flexible and can allocate more redundancy to the precedent instructions, thereby not only preventing resources from being depleted by over-aggressive speculative execution but also accelerating the confirmation of precedent instructions. This is important to reduce re-execution, especially when the fault rate is high. Therefore, the proposed technique enables high performance with a predictable manner. This is also beneficial for synchronizing with other nanocomputing subsystems.

5.4.4 Scalability

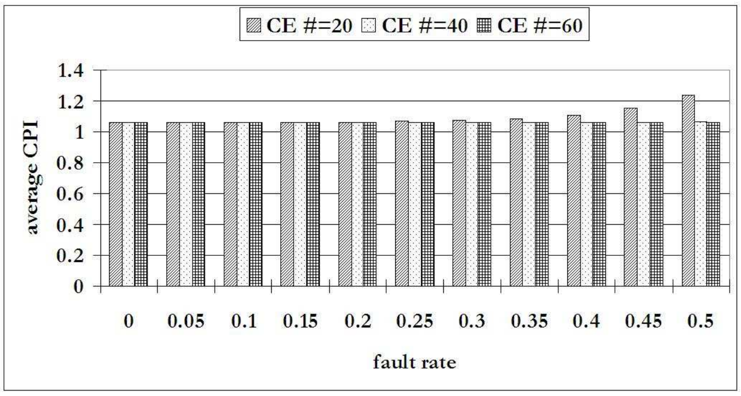


Figure 5.7: The impact of available CEs on average CPI.

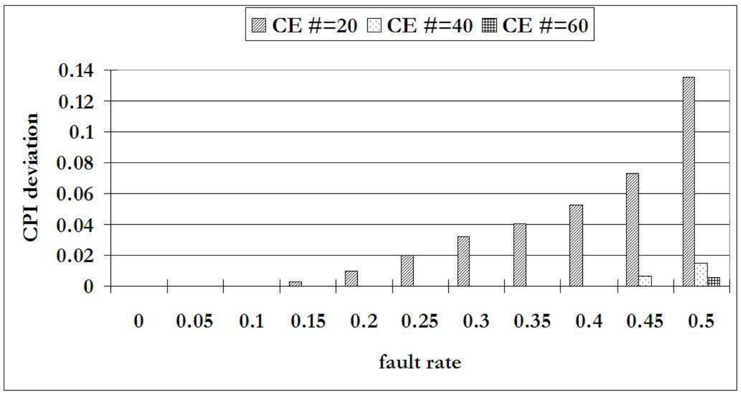


Figure 5.8: The impact of available CEs on CPI deviation.

We also study the scalability of the proposed technique. Figures 5.7 and 5.8 show the results of average CPI and the deviations with different numbers of CEs under a range of fault rates. We can easily see that the average performance of the proposed technique is very stable for different numbers of CEs when the fault rate is below 0.3. When the system is experiencing a rather high fault rate at around 0.5, the dynamic redundancy allocation can still achieve an average CPI at around 1.2 with only 20 CEs. This result is even better than the fixed redundancy techniques with 40 CEs (see Fig. 5.5). Thus, the proposed technique can reduce the hardware resources needed for fault tolerance and performance. In addition, the dynamic redundancy allocation maintains performance predictability at different numbers of CEs, as shown in Fig. 5.8. Thus, the results in Fig. 5.7 and 5.8 justify the good scalability of the proposed technique.

Chapter 6

Defect-Insensitive Signal Processing

6.1 Introduction

As discussed earlier, it is important to achieve reliable computation out of defect-prone devices. In chapter 5, we have investigated solutions for nanocomputing systems. In this chapter, we address the defect tolerance in nanoelectronic implementation of DSP systems by developing a new algorithmic design framework enabling *defect-insensitive* signal processing.

The underlying idea is based on the fact that most DSP applications do not require absolute correctness in hardware operations. An example is frequency-selective filtering where the input signals always contain a significant amount of channel noise and out-of-band interferences, which lead to signaling errors at the output. The reliable performance of these DSP applications is typically determined by a desired level of signal-to-nose ratio (SNR) or bit-error rate (BER), whereas the absolute correctness on a per-symbol basis is not necessary. Exploiting this fact, we conduct a theoretical analysis on algorithmic enhancements for defect-insensitive signal processing with unreliable nanoelectronics. It is worth notice that in [56], [57], algorithmic approaches are developed to jointly improve energy efficiency and soft-error tolerance by minimizing energy overhead while satisfying performance constraint specified by

SNR. Different from that work, we aim at cost-efficient and low-complexity solution for addressing the excessive permanent defects in nanoelectronics. We also develop a heuristic design approach to achieve reliable signal processing while reducing the complexity and cost related to post-fabrication testing and defect diagnosis. Simulation results on crossbar-based finite impulse response (FIR) filters demonstrate the effectiveness and scalability of our approach under a large range of defect rates.

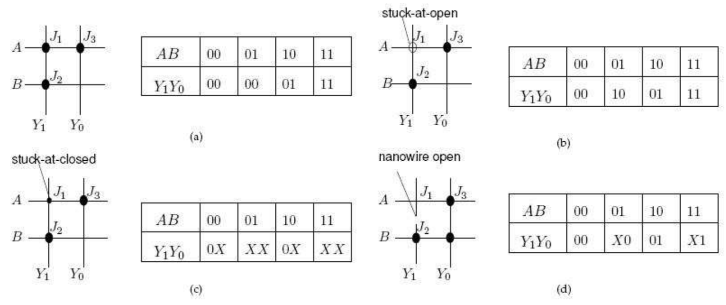
This chapter is organized as follows. In section 6.2, we discuss the major defects in nanowire crossbars and their impact on system-level performance of DSP applications. In section 6.3, we propose the algorithmic design framework for defect-insensitive signal processing. Simulation results of frequency selective filtering are provided in section 6.4 for demonstration of the proposed approach.

6.2 Defects in Crossbar-Based Nanoelectronics

In this section, we will show how excessive defects in crossbar-based nanoelectronics can affect signal processing systems at the algorithm level.

Most defects from the self-assembly fabrication of nanoelectronics can be placed into three broad categories [58]: (1) *crosspoint stuck-at-open*, caused by a missing switch at the crosspoint, (2) *crosspoint stuck-at-closed*, where two orthogonal nanowires are shorted together at the crosspoint, and (3) *nanowire open*, indicating a broken nanowire.

Due to the different physical mechanisms, these defects may manifest as various logic errors at the circuit level. Consider an AND gate array shown in Fig. 6.1 (the associated pull-up and pull-down arrays [101] are omitted for simplicity). If the crosspoint J_1 has a stuck-at-open fault, the output column Y_1 cannot be pulled down by the input due to the missing switch. Thus, the output Y_1 will give an incorrect result when $AB = 01$ (see Fig. 6.1(b)). On the other hand, stuck-at-closed and nanowire open defects generate undermined output values (see Fig. 6.1(c) and (d)), e.g., the output can be read as either logic "1" or "0" depending on the resistance ratio and the receiving circuits. An important observation here is that, while the defects are permanent, the induced logic errors may not be constant but show strong dependency on the input. For most digital logic where the input can be modeled as a random signal, the defect-induced errors behave more or less randomly.



AB	00	01	10	11
Y_1Y_0	00	00	01	11

AB	00	01	10	11
Y_1Y_0	00	10	01	11

AB	00	01	10	11
Y_1Y_0	$0X$	XX	$0X$	XX

AB	00	01	10	11
Y_1Y_0	00	$X0$	01	$X1$

Figure 6.1: Defects in nanowire crossbars: (a) defect-free implementation and (b)-(d) three major types of defects.

It is expected that in a signal processing integrated system containing complex logic networks, the defect-induced errors will show extremely complicated relationship with the input statistics, circuit topology, and algorithm specifications. Consider a crossbar-based implementation of a low-pass FIR filter with 40-taps and bandwidth $w_l = 0.3\pi$. A wideband signal is applied as the input to this filter. For the purpose of illustration, we randomly choose defective crosspoints with defect rates of 0.01 and 0.1, where defects are assumed to be uniformly distributed. Figures 6.2(a) and (b) show the defect-induced errors measured at the filter output under the two defect rates. Apparently, the magnitude of defect-induced errors increases with the defect rate. In addition, the output errors show strong randomness, which is consistent with our discussion regarding Fig. 6.1.

Dealing with defect-induced errors at device and circuit levels requires exhaustive testing and diagnosis on a per-chip basis. In general, this approach is very time-consuming and becomes less effective as the integration complexity scales up. On the other hand, defect-induced errors behave like an additive noise-like signal at the system level. Since digital filters are designed to mitigate signaling noise, it is possible to deliberately compensate for the defect-induced errors through algorithm enhancements, as discussed in the next section.

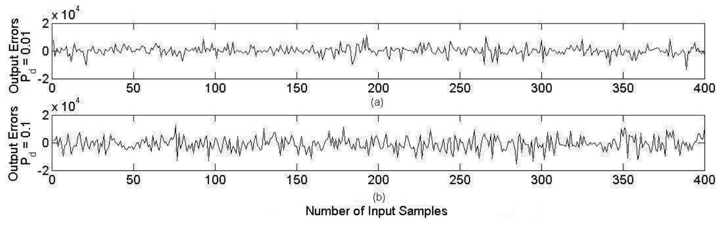


Figure 6.2: Defect-induced errors at a filter output different defect rate ($P_d = 0.01,\ P_d = 0.1$).

6.3 Defect-Insensitive Signal Processing with Nanoelectronics

In this section, we propose an algorithmic design framework enabling new opportunities of defect-insensitive signal processing using unreliable nanoelectronics.

Different from general-purpose computing, signal processing applications such as digital filtering usually do not require absolute correctness in the symbol-by-symbol execution. In fact, channel noise and out-of-band interferences inevitably generate residual signaling noise at the system output. Thus, the reliable performance of signal processing is usually measured by a desired level of signal-to-nose ratio (SNR). This opens up an opportunity to developing algorithmic enhancements that not only ensure the overall reliable signal processing performance but also reduce the complexity of defect tolerance.

In what follows, we will use frequency selective filtering to illustrate our approach of defect-insensitive signal processing. Frequency selective filtering is a basic signal processing module in various DSP systems. Conventional algorithm design concerns with a performance specification given in terms of the output SNR by assuming the availability of error-free hardware substrates. As shown in Fig. 6.3(a), the input of an ideal defect-free filter $H_1(e^{j\omega})$ includes signal $x(n)$ and noise $\eta(n)$, and the output contains the desired signal $y_1(n)$ and residual noise $\eta_1(n)$. The SNR of this filter can be calculated as

$$\mathrm{SNR}_1 = 10\log_{10}\left(\frac{\sigma_{y_1}^2}{\sigma_{\eta_1}^2}\right), \tag{6.1}$$

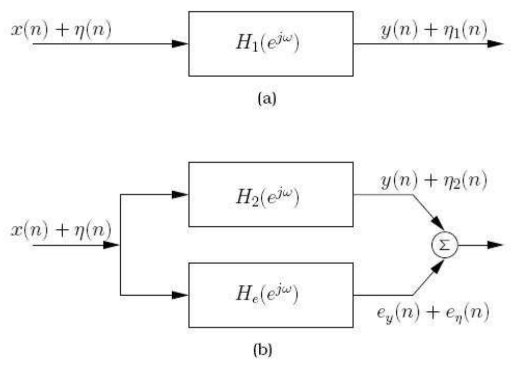


Figure 6.3: Modeling of (a) ideal defect-free filter in direct implementation and (b) defect-insensitive filter.

where $\sigma_{y_1}^2$ and $\sigma_{\eta_1}^2$ are the variances (energy) of the signal and noise at the output, respectively.

Due to excessive defects in nanoelectronics, the output will contain defect-induced noise-like errors. Thus, the SNR might be reduced below the required level. Conventional techniques on defect tolerance rely upon per-chip based testing to locate all the defective crosspoints. In this chapter, we propose a new approach where algorithm design explicitly addresses the defect-induced errors at the system level so that defect tolerance becomes self-enabled. Specifically, we look for an over-designed filter $H_2(e^{j\omega})$ that deliberately introduces extra performance margins for the purpose of offsetting the defect-induced errors. A defect-prone implementation can be modeled as the sum in frequency response of $H_2(e^{j\omega})$ with another "filter" $H_e(e^{j\omega})$ as shown in Fig. 2(b), where $H_e(e^{j\omega})$ actually reflects the statistics of defect-induced errors in $H_2(e^{j\omega})$. For a given implementation of $H_2(e^{j\omega})$, the defect-induced errors are input-dependent and thus can be modeled as the output of $H_e(e^{j\omega})$. The goal is to design $H_2(e^{j\omega})$ such that the output SNR in the presence of defect-induced errors (i.e., output of $H_e(e^{j\omega})$) will be at least equal to (6.1) to match the ideal defect-free filter $H_1(e^{j\omega})$.

To meet the performance requirement, the proposed defect-insensitive filter should satisfy

$$\mathrm{SNR}_2 = 10\log_{10}\left(\frac{\sigma_{y2}^2}{\sigma_{\eta_2}^2 + \sigma_{e_y}^2 + \sigma_{e_\eta}^2}\right) \geq \mathrm{SNR}_1. \tag{6.2}$$

where $\sigma_{\eta_2}^2 < \sigma_{\eta_1}^2$ as $H_2(e^{j\omega})$ is over-designed, and $\sigma_{e_y}^2$ and $\sigma_{e_\eta}^2$ are the variances of defect-induced output errors depending upon the inputs $x(n)$ and $\eta(n)$, respectively.

Because $H_1(e^{j\omega})$ and $H_2(e^{j\omega})$ have the same input signal component $x(n)$, we can expect $\sigma_{y_2}^2 \approx \sigma_{y_1}^2$, i.e., the output signal components of the two filters are also nearly equal. Comparing (6.1) and (6.2), defect-insensitive signal processing can be achieved if the following condition is satisfied

$$\sigma_{e_y}^2 + \sigma_{e_\eta}^2 \leq \sigma_{\eta_1}^2 - \sigma_{\eta_2}^2. \tag{6.3}$$

Assume that the input noise $\eta(n)$ is zero-mean with variance σ_η^2. The RHS of (6.3) can be obtained as [59]

$$\sigma_{\eta_1}^2 - \sigma_{\eta_2}^2 = \sigma_\eta^2\left(\sum_{k=0}^{N_1-1} h_1^2(k) - \sum_{k=0}^{N_2-1} h_2^2(k)\right), \tag{6.4}$$

where N_1 and N_2 are the numbers of taps of $H_1(e^{j\omega})$ and $H_2(e^{j\omega})$, respectively.

Denote the autocorrelation of input signal $x(n)$ as $\phi_{xx}(l) \triangleq E(x(n)x(n-l))$. The LHS of (6.3) can be obtained as

$$\sigma_{e_y}^2 + \sigma_{e_\eta}^2 = (\sigma_x^2 + \sigma_\eta^2)\sum_{k=0}^{N_2-1} h_e^2(k) + 2\sum_{l=1}^{N_2-1} \phi_{xx}(l)c_{h_e h_e}(l), \tag{6.5}$$

where

$$c_{h_e h_e}(l) = \sum_{k=0}^{N_2-1-l} h_e(k)h_e(l+k), \forall l \geq 0. \tag{6.6}$$

Note that $h_e(k) \neq 0$ when the k^{th} tap in $H_2(e^{j\omega})$ contain defects. Substituting (6.4)−(6.6) back into (6.3), we can verify whether the defect-insensitive design could achieve the specified performance in terms of the output SNR. Furthermore, due to the extra performance margin, we can allow some taps in $H_2(e^{j\omega})$ to bear defects while still satisfying the specified output SNR. Actually, only a few taps need to be configured as defect-free (see the heuristic design algorithm given below). As shown in section 6.3, the proposed defect-insensitive design is able to reduce the complexity and cost related to per-chip testing while achieving reliable signal processing. In addition, the hardware overheads in $H_2(e^{j\omega})$ are manageable as nanoelectronics provide abundant device resources and redundancy. Note that a closed-form expression of $H_e(e^{j\omega})$ is not necessary in the proposed heuristic algorithm design of defect-insensitive signal processing, as summarized next.

(i) To provide performance margin in $H_2(e^{j\omega})$ for defect tolerance, (6.4) should be positive and large enough. The design procedure can start from a relatively conservative design $H_2(e^{j\omega})$ that provides a small margin, and then check the number of taps needed to be defect-free under a certain defect rate and distribution, as explained in the following steps.

(ii) For a given $H_2(e^{j\omega})$, the defect-free taps should be those with large algorithmic weights. As $\phi_{xx}(l)$ in (6.5) is a decreasing function of l for $l > 0$ and $C_{h_e h_e}(l)$'s may cancel out from each other, it is reasonable to assume that the first term in the RHS of (6.5) is a dominant factor when offsetting defect-induced output errors. Therefore, we will primarily consider the first term and use the second term only for verification. This leads to the following adjustments.

(iii) We first group the taps $h_2(k)$'s of $H_2(e^{j\omega})$ based on their effective bit-width M_k's. If $h_2(k)$ has a bit-width M_k, the defect-induced errors from this tap is proportional to $h_e^2(k)$ (see (6.5)) with a maximum value of $\max\{h_2^2(k), (2^{M_k} - 1 - |h_2(k)|)^2\}$, which is less than 2^{2M_k}. It is easy to see that for $h_2(k)$ with large bit-width M_k (i.e., large coefficient value), $h_e^2(k)$ is also large thereby resulting in large defect-induced output errors. These taps have large algorithmic weights and thus need to be configured as defect-free (through testing and replacing the defective crosspoints). Thus, design effort needs to be directed towards ensuring defect-free computation in taps with large coefficients only. This is reasonable because these taps contribute more to the output. We can start with protecting the coefficient with the largest value and continue until (6.2) is just met. Note that this algorithm design step can be performed through simulation under certain defect rates and distributions that match the actual implementation. Eventually we can determine a few taps that need to be defect-free. Thus, the complexity and cost related to post-fabrication testing and diagnosis are reduced significantly .

(iv) Repeating the above steps until an optimal $H_2(e^{j\omega})$ is obtained. The optimal $H_2^{opt}(e^{j\omega})$ is the one that can afford the minimum number of defect-free taps while still satisfying the performance requirement (6.3). This optimal $H_2^{opt}(e^{j\omega})$ is the final algorithm design of defect-insensitive signal processing.

The above design procedure is summarized in Algorithm 4 Note that this design procedure indicates that defect-induced errors should only be allowed from the taps in $H_2(e^{j\omega})$ with smaller coefficients. This, however, does not imply that these taps have no algorithmic contribution at all and thus can be removed from the implementation. It only shows that the defect-induced errors in these taps have a minor impact on the algorithmic performance (e.g., the induced SNR degradation is acceptable). Furthermore, this design procedure does not depend on specific defect distribution in the implementation (i.e., clustered or non-clustered). As long as the performance requirement (6.3) can be satisfied after applying the design procedure, the system is able to tolerate the existing defects.

6.4 Evaluation and Discussion

In this section, we evaluate the proposed algorithm design methodology for defect-insensitive signal processing.

Figure 6.4 shows the block diagram of the defect-insensitive filter. The spectrum of the input signal $x(n)$ consists of a primary signal $x_1(n)$ occupying the $[0, \omega_l]$ and a bandpass signal $x_2(n)$ in $[\omega_l + 0.1\pi, \pi]$. A white Gaussian noise is added into the input signal. The performance requirement of the filter is to extract the primary signal $x_1(n)$ with 22dB output SNR, i.e., $\text{SNR}_{spec} = 22\text{dB}$. For defect-insensitive implementation, we design the filter with 24dB output SNR. This performance margin is then utilized to adjust the configuration of defect-bearing taps. All the filters are designed using the Parks-McClellan method [60].

We simulate the defect-induced errors by adding defects at crosspoints with a

Algorithm 4: The general algorithm of design procedure.

```
Input:
  • $\omega_{p,i}$, $\delta_{p,i}$, $\omega_{s,i}$, $\delta_{s,i}$ (filter design specification)
  • $\sigma_\eta$, $\phi_{xx}$ (noise and signal statistics in typical applications)
  • $N_{max}$ (maximum filter order affordable)
Output:
  • $\{h_2(k)^{opt}\}_{k=0}^{N_2^{opt}}$ (the optimal defect-tolerant filter)
  • $\{T(k)\}_{k=0}^{N_2^{opt}}$ (indicating whether $h_2(k)$'s must be defect-free)
begin
    design $\{h_1(k)\}_{k=0}^{N_1}$ according to specification
    calculate $\sigma^2_{\eta_1}$
    for $N_2 \leftarrow N_1 + 1$ to $N_{max}$ do
        design $\{h_2(k)\}_{k=0}^{N_2}$ subject to $\sigma^2_{\eta_1} - \sigma^2_{\eta_2} > 0$
        group $h_2(k)$'s according to $M_k \rightarrow G_1 = \{h_2(k_1^1), h_2(k_1^2), ...\}$,
        $G_2 = \{h_2(k_2^1), h_2(k_2^2), ...\}$,
        ......
        $G_{max} = \{h_2(k_{max}^1), h_2(k_{max}^2), ...\}$
        for $k \leftarrow 0$ to $N_2$ do
            $T(k) = 1$ (start with a filter with only defect-free taps)
        $\sigma^2_{e_y} + \sigma^2_{e_\eta} = 0$
        $j = 0$
        repeat
            $j = j + 1$
            allow taps in group $G_j$ to be defect-bearing by letting $T(k_j^i) = 0$
            calculate $\sigma^2_{e_y} + \sigma^2_{e_\eta}$
        until $\sigma^2_{e_y} + \sigma^2_{e_\eta} < \sigma^2_{\eta_1} - \sigma^2_{\eta_2}$
        record $S_T(N_2) = \sum_{k=0}^{N_2-1} T(k)$, the number of defect-free taps
    determine the optimal design by finding the minimum $S_T(N_2)$
end
```

range of defect rates. In addition, we assume that the defects are uniformly distributed across the nanowire crossbar. For stuck-at-closed and nanowire open defects, we assume the affected outputs are fixed at either logic "1" or "0" with equal

Table 6.1: Results of defect rates, bandwidth, and the number of taps needed to be defect-free in defect-insensitive filter implementation.

			Defect Rate									
Bandwidth ($\times\pi$)	N_{spec}	$N_{over-designed}$	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.1
0.3	40	45	23	24	26	29	30	30	31	32	32	33
0.4	40	45	19	19	20	22	24	25	27	28	29	30
0.5	40	45	17	18	22	25	26	26	27	27	28	29
0.6	40	45	16	17	17	18	20	21	21	23	23	24

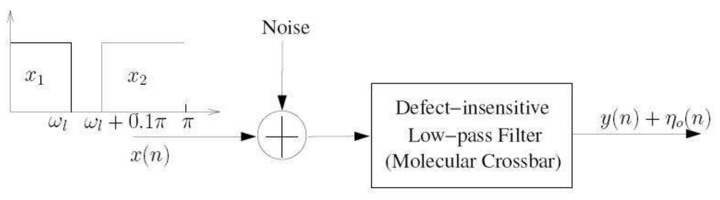


Figure 6.4: The low-pass FIR filter for simulation.

probabilities. In order to evaluate the effectiveness of the proposed technique, we vary the bandwidth ω_l from 0.3π to 0.7π. Table 6.1 shows the relationship of filter bandwidths, defect rates, and the configuration of defect-insensitive filters. Note that the values under the defect rates are the number of taps needed to be defect-free when the output SNR of defect-insensitive filters degrades to 22dB.

For a given filter bandwidth, the number of defect-free taps needed in the defect-insensitive filter increases with the defect rate. This is expected as more crosspoints could become defective and thus potentially contribute to the increase in defect-induced errors at the output. However, the proposed defect-insensitive design significantly relaxes the need for post-fabrication defect testing and diagnosis. For example, at $\omega_l = 0.3\pi$, the number of taps needed to be defect-free increases from only 23 to 33 as defect rate increases from 1% to 10%, while in the direct implementation all of the 40 taps need to be defect-free. This reflects 40% to 19% savings in defect tolerance. As shown in Table 6.1, this benefit is consistent (thus scalable) across a wide range of bandwidths and defect rates.

Another important observation is that the proposed technique achieves better trade-offs of defect tolerance as the filter bandwidth increases, i.e., our technique is well-suited for broadband signal processing. The reason is that as the bandwidth increases, fewer taps will have coefficients with large magnitude. As explained in section 6.3, the taps with small coefficients can bear defects without degrading output SNR beyond the design target. Again, this does not imply that these taps have no algorithmic contribution and thus can be removed from the implementation. It only shows that the defect-induced errors in these taps have a minor impact on the algorithmic performance. As expected, the over-designed defect-insensitive filters introduce some hardware overheads in terms of more filter taps. However, these overheads are practically acceptable as nanoscale integration offers abundant device resources and redundancy.

Chapter 7

Power Equilibrium for Register File Security

7.1 Introduction

Recently, security has become an important issue in the design of reliable computer systems. The security of cryptographic algorithms such as block ciphers and public-key algorithms relies on the secrecy of the key.

Traditionally, cryptanalysists break a cryptographic algorithm by finding the weakness in the algorithm that can be exploited with a complexity less than brute force. However, a real computing device not only performs tasks specified by the cryptographic algorithm but also inevitably produces some other information such as timing, power, and electromagnetic leak. These types of information, called side-channel information, can be exploited in *side-channel attacks* to retrieve secret keys much more easily. Side-channel attacks have successfully broken many algorithms that are secure under traditional attacks [64, 65]. Mobile devices and sensor nodes that work in the field, not protected by physical security mechanisms, are more vulnerable to side-channel attacks.

Among all side channel attacks, *power analysis*, which exploits the power consumption of a cryptographic system, can be carried out easily. It is very effective in breaking cryptographic algorithms [64, 65]. In this type of attacks, adversaries learn what operations are performed and what data are processed by analyzing the power traces of computations. They can then figure out part or all of the bits in the secret key.

A lot of work has been done on the countermeasures against power analysis attacks. Many countermeasures studied software implementations, trying to make the power consumption of a crypto-system either random or identical for different keys [72]. The software countermeasures usually work only for specific algorithms and have large performance overhead. Very often, the countermeasures are found vulnerable to more advanced attacks..

There are some hardware countermeasures on the circuit level [73, 69, 70]. The use of self-timed dual-rail logics is proposed in [69] to provide protection to power analysis attacks. Dynamic and differential logic is also employed in [73]. Both the logic styles can make the power consumption of logic gates independent of the data values. One of the drawbacks of these methods is large area and power overhead. The method described in [70] compensates the power consumption of the system with voltage and frequency scaling techniques and an analog current injection circuit. However, frequency scaling affects the performance of software and may make the system vulnerable to timing attacks, another type of side-channel attacks. It should be pointed out that memory security is not the primary focus of these above techniques.

Some other hardware countermeasures are proposed at the architectural level [67, 68]. They randomize either the register renaming [67] or the issue of instructions in the instruction window [68] to make the power analysis attacks more difficult. However, these methods may not fit well with low-end processors, which typically do not have the register renaming mechanism or large instruction window to support out-of-order execution.

Despite the previous work, it is still desirable to study an effective, low-cost, algorithm-independent countermeasure for general-purpose processors, especially for low-end processors used in mobile devices or sensor nodes. Considering the fact that memory and register files contribute a significant part to the power consumption and hence are vulnerable to many power attacks [61, 71], we study how to make register files resistant to power analysis attacks in this work. Although register files are only one of many possible targets of power analysis attacks, it is expected that the work on register files will help identify the effective mechanisms that mitigate power analysis attacks and hence can be extended to other components in the system as well.

Different from existing countermeasures, we propose a new register file architecture, referred to as the *Secured Power-Equilibrium Register File (SPERF)*, which is an algorithm-independent countermeasure. In SPERF, the original data and its redundant flipped copy are maintained in two register banks to achieve an equilibrium state of power consumption, where power is always balanced and independent of data values. Thus, security information is withhold from power leaks. Some preliminary work is reported in [29]. In this chapter, we extend our past work by making the following contributions. First, the design space of SPERF is exploited and tradeoffs between different requirements are investigated. Second, architectural supports including a register allocation algorithm are developed to facilitate the hardware/software co-design addressing tradeoffs in SPERF. Third, a comprehensive suite of simulations and board experiments is performed covering the resistance to power analysis attacks, energy overhead, and performance-area tradeoffs.

The organization of this chapter is as follows. Section 7.2 studies the vulnerability of conventional register file to power analysis attacks. Section 7.3 presents the proposed technique and why it can protect register file from power analysis attacks. Section 7.4 exploits the design space and section 7.5 evaluates our method with simulations and board experiments.

7.2 Vulnerability of Register Files to Power Analysis Attacks

On-chip memory is a major source of power information leakage that can be exploited in power analysis attacks. In particular, the register file, the most frequently accessed memory, contributes a significant part to the power traces of computation.

Figure 7.1 shows the generic architecture of a conventional register file. For illustration purpose we consider the register file to comprise of 32 registers with 32 bits

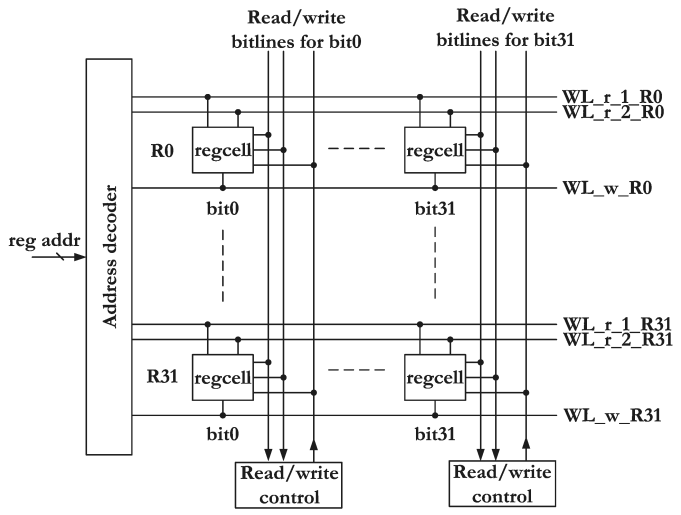


Figure 7.1: An example of a 32 × 32 register file.

in each. The address decoder selects the target register cells according to the register address and asserts the corresponding wordlines. Each register cell is an SRAM cell with multiple read/write ports [63] (2r/1w in this example). During a read operation, the wordline $WL\_r\_1\_Rx$ or $WL\_r\_2\_Rx$ is asserted, and the data stored in the selected register Rx are evaluated by the read bitlines. Upon a write operation, data can be written from the write bitlines into the register Ry selected by WL_w_Ry. Typically, read logic is implemented in dynamic CMOS circuits that are precharged in the first half of the clock cycle and evaluated and sensed in the second half of the cycle, whereas write logic is in static circuits that are activated in the first half of the cycle. Thus, this register file supports back-to-back read and write operations with single-cycle latency.

While the average power consumption of a register file is usually small, its frequent activities can generate substantial transient responses that contribute to data-dependent power traces. To illustrate, we consider the power consumption on register bitlines, a dominant component in the power profile of register file as bitlines typically have much larger parasitic capacitance than bit cells. We will also consider capacitive power consumption due to the high switching activities and the small size of register files where the leakage power is relatively small.

We investigate how power consumption reveals data values in register file based on switching distance model [66] and Hamming distance model [65]. A data of m-bit is coded as $X = \sum_{i=0}^{m-1} x_i 2^i$, where $x_i = 0$ or 1. Its Hamming weight is the number of 1's, i.e., $H(X) = \sum_{i=0}^{m-1} x_i$.

We assume that power consumption is a linear function of $0 \rightarrow 1$ and $1 \rightarrow 0$ switches. This simplification, although has its limitation, allows us to focus on the major source of power leaks and identify the vulnerability of register file to power analysis attacks. Generally, the two types of switches consume different amount of power. We denote $\delta = \frac{P_{0\rightarrow 1} - P_{1\rightarrow 0}}{P_{0\rightarrow 1}}$ as the normalized difference of power consumption for each switch between the two types, where $P_{0\rightarrow 1}$ and $P_{1\rightarrow 0}$ are the power consumed in a single $0 \rightarrow 1$ switch and a single $1 \rightarrow 0$ switch, respectively. Thus, power consumption when data is changing from X to Y can be calculated as

$$\begin{aligned} P(X,Y) &= a[N_{0\rightarrow 1}(X,Y) + (1-\delta)N_{1\rightarrow 0}(X,Y)] \\ &\quad +b \\ &= a[H(\overline{X}Y) + (1-\delta)H(X\overline{Y})] + b, \end{aligned} \tag{7.1}$$

where $N_{0\rightarrow 1}(X,Y)$ and $N_{1\rightarrow 0}(X,Y)$ are the numbers of the two types of switches, respectively, during the X-to-Y transition, a is the scalar gain, and b is the static power that is independent of the switching. Notation $\overline{X}$ refers to the complement of X.

During a register read, all read bitlines are precharged to the supply voltage V_{dd} in the first half of a clock cycle. This indicates a data transition from X^r_{old} (the data on the read bitlines due to the previous read operation) to All_1 (m bits of '1'). Thus, power consumption can be modeled as

$$\begin{aligned} P^{precharge}_{read} &= P(X^r_{old}, All\_1) \\ &= a_r[m - H(X^r_{old})] + b_r, \end{aligned} \tag{7.2}$$

where a_r and b_r are the scalar gain and static power component in read circuitry, respectively.

During the second half of the cycle, some read bitlines will be discharged if 0's (or 1's depending on the read logic) are stored in the corresponding bit cells. The power during this evaluation phase where X^r_{new} is read out can be calculated as

$$\begin{aligned} P^{evaluate}_{read} &= P(All\_1, X^r_{new}) \\ &= a_r(1-\delta)(m - H(X^r_{new})) + b_r. \end{aligned} \tag{7.3}$$

From (7.2) and (7.3), power consumption during both phases of read operation is data-dependent.

When the register is written in a new data X^w_{new} to replace the currently stored old data X^w_{old}, the power consumption can be calculated as

$$\begin{aligned} P_{write} &= P(X^w_{old}, X^w_{new}) \\ &= a_w[H(\overline{X^w_{old}}X^w_{new}) + (1-\delta)H(X^w_{old}\overline{X^w_{new}})] \\ &\quad +b_w, \end{aligned} \tag{7.4}$$

where a_w and b_w are the corresponding a and b in write circuitry. Again, the power consumption in write operation is also data-dependent. This state of power consumption for different data is the source of power leaks that are exploited by power analysis attacks.

It is worth to notice that read and write operations induce different data-dependent patterns on power traces, as shown in (7.4) compared with (7.2), (7.3). because write logic is typically implemented in static circuits without a precharge phase. Therefore, hardware countermeasures against power analysis attacks need to take into account this effect.

7.3 Secured Power-Equilibrium Register File

As explained in section 7.2, conventional register files are vulnerable to power analysis attacks as the power consumption is highly data-dependent. In this section, we propose our method to make register files resistant to this kind of attacks by achieving an equilibrium state of power consumption.

7.3.1 Basic Architecture

As power consumption is determined by the patterns of $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions, one approach to protect register file from power analysis is to make the number of transitions balanced and independent of the data values accessed during read and write operations. We propose to exploit this idea by introducing redundant circuitry and operations in the register file so that each register access will always incur the same number of transitions on the read and write bitlines. Specifically, data are stored at two locations in the register file. One location has the normal bit representation of the value and the other has the complemented bits. Writing to a register updates the bits at both locations. Reading from the register file also retrieves values from both locations. This new register file design is referred to as the Secured Power-Equilibrium Register File ($SPERF$).

We consider a 32×32 original register file for the purpose of illustration. SPERF is implemented by hardware to introduce R_h redundant registers (*hardware-enabled redundancy*) and/or by software to release R_s redundant registers (*software-enabled redundancy*) and then reorganizing the register file into two banks as shown in Fig. 7.2. Both the original data bank and the redundant data bank have N_s registers. Details of the hardware/software co-design and its tradeoffs are discussed in section 7.4.

We define two operation modes for SPERF, the normal mode and the secure mode. Redundant operations are enabled only when the system is in the secure mode. In the normal mode, SPERF works in the same way as a conventional register file. The original data bank and part of the redundant data bank are both used for the original *read_data* and *write_data*. In the secure mode ($\overline{secure\_mode} = 0$), the redundant bank performs write and read operations on flipped copies of the data. On a write, *write_data* and its flipped bits are simultaneously stored in the original and redundant banks, respectively. On a read, both banks are accessed. The original and flipped *read_data* are read out from the register file, where the corresponding read bitlines are evaluated by these data. Both the original data and the flipped copy are written to the pipeline registers, which are placed between pipeline stages, although only the original data is used in the functional units for further computations. Thus, the capacitive load of the two read ports can match each other in order to avoid power information leakage. Note that the flipped copy could potentially be used in logic operations in order to enhance the security of ALU and other processor components as well, which may lead to a comprehensive solution to power analysis attacks.

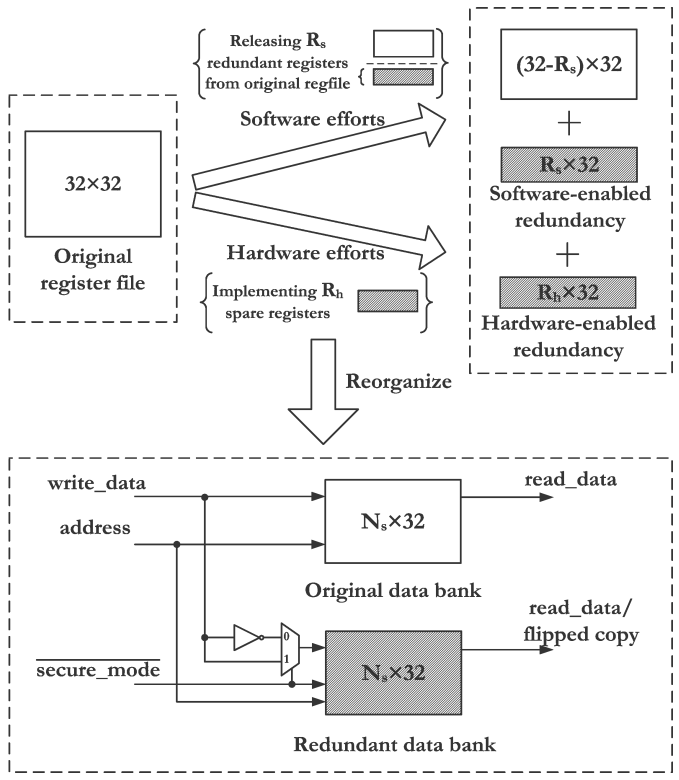


Figure 7.2: Secured Power-Equilibrium Register File. R_h entries of hardware-enabled redundancy and R_s entries of software-enabled redundancy are reorganized into a redundant data bank to achieve power equilibrium. N_s represents the number of architectural registers in the secure mode. ($N_s = 32 - R_s = R_h + R_s$.) Further details are discussed in section 7.4.

7.3.2 Power Equilibrium and Circuit Implementation

We now study the mechanisms and circuit implementation to achieve power equilibrium.

Read Operation

During a read operation, power consumption in register file is determined by the numbers of $1 \rightarrow 0$ transitions during the evaluate phase and $0 \rightarrow 1$ transitions during the precharge phase on read bitlines. It can be shown that in the secure mode, the balanced power consumption is always achieved.

Revisiting (7.2), (7.3) in section II, we can also derive the power consumption during read operation as follows. During precharge phase, bit lines in the original bank and the redundant bank experience transitions from X^r_{old} and $\overline{X^r_{old}}$, respectively, to All_1. Thus, power in precharge phase can be written as

$$\begin{aligned} P^{precharge}_{new\_read} &= P(X^r_{old}, All\_1) + P(\overline{X^r_{old}}, All\_1) \\ &= a_r[m - H(X^r_{old})] + b_r \\ &\quad + a_r[m - H(\overline{X^r_{old}})] + b_r \\ &= a_r m + 2b_r. \end{aligned} \tag{7.5}$$

Likewise, the power consumption in evaluation phase can be written as

$$\begin{aligned} P^{evaluate}_{new\_read} &= P(All\_1, X^r_{new}) + P(All\_1, \overline{X^r_{new}}) \\ &= a_r(1-\delta)[m - H(X^r_{new})] + b_r \\ &\quad + a_r(1-\delta)[m - H(\overline{X^r_{new}})] + b_r \\ &= a_r(1-\delta)m + 2b_r. \end{aligned} \tag{7.6}$$

(7.5) and (7.6) both show that a read access in the secure mode indeed results in a balanced and data-independent state of power consumption and thereby being protected from power analysis attacks.

Write Operation and Circuitry

Write logic in conventional register files is usually implemented in static circuits, leading to a different pattern of power traces as compared with read operations. Without a precharge phase, the number of bit transitions during a write operation depends not only on the value being written to the register but also on the value currently stored in the register. Simply writing a redundant copy does not defeat power analysis attacks. In fact, it exacerbates the power variance among different data values and makes it easier for adversaries to launch power analysis attacks.

Consider an example where Ry stores 32'h`0000_0000` and will be updated with a new value 32'h`0000_0001`. In the normal mode, there is only one $0 \rightarrow 1$ transition in the original bank. In the secure mode, in addition to this transition, there would be one more $1 \rightarrow 0$ transition in the redundant bank. In total, there would two bit transitions. If the original value in Ry is 32'h`FFFF_FFFF` instead, there would be a total of 62 ($= 2 \times 31$) transitions. The redundant bank actually could double the number of transitions, making it easier for adversaries to observe the power differences.

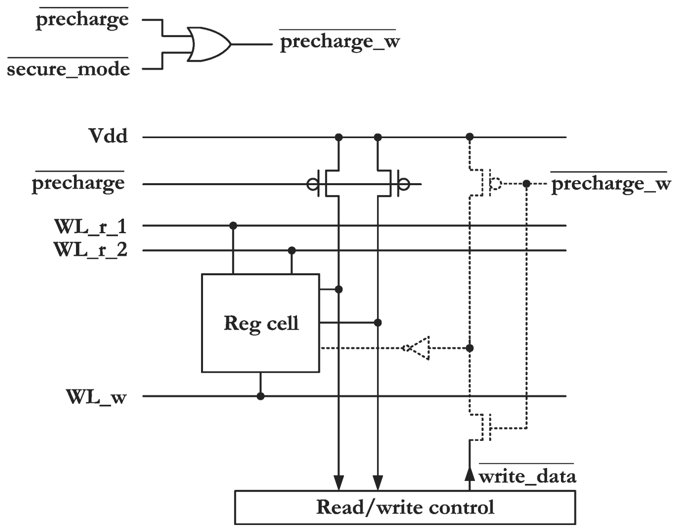

$\overline{\text{precharge}}$
$\overline{\text{secure\_mode}}$
$\overline{\text{precharge\_w}}$
Vdd
$\overline{\text{precharge}}$
$\overline{\text{precharge\_w}}$
WL_r_1
WL_r_2
Reg cell
WL_w
$\overline{\text{write\_data}}$
Read/write control

Figure 7.3: Modified write circuitry. (Dotted lines represent the modified write bit-lines. $\overline{secure\_mode} = 0$ indicates the secure mode.)

To address this issue, we need to modify the write operation so that its power consumption cannot be exploited. Inspired by the precharge in read operations, which makes the discharging power dependent only on the data value currently being read out, we add a precharge phase in write operations.

Figure 7.3 shows the modified write circuitry. In Fig. 7.3, a signal $\overline{precharge\_w}$, generated from $\overline{secure\_mode}$ and $\overline{precharge}$ (the precharge control in read operations), controls the precharge in write operations. In addition, an inverter is added to prevent the write bitline from being pulled down by a 0 in the cell. If the register file is in the normal mode ($\overline{secure\_mode} = 1$), the NMOS in dotted line is open while the PMOS in dotted line is closed. The register file works in the same way as a conventional implementation. If the register file is in the secure mode ($\overline{secure\_mode} = 0$), however, the write bitline will be precharged in the first half of the clock cycle by the PMOS, where 0 will be written into the register selected by the write wordline, replacing the old value temporarily. In the second half of the cycle, the PMOS is closed and the NMOS is open. Actual data will be written into the selected register.

During the precharge phase, data in the original bank and that in the redundant bank change from X^w_{old} and $\overline{X^w_{old}}$, respectively, to All_0. This causes power consumption expressed by

$$\begin{aligned} P^{precharge}_{new\_write} &= P(X^w_{old}, All\_0) + P(\overline{X^w_{old}}, All\_0) \\ &= a'_w(1-\delta)H(X^w_{old}) + b'_w \\ &\quad + a'_w(1-\delta)H(\overline{X^w_{old}}) + b'_w \\ &= a'_w(1-\delta)m + 2b'_w, \end{aligned} \tag{7.7}$$

where a'_w and b'_w are the scalar gain and static power in the proposed write circuitry.

During evaluation phase, new data X^w_{new} and its inverted copy $\overline{X^w_{new}}$ are actually written into the two banks, resulting power consumption by

$$\begin{aligned} P^{evaluate}_{new\_write} &= P(All\_0, X^w_{new}) + P(All\_0, \overline{X^w_{new}}) \\ &= a'_w H(X^w_{new}) + b'_w \\ &\quad + a'_w H(\overline{X^w_{new}}) + b'_w \\ &= a'_w m + 2b'_w. \end{aligned} \tag{7.8}$$

Thus, both phases in write operation achieve a balanced and data-independent state of power consumption.

Note that the additional precharge for write introduces energy overhead, but the overhead occurs only in the secure mode. In the normal mode, no precharge is performed and SPERF behaves as a conventional register file.

7.4 Exploiting Design Space for the New Register File

In this section, we investigate the design options of SPERF and study the compiler/operating system supports that are needed to exploit the design space.

7.4.1 Design Options

There are different ways to implement the original register bank and the redundant register bank. One design option [29] is to implement both the original register bank and the redundant register bank with the same size as in conventional register file. For example, if the number of architectural registers is 32, i.e., $N_s = 32$ in Fig. 7.2, the conventional register file comprises 32 registers with 32 bits in each, whereas this design option of SPERF introduces an additional 32×32 registers. Redundancy of this approach relies purely on hardware efforts ($R_h = 32, R_s = 0$) and has large area overhead ($2X$ compared with conventional register file). Thus, we refer to this design option as SPERF with *hardware-enabled redundancy.*

Alternatively, with software/compiler support (as discussed in section 7.4-B), the codes of security programs can be rewritten in a way that the number of architectural registers is cut to half as the one in the original codes, i.e., $N_s = 16$. Thus, some physical registers in the register file can be released by software efforts and serve as redundancy for the security enhancement. This is referred to as SPERF with *software-enabled redundancy.* For example, the security programs can be rewritten to have only 16 architectural registers, if the original ones have 32 architectural registers. Thus, no additional register entries need to be implemented and SPERF can be implemented with two 16×32 register banks ($R_h = 0, R_s = 16$).

In comparison with conventional register file, hardware-enabled redundancy effectively doubles the size of register file, whereas software-enabled redundancy does not introduce additional register entries, but comes at the cost of software efforts and likely runtime performance overhead for security programs due to the reduced architectural registers in the secure mode. Note that software-enabled redundancy does not lead to performance slowdown in the normal mode as both the original register bank and the redundant register bank are available for the non-security programs. Besides, software-enabled redundancy is more flexible and scalable compared with hardware-enabled redundancy.

Hardware-enabled redundancy and software-enabled redundancy both have their advantages and disadvantages. Hardware/software co-designs can be adopted using a combination of both forms of redundancy, so that they can complement and enhance each other for efficiency and scalability of security enhancement. Specifically, redundancy is created via not only hardware enhancements to introduce additional register entries but also software efforts to reduce the number of architectural registers for security programs. Careful tradeoffs among the conflicting requirements on performance and area will help to decide the optimized design. This is further studied with board experiments in section 7.5-C.

7.4.2 Hardware/Software Co-Design

To support hardware/software co-designs combining both hardware-enabled redundancy and software-enabled redundancy as discussed in section 7.4-A, compilation algorithms, especially register allocation algorithms, that are aware of the security enhancement are needed.

First, the optimal number of architectural registers that should be implemented (i.e., N_s) can be decided via simulation-based evaluation on the security programs under performance and area constraints. Then, the designers can compile the security programs accordingly. Note that programs to be running in the normal mode do not need this special support from compiler.

To compile the security programs, the available architectural registers in the secure mode (N_s) are assigned to variables at different program points. This process is called register allocation and is an important optimization in the compiler design. Most existing approaches for register allocation is based on the graph coloring framework [74]. By analyzing the live ranges of each variables in the program, the interference graph can be constructed. Thus, the register allocation problem is modeled as a graph coloring problem: K colors (representing K registers) are to be assigned to the nodes (representing variables) of the graph so that no two linked nodes (representing interfering variables) are of the same color. If graph coloring fails, some of the variables have to be spilled to memory.

A number of improvements have been achieved on coloring, spilling, coalescing, and static single assignment form. Due to the large overheads at compile time, especially the needs for constructing and reconstructing the interference graph, linear scan [75, 76] is proposed as an alternative to graph coloring. Compared with graph coloring, this algorithm scans the live ranges and allocates registers in a single pass, hence is simple and efficient. As reported in [76], linear scan does not necessarily sacrifice execution time performance.

In our work, we develop a heuristic register allocation algorithm to support hardware/software co-design to address the performance-overhead tradeoffs in SPERF design. This algorithm is summarized in Algorithm 5. Due to the simplicity and efficiency of linear scan, our algorithm is based on linear scan with a few improvements.

In step 1 shown in Algorithm 5, the intermediate representation of the code is preprocessed. When any variable is redefined (i.e., reassigned with a new value), the variable is renamed and becomes a new variable from the redefinition point. By doing so, the live interval can be shortened, thereby avoiding unnecessary interference with other variables. In step 2, the start point and the end point of each variable are recorded and the variables are ranked in increasing order of their start points. In step 3, registers allocated to expired variables are reclaimed. Then, register allocation is decided for each variable at its start point according to the following rules: If there is a free register available, it is allocated to the variable. Otherwise, the variable whose next use is furthest away is spilled. This is based on the observation that the variable staying alive but idle for a longer time has a higher possibility, hence higher performance penalties, of interfering with others. By spilling this variable, its live

Algorithm 5: The register allocation algorithm for supporting hardware/software co-designs of SPERF.

```
Input:
$IR$, the intermediate representation of the code
$Regs$, the set of available architectural registers determined by the performance-overhead tradeoffs
Output:
$MC$, the modified code
Algorithm:
1. Preprocess the $IR$
(a) linearize the code using a specific order (e.g., depth-first) and copy it to $MC$
(b) when the variables are redefined, rename them, make changes to $MC$
2. Build the list of variables, $Var$
(a) initialize: $Var := \{\}$
(b) build $Var$ and include live interval info.:
foreach $v_i$ in the code do
    scan and obtain the start point $t_{s\_i}$ (at the definition) and end point $t_{e\_i}$ (at the last use)
    $Var := Var + \{v_i[t_{s\_i}, t_{e\_i}]\}$
end
(c) sort $Var$ in increasing order of the $t_{s\_i}$'s
3. Register allocation and spilling
(a) initialize: $FreeRegs := Regs$, $AllocatedVar := \{\}$
(b) allocate physical registers:
foreach $t_{s\_i} \in Var$ (in increasing order) do
    i. release the expired variables in $AllocatedVar$:
    while $AllocatedVar$ not empty do
        forall $t_{e\_j} \in AllocatedVar$ do
            if $t_{s\_i} > t_{e\_j}$ then
                $AllocatedVar := AllocatedVar - \{v_j\}$
                $FreeRegs := FreeRegs + \{R[v_j]\}$
            end
        end
    end
    ii. allocate or spill:
    if $\exists r \in FreeRegs$ then
        $R[v_i] := r$
        $FreeRegs := FreeRegs - \{r\}$
        $AllocatedVar := AllocatedVar + \{v_i\}$
    else
        find $v_j \in \{\{v_i\} + AllocatedVar\}$ whose next use is furthest away
        if $v_j \neq v_i$ then
            $R[v_i] := R[v_j]$
            $AllocateVar := AllocateVar - \{v_j\} + \{v_i\}$
        end
        update $t_{s\_j}$ to the next use point of $v_j$
        update $Var$ according to this change
        add spill code for $v_j$ at this point and update $MC$
    end
end
```

interval is effectively cut short.

In our algorithm, we do not spill the same variable every time it is used. Instead, after spilling a variable we update the start point of the spilled variable with its next use point. Thus, it is possible that when the spilled variable is used again, there are available registers for it. This can potentially improve the performance. Finally, the codes are generated to meet the requirements of SPERF.

7.4.3 Operation Modes

As discussed in section 7.3-A, we maintain two operation modes, the normal mode and the secure mode. In the normal mode, the redundant bank stores original data and there is no precharge phase on the write bitlines. When it is necessary, the redundant bank may even be shut down for leakage power saving. This is especially useful when the redundant bank is purely hardware-enabled redundancy. In the secure mode, the redundant bank operates on flipped copies and precharging operations are involved.

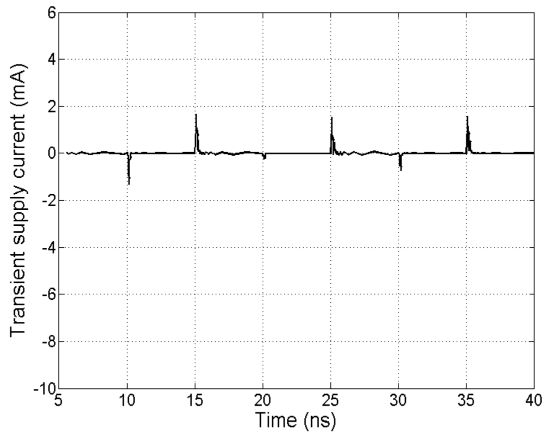
Thus, we need to add a processor status bit $\overline{secure\_mode}$ to indicate the operation mode of the register file. In addition, new instructions need to be added to allow the security applications to enter and leave the secure mode and to allow the operating system to save and restore the status bit. When an application needs to perform a cryptographic algorithm, it first enters the secure mode with the new instruction that sets $\overline{secure\_mode}$ to 0. When the cryptographic operation is done, the application leaves the secure mode by setting $\overline{secure\_mode}$ to 1.

7.5 Evaluation and Discussion

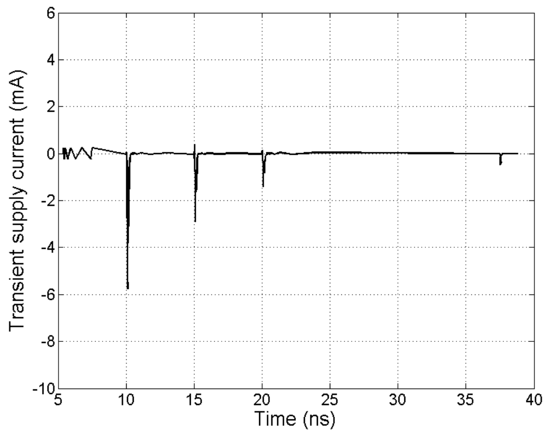
In this section, we first validate the security enhancement of our register file design. Then, energy overhead and performance-area tradeoffs are also studied.

7.5.1 Resistance to Power Analysis Attacks

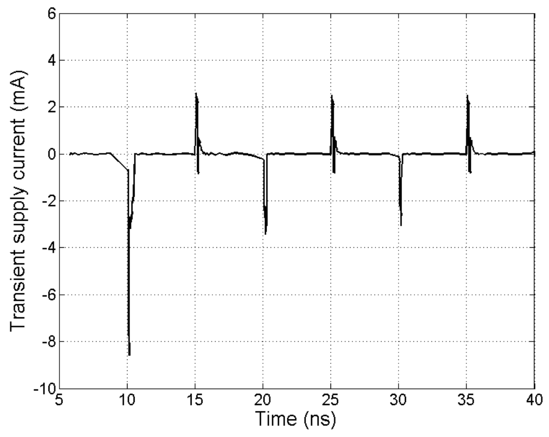
To validate the security enhancement for power analysis attacks, we assume a 32X32 register file that has 4 read ports and 3 write ports. In order to obtain supply current traces as accurately as possible, we implemented the physical design of the register file in TSMC 0.18-μm process technology [77] and ran simulations at the transistor level, considering parasitics with Spectre (a SPICE simulation tool from Cadence [62]). The simulations ran with a clock of $100MHz$.



(a) One read per cycle



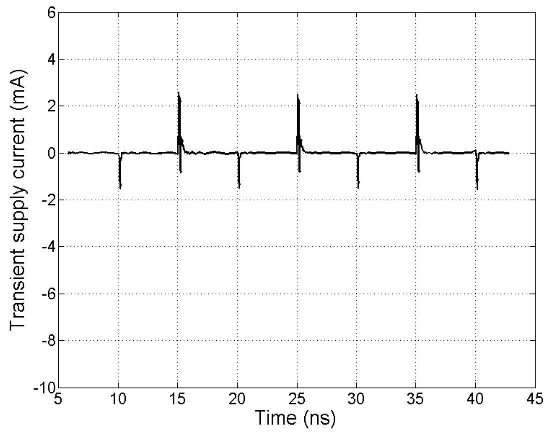
(b) One write per cycle



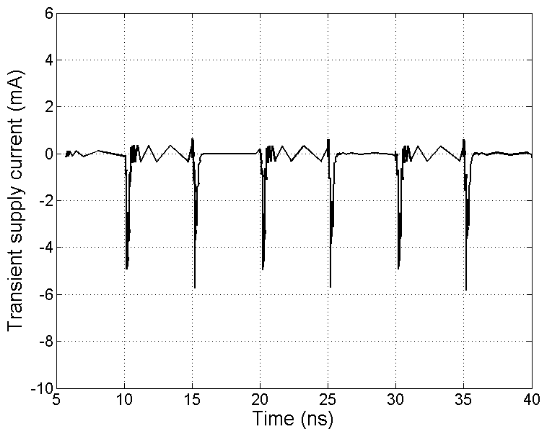
(c) Two reads and one write per cycle

Figure 7.4: Supply current for conventional register file.

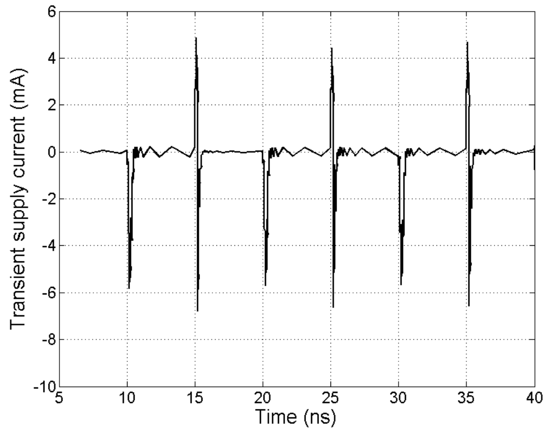
To validate our proposed technique, we first compare SPERF with the conventional approach. Figure 7.4 shows the transient supply current for a conventional register file. Figure 7.4(a) shows the results for three read operations, each reading a different register. The data in the three cycles are 32'h0000_0000, 32'h0000_FFFF, and 32'h0000_5555. The three reads are performed through the same bit lines, which has the value 32'hFFFF_FFFF before the first read. Figure 7.4(b) shows the results for three write operations, writing 32'h0000_0000, 32'h0000_FFFF, and 32'h0000_5555 to the same register that holds 32'hFFFF_FFFF originally. The same write bit lines



(a) One read per cycle



(b) One write per cycle

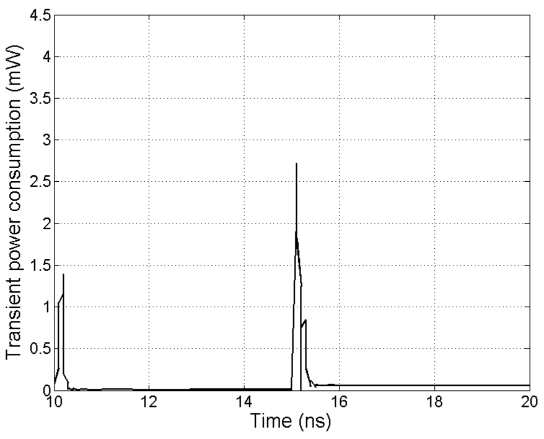


(c) Two reads and one write per cycle

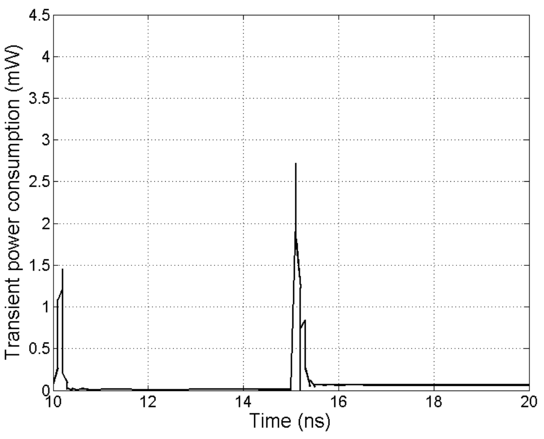
Figure 7.5: Supply current for SPERF.

were employed for all three write operations. Figure 7.4(c) shows the results of performing two read operations and one write operation per cycle for three cycles. The operations are on different registers, but the data that are read out or written to registers are same in each cycle. The data in cycles 1, 2, and 3 are 32'h0000_0000, 32'h0000_FFFF, and 32'h0000_5555, respectively. From the figures, we can see that the power consumption of the conventional register file is highly data-dependent.

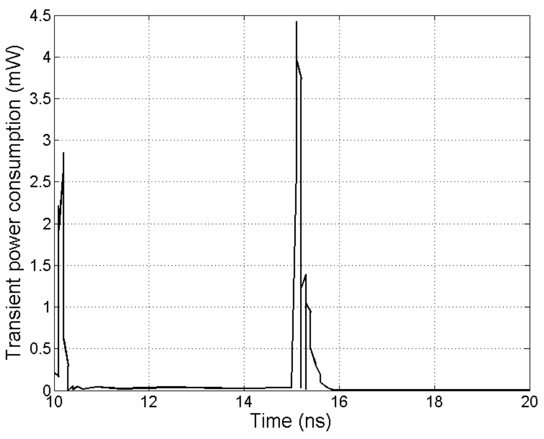
In Fig. 7.5, we present the simulation results for SPERF performing the same operations. Figures 7.5(a), (b), and (c) show the results for reading one register,



(a) Conventional register file



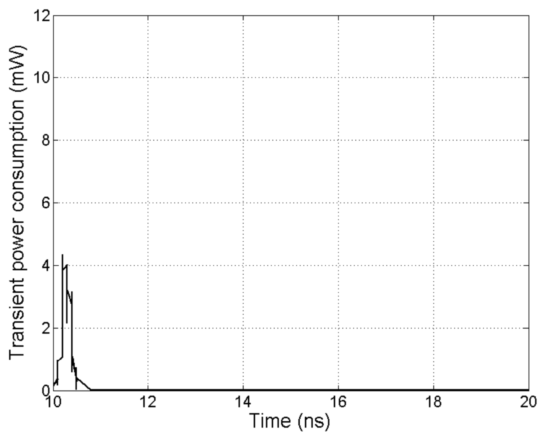
(b) SPERF in the normal mode



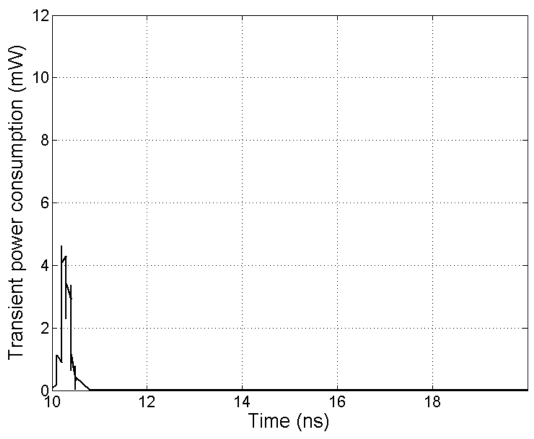
(c) SPERF in the secure mode

Figure 7.6: Transient power consumption during a read operation.

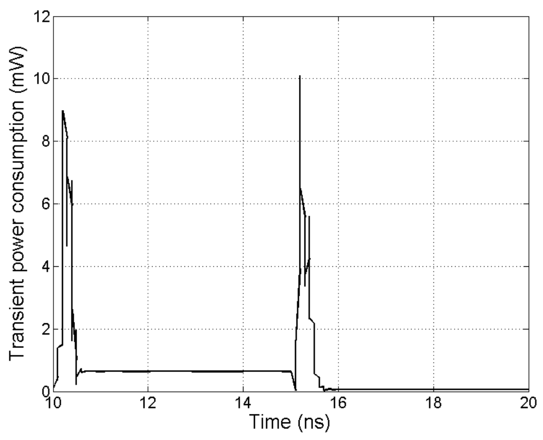
writing to one register, and reading two register and writing to one register in the same cycle, respectively. Figure 7.5(b) clearly shows the precharges for write operations. It can be seen that the power consumption of each type of operation is similar in all three scenarios. As the power consumption is balanced and independent of data value, the proposed SPERF can mitigate power analysis attacks. The access latencies of SPERF read and write operations are only slightly increased from the conventional register file. This is in general acceptable for low-end processors vulnerable to power analysis attacks. Note that although there is an additional precharge phase for the



(a) Conventional register file



(b) SPERF in the normal mode



(c) SPERF in the secure mode

Figure 7.7: Transient power consumption during a write operation.

write operations, performance is not degraded as long as the writes can finish in the second half of the cycle and the back-to-back write and read accesses are supported by data forwarding when necessary.

7.5.2 Energy Overhead

As mentioned in section 7.3, the improvement in security comes at the cost of energy overhead. We compare the average power consumption for the following three settings: conventional register file, SPERF in the normal mode, and SPERF in the secure mode. Figures 7.6 and 7.7 compares the transient power consumption for read and write operations, respectively. In the simulations, we assume that half of the bits read from a register are 0 and that half of the bit cells are switched during write operations. From the figures, we can see that in the normal mode, SPERF consumes almost the same amount of power as the conventional design, for both read and write. In the secure mode, however, the power consumption of SPERF is roughly doubled for read operations. Because of the newly added precharge phases, write operations consumes almost 3.6x power than the conventional design. The overhead is large. Fortunately, the overhead is only for the secure mode. The overall power overhead is determined by how often and how long the specific workloads need the system to stay in the secure mode.

7.5.3 Performance-Area Tradeoffs

Table 7.1: Performance overhead of different design options compared with conventional register file.

Security prog.	**Functions**	**Performance overhead**				
		16-reg ($R_s = 16$, $R_h = 0$)	20-reg ($R_s = 12$, $R_h = 8$)	24-reg ($R_s = 8$, $R_h = 16$)	28-reg ($R_s = 4$, $R_h = 24$)	32-reg ($R_s = 0$, $R_h = 32$)
AES-128	all functions	0	0	0	0	0
SHA-1	all functions	0	0	0	0	0
ECC-160	addition	0	0	0	0	0
	subtraction	0	0	0	0	0
	inversion	0	0	0	0	0
	modular reduction	4.8%	2.2%	0	0	0
	scalar multiplication	32.6%	17.2%	9.9%	7.4%	0
	point addition	4.9%	2.4%	1.2%	0.8%	0
	point multiplication	30.5%	16.2%	7.5%	5.7%	0

As discussed in section 7.4, there are different design options for SPERF. Hardware-enabled redundancy trades off area for performance, whereas software-enabled redundancy minimizes area overhead but generally may cause performance slowdown for security programs. Thus, a hardware/software co-design using a combination of both forms of redundancy may lead to better tradeoffs for conflicting requirements on performance and area overhead.

To evaluate the performance-area tradeoffs of SPERF, we set up experiments using an MPC875 board (a member of MPC885 family [79]), which contains a 32-bit PowerPC processor core with instruction and data caches (both are 8KB size). In the experiments, we study three widely-used cryptographic algorithms, i.e., Advanced Encryption Standard (AES) [80, 81], Secure Hash Algorithm (SHA) [82], and Elliptic Curve Cryptography (ECC) [7, 8, 83] and evaluate the performance of these algorithms during secure mode. The cryptographic algorithms are implemented according to different design options of SPERF. Specifically, the number of architectural registers varies from 16, 20, 24, 28, to 32, and assembly codes are generated accordingly using Algorithm. 1 in section 7.4-B. Then, runtime performance in terms of execution time is measured in 500 rounds of execution with random inputs. The timebase register of the processor, which is a 64-bit free-running binary counter clocked by a 1-MHz clock, is accessed to measure runtime performance at a reasonable accuracy.

We compare SPERF implemented in different ways with conventional register file design. The performance overhead is shown in Table. 7.1. Interestingly, SPERF does not introduce any performance overhead in many cases, such as all the functions of AES-128 and SHA-1, as well as addition, subtraction, and inversion of ECC-160. This observation indicates that for many specific security applications SPERF implemented with software-enabled redundancy can achieve security enhancement at no cost of performance or area overheads.

On the other hand, when the system needs to run public-key cryptographic algorithms, such as ECC, SPERF with software-enabled redundancy may not meet the performance requirements. For example, scalar multiplication and point multiplication of ECC-160 experience 32.6% and 30.5% slowdown during secure mode, respectively. SPERF with hardware-enabled redundancy can minimize the performance overhead but doubles the area of register file. Alternatively, hardware/software codesigns taking advantage of both forms of redundancy may be a better option. For example, designs using 20 and 24 architectural registers, shown as 20-reg and 24-reg in Table. 7.1, respectively lead to 17.2% and 9.9% slowdown in scalar multiplication, 16.2% and 7.5% slowdown in point multiplication, and negligible performance overhead in other operations during secure mode. In contrast to hardware-enabled redundancy, these two designs only introduces 25%-50% area overhead as compared with conventional design without protection from power analysis attacks. Thus, they may achieve a better tradeoff. On the other hand, another hardware/software co-design (28-reg) can further improve performance a bit (only approximately 2% for the above two operations) but at the cost of 25% more area overhead.

Chapter 8

Conclusions and Further Research

This dissertation summarizes the research that I have done during my PhD study under my major advisor Lei Wang. In the past years, we have investigated on how to address the emerging challenges in nanoscale computer systems, such as reliability, performance, and chip security. These challenges are often conflicting with each other and require careful tradeoffs and effective resource management. We have proposed novel techniques to satisfy the different requirements in a unified manner. These techniques are different from many existing solutions that mainly focus on addressing one particular challenge. This is expected to be a promising direction of research that can lead to the advancement of computer systems and my past research has paved the way for this goal. In my future research, I plan to investigate on various design requirements and to propose optimal solutions that can achieve better tradeoffs in the design space spanning over these correlated dimensions. Some of the directions of my future research are described below.

It is predicted that the number of cores per die will be doubling every two years. This trend towards multi-core/many-core computing compels us to explore effective solutions to unfold the full potentials of nanoscale integration. How to utilize the abundant hardware resources to achieve reliable, high-performance, and low power computing in the multi-core era is an important research topic. In particular, the massive computing horsepower rendered by the ultra-high integration density needs to be effectively converted to high performance. At the same time, the processor cores are inevitably facing reliability degradation, power/thermal constraints, and security challenges, etc. This compels us to find solutions that take into consideration various design requirements and achieve better tradeoffs.

In my past research, I propose the dynamic redundancy allocation technique to efficiently manage the abundant computation elements according to the requirements of reliability and performance in order to address both issues jointly. Besides, in my proposed soft redundancy techniques, under-utilized memory resources are exploited dynamically according to the requirements of reliability, performance, and bandwidth usage. To address the challenges in the multi-core era, the proposed techniques can be further studied. Design considerations for power, thermal, and security will also be included in the future research.

Besides, I am also very interested in studying the following questions: (a) How to dynamically distribute the workloads among the processor cores in order to satisfy the various requirements in an optimal manner? (b) How should the processor cores collaborate with each other and achieve self-organizing, like what ant colonies do, for instance? (c) How to achieve self-assembly, self-repairing, and self-reconfiguration based on the varying requirements of applications? and (d) How to design memory architectures and software to provide better support?

Moreover, further studies on chip security against side-channel attacks would be an interesting topic in the future. Many existing solutions are ad-hoc and only targeting specific attacks. A major obstacle for this research is the lack of a sound theoretical framework/model to accurately analyze the side-channel security and to evaluate attacks/countermeasures. Establishing such a model would have a large impact to this area.

Bibliography

[1] The International Technology Roadmap for Semiconductors, 2001 edition, *http://www.itrs.net/Links/2001ITRS/Home.htm.*

[2] R. Martel, V. Derycke, J. Appenzeller, S. Wind, and P. Avouris, "Carbon nanotube field-effect transistors and logic circuits," *Design Automation Conf.*, pp. 94-98, 2002.

[3] Y. Huang, X. Duan, Y. Cui, L. J. Lauhon, K-Y. Kim, and C. M. Lieber, "Logic gates and computation from assembled nanowire building blocks," *Science*, vol. 294. no. 5545, pp. 1313-1317, 2001.

[4] C. S. Lent, P. D. Tougaw, W. Porod and G. H. Bernstein, "Quantum cellular automata," *Nanotechnology*, vol. 4, pp. 49-57, 1993.

[5] P. Mazumder, S. Kulkarni, M. Bhattacharya, J. P. Sun and G. I. Haddad, "Digital circuit applications of resonant tunneling devices," *Proc. the IEEE*, vol. 86, no. 4, pp. 664-686, 1998.

[6] M. Mishra and S. C. Goldstein, "Defect tolerance at the end of the roadmap," *Intl. Test Conf.*, pp. 1201-1211, 2003.

[7] C. M. Jeffery and R. Figueiredo, "Hierarchical fault tolerance for nanoscale memories," *IEEE Trans. Nanotechnology*, vol. 5, no. 44, pp. 407-414, 2006.

[8] P. J. Kuekes, W. Robinett, G. Seroussi and R. S. Williams, "Defect-tolerant interconnect to nanoelectronic circuits: internally redundant demultiplexers based on error-correcting codes," *Nanotechnology*, vol. 16, no. 6, pp. 869-882, 2005.

[9] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," in C. Shannon and J. McCarthy, editors, *Automata Studies*, Princeton University Press, 1956.

[10] SPEC CPU2000 at *http://www.spec.org/cpu/.*

[11] D. M. Tullsen, S. J. Eggers, and H. M. Levy. "Simultaneous multithreading: Maximizing on-chip parallelism," *Intl. Symp. Computer Architecture*, pp. 392-403, 1995.

[12] J. L. Lo, et al., “Converting Thread-Level Parallelism Into Instruction-Level Parallelism via Simultaneous Multithreading,” *ACM Trans. Computer Systems*, pp. 322-354, Aug. 1997.

[13] K. Olukotun *et al.*, “The Case for a Single Chip Multiprocessor,” *Intl. Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, pp. 2-11, 1996.

[14] L. Hammond, B. Nayfeh, and K. Olukotun, “A Single-Chip Multiprocessor,” *IEEE Trans. Computers*, vol. 30, no. 9, pp. 79-85, 1997.

[15] T.-F. Chen and J.-L. Baer, “Effective hardware-based data prefetching for high-performance processors,” *IEEE Trans. Computers*, vol. 44, no. 5, pp. 609-623, 1995.

[16] D. Callahan, K. Kennedy, and A. Porterfield, “Software prefetching,” *Intl. Conf. Architectural Support For Programming Languages and Operating Systems*, pp. 40-52, 1991.

[17] D. Chandra, et al., “Predicting inter-thread cache contention on a chip multi-processor architecture,” *Intl. Symp. High Performance Computer Architecture*, pp. 340-351, Feb. 2005.

[18] G. E. Suh, L. Rudolph, and S. Devadas, “Dynamic partitioning of shared cache memory,” *The Journal of Supercomputing*, pp. 7-26, Apr. 2004.

[19] A. Settle, et al., “A Dynamically Reconfigurable Cache for Multithreaded Processors,” *The Journal of Embedded Computing*, special issue, Dec. 2005.

[20] S. Goldstein and M. Budiu, “NanoFabrics: spatial computing using molecular electronics,” *Intl. Symp. Computer Architecture* pp. 178-189, 2001.

[21] S. Narendra, D. Blaauw, A. Devgan, and F. Najm, “Leakage issues in IC design: Trends, estimation and avoidance,” *Intl. Conf. Computer Aided Design (Tutorial)*, pp. 11, 2003.

[22] S. Wang and L. Wang, “Exploiting soft redundancy for error-resilient on-chip memory design,” *Intl. Conf. Computer-Aided Design*, pp. 535-540, 2006.

[23] S. Wang and L. Wang, “Soft-redundancy allocated cache microarchitecture,” *Annual Boston Area Computer Architecture Workshop*, pp. 43-48, 2007.

[24] S. Wang and L. Wang, “Exploiting memory soft redundancy for joint improvement of error tolerance and access efficiency,” *IEEE Trans. VLSI Systems*, accepted as a regular paper.

[25] S. Wang and L. Wang, “Joint performance improvement and error tolerance for memory design based on soft indexing,” *Intl. Conf. Computer Design*, pp. 25-30, 2006.

[26] S. Wang and L. Wang, “Thread-associative memory for multicore and multithreaded computing,” *Intl. Symp. Low Power Electronics and Design*, pp. 139-142, 2006.

[27] S. Wang and L. Wang, “Design of Error-Tolerant Cache Memory for Multithreaded Computing,” *Intl. Symp. Circuits and Systems*, pp. 1850-1853, 2008.

[28] S. Wang, L. Wang and F. Jain, “Towards achieving reliable and high-performance nanocomputing via dynamic redundancy allocation,” *ACM Journal of Emerging Technologies in Computing*, accepted as a regular paper.

[29] S. Wang, F. Zhang, J. Dai, L. Wang, and J. Shi, “Making register file resistant to power analysis attacks,” *Intl. Conf. Computer Design*, 2008, accepted.

[30] S. Wang, J. Dai, E.-S. Hasaneen, L. Wang, and F. Jain, “Programmable Threshold Voltage Using Quantum Dot Transistors for Low-Power Mobile Computing,” *Intl. Symp. Circuits and Systems*, pp. 3350-3353, 2008.

[31] S. Wang, J. Dai, and L. Wang, “Defect-Tolerant Digital Filtering with Unreliable Molecular Electronics,” *Workshop on Signal Processing Systems*, 2008, accepted.

[32] S. Wang, J. Dai, and L. Wang, “Digital filtering with unreliable molecular electronics,” *IEEE Intl. Conf. Nanotechnology*, pp. 891-894, 2008.

[33] S. Wang and L. Wang, “A defect-tolerant memory nanoarchitecture exploiting hybrid redundancy,” *IEEE Intl. Conf. Nanotechnology*, pp. 707-710, 2008.

[34] S. Wang and L. Wang, “Dynamic Redundancy Allocation for Reliable and High-Performance Nanocomputing,” *Intl. Symp. Nanoscale Architectures*, pp. 1-6, 2007.

[35] S. Wang, J. Dai, E.-S. Hasaneen, R. Shankar, R. Velampati, L. Wang, and F. Jain, “Low-Power CMOS using Programmable Threshold QD-FETs,” *Connecticut Microelectronics and Optoelectronics Consortium*, 2007.

[36] L. Wang and S. Wang, “Adaptive timing for analysis of skew tolerance,” *Intl. Symp. Circuits and Systems*, pp. 976-979, 2006.

[37] P. Dusart, G. Letourneux, and O. Vivolo, “Differential fault analysis on A.E.S,” *Cryptology ePrint Archive*, Report 2003/010. Available at http://www.iacr.org.

[38] P. Maistri and R. Leveugle, “Double-data-rate computation as a countermeasure against fault analysis,” *IEEE Trans. Computers*, vol. 57, no. 11, pp. 1528-1539, 2008.

[39] L. Breveglieri, I. Koren, and P. Maistri, “Incorporating error detection and online reconfiguration into a regular architecture for the Advanced Encryption Standard,” *IEEE Int’l Symp. Defect and Fault-Tolerance in VLSI Systems*, pp. 72-80, 2005.

[40] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," *CRYPTO*, pp. 104-113, 1996.

[41] D.J. Bernstein, "Cache-timing attacks on AES," *Technical Report*, Apr. 2005. Available at http://palms.ee.princeton.edu.

[42] D. A. Osvik, A. Shamir, and E. Tromer, "Cache-attacks and countermeasures: the Case of AES," *Lecture Notes in Computer Science*, vol. 3860, pp. 1-20, 2005.

[43] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *Intl. Symp. Computer Architecture*, pp. 494-505, 2007.

[44] P. Kocher, J. Jaffe, B. Jun, "Differential power analysis," *Lecture Notes in Computer Science* vol. 1666, pp. 388-397, 1999.

[45] J.S. Coron, "Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems," *Lecture Notes in Computer Science* vol. 1717, pp. 292-302, 1999.

[46] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," *Intl. Symp. Microarchitecture*, pp. 248-259, 1999.

[47] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The V-way cache : demand-based associativity via global replacement," *Intl. Symp. Computer Architecture*, pp. 544-555, 2005.

[48] C. Zhang, "Balanced cache: reducing conflict misses of direct-mapped caches," *Intl. Symp. Computer Architecture*, pp. 155-166, 2006.

[49] Y. Solihin, J. Lee, and M. Kharbutli, "Eliminating conflict misses using prime number-based cache indexing," *IEEE Trans. Computers*, vol. 54, no. 5, pp. 573-586, 2005.

[50] Y. Chen, G. Y. Jung, D. A. A. Ohlberg, X. Li, D. R. Stewart, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, and R. S.Williams, "Nanoscale molecular-switch crossbar circuits," *Nanotechnology*, vol. 14, pp. 462-68, 2003.

[51] European Commission, *Technology Roadmap for Nanoelectronics*, 2001.

[52] L. J. Durbeck and N. J. Macias, "The Cell Matrix: an architecture for nanocomputing," *Nanotechnology*, pp. 217-230, 2001.

[53] C. M. Jeffery and R. Figueiredo, "Hierarchical fault tolerance for nanoscale memories," *IEEE Trans. Nanotechnology*, vol. 5, pp. 407-414, 2006.

[54] M. H. Lee, Y. K. Kim, and Y. H. Choi, "A Defect-tolerant memory architecture for molecular electronics," *IEEE Trans. Nanotechnology*, vol. 3, pp. 152-157, 2004.

[55] W. Rao, A. Orailoglu, and R. Karri, “Architectural-level fault tolerant computation in nanoelectronic processors,” *Intl. Conf. Computer Design*, pp. 533-539, 2005.

[56] L. Wang and N. R. Shanbhag, “Low-power filtering via adaptive error-cancellation,” *IEEE Trans. Signal Processing*, vol. 51, no. 2, pp. 575-583, 2003.

[57] R. Hegde and N. R. Shanbhag, “Soft digital signal processing,” *IEEE Trans. VLSI Systems*, vol. 9, no. 6, pp. 813-823, 2001.

[58] J. Dai, L. Wang, and F. Jain, “Analysis of defect tolerance in molecular electronics using information-theoretic measures,” *IEEE/ACM Symp. Nanoscale Architectures*, pp. 21-26, 2007.

[59] A. V. Oppenheim and R. W. Schafer, “Discrete-Time Signal Processing,” 2nd ed., *Prentice Hall, Englewood Cliffs, NJ,* 1992.

[60] L. R. Rabiner, J. H. McClellan, and T. W. Parks, “FIR digital filter design techniques using weighted Chebyshev approximation,” *Proc. of the IEEE*, vol. 63, pp. 595-610, 1975.

[61] D. J. Bernstein, “Cache-timing attacks on AES,” *Preliminary report, available at cr.yp.to/antiforgery/cachetiming-20050414.pdf*, 2005.

[62] Cadence: *http://www.cadence.com.*

[63] E. Fetzer, L. Wang, and J. Jones, “The multi-threaded, parity protected, 128 word register files on a dual-core Itanium ® family processor,” *Intl. Solid-State Circuits Conf.*, pp. 382–383, 2005.

[64] P. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” *Annual International Cryptology Conference, Lecture Notes in Computer Science*, vol. 1666, pp. 388–397, 1999.

[65] E. Brier, C. Clavier, F. Olivier, “Correlation power analysis with a leakage model,” *Workshop on Cryptographic Hardware and Embedded Systems, Lecture Notes in Computer Science*, vol. 3156, pp. 16–29, 2004.

[66] E. Peeters, F.-X. Standaert, J.-J. Quisquater, “Power and electromagnetic analysis: improved model, consequences and comparisons,” *Integration, the VLSI Journal*, vol. 40, no. 1, pp. 52–60, 2007.

[67] D. May, H. L. Muller, and N. Smart, “Random register renaming to foil DPA,” *Workshop on Cryptographic Hardware and Embedded Systems*, pp. 28–38, 2001.

[68] D. May, H. L. Muller, and N.P. Smart, “Non-deterministic processors,” *Australasian Conference on Information Security and Privacy, Lecture Notes In Computer Science*, vol. 2119, pp. 115–129, 2001.

[69] S. Moore, R. Anderson, P. Cunningham, R. Mullins, and G. Taylor, “Improving smart card security using self-timed circuits,” *Intl. Symp. Asynchronous Circuits and Systems*, pp. 211–218, 2002.

[70] R. Muresan, H. Vahedi, Y. Zhanrong, and S. Gregori, “Power-smart system-on-chip architecture for embedded cryptosystems,” *Intl. Conf. Hardware/Software Codesign and System Synthesis*, pp. 184–189, 2005.

[71] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: the case of AES,” *The Cryptographers'Track at the RSA Conference, Lecture Notes in Computer Science*, vol. 3860, pp. 1–20, 2006.

[72] E. Oswald and M. Aigner. “Randomized addition-subtraction chains as a countermeasure against power attacks,” *Workshop on Cryptographic Hardware and Embedded Systems*, pp. 39–50, 2001.

[73] K. Tiri and I. Verbauwhede, ”Securing encryption algorithms against DPA at the logic level: Next generation smart card technology,” *Workshop on Cryptographic Hardware and Embedded Systems, Lecture Notes in Computer Science*, vol. 2779, pp. 125–136, 2003.

[74] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, “Register allocation via coloring,” *Computer Languages*, vol. 6, pp. 47–57, 1981.

[75] M. Poletto and V. Sarkar, “Linears scan register allocation,” *ACM Trans. Programming Languages and Systems*, vol. 21, no. 5, 895–913, 1999.

[76] V. Sarkar and R. Barik, “Extended linear scan: an alternate foundation for global register allocation,” *Compiler Construction*, pp. 141–155, 2007.

[77] TSMC. [Online] At *http://www.tsmc.com.*

[78] F. Zhang and Z. J. Shi, “Power analysis attacks on ECC randomized automata,” *Intl. Conf. Information Technology*, pp. 900–901, 2007.

[79] Freescale Semiconductor, “MPC885 PowerQUICC Family Reference Manual.” [Online] At *http://www.freescale.com/files/32bit/doc/ref_manual/MPC885RM.pdf.*

[80] J. Daemen and V. Rijmen, “The Design of Rijndael: AES - The Advanced Encryption Standard,” *Springer-Verlag*, 2002.

[81] NIST, “Advanced Encryption Standard.” [Online] At *http://csrc.nist.gov/archive/aes/index.html.*

[82] NIST, “Secure Hash Standard.” [Online] At *http://www.itl.nist.gov/fipspubs/fip180-1.htm.*

[83] H. Yan and Z. Shi, "Studying software implementations of Elliptic Curve Cryptography," *Intl. Conf. Information Technology*, pp. 78–83, 2006.

[84] Han, J. and Jonker, P., 2002. "A system architecture solution for unreliable nanoelectronic devices," *IEEE Trans. Nanotechnology*, vol. 1, no. 4, pp. 201–208.

[85] Bhaduri, D., Shukla, S., Graham, P., and Gokhale, M., 2007. "Comparing reliability-redundancy tradeoffs for two von Neumann multiplexing architectures," *IEEE Trans. on Nanotechnology*, vol. 6, no. 3, pp. 265–279.

[86] Roy, S. and Beiu, V., "Majority multiplexing-economical redundant fault-tolerant designs for nanoarchitectures," *IEEE Trans. Nanotechnology*, vol. 4, no. 4, pp. 441–451, 2005.

[87] Rao, W., Orailoglu, A., and Karri, R., "Architecture-level fault tolerant computation in nanoelectronic processors," *Intl. Conf. Computer Design*, pp. 533–542, 2005.

[88] Rao, W., Orailoglu, A., and Karri, R., "Towards nanoelectronics processor architectures," *Journal of Electronic Testing: Theory and Applications*, vol. 23, pp. 235–254, 2007.

[89] A. Nicolau and J. A. Fisher, "Measuring the parallelism available for very long instruction word architectures," *IEEE Trans. Computers*, vol. 33, no. 11, pp. 968–976, 1984.

[90] D. W., Wall, "Limits of instruction-level parallelism," *Wester Research Laboratory Research Report 93/6*, Digital Equipment Corporation, 1993.

[91] P. Beckett and A. Jennings, "Towards nanocomputer architecture," *Asia-Pacific Conf. Computer Systems Architecture*, pp. 141–150, 2002.

[92] Fountain, T. J., Duff, M. J. B. D., Crawley, D. G., Tomlinson, C. and Moffat, C., "The use of nanoelectronic devices in highly-parallel computing systems," *IEEE Trans. VLSI Systems*, vol. 6, no. 1, pp. 31–38, 1998.

[93] Franklin, M. and Sohi, G. S., "The expandable split window paradigm for exploiting fine-grain parallelism," *Intl. Symp. Microarchitecture*, pp. 58–67, 1992.

[94] Farkas, K., Chow, P., Jouppi, N., and Vranesic, Z., "The multicluster architecture: reducing cycle time through partitioning, " *Intl. Symp. Microarchitecture*, pp. 149–159, 1997.

[95] Zhu, W. and Fleisch, B., "Performance evaluation of soft real-time scheduling on a multicomputer cluster," *Intl. Conf. Distributed Computing Systems*, pp. 610–617, 2000.

[96] He, L., Jarvis, S. A., Spooner, D. P., Chen, X., and Nudd, G. R., "Dynamic scheduling of parallel jos with QoS demands in multiclusters and grids," *Intl. Workshop on Grid Computing*, pp. 402–409, 2004.

[97] Tang, X. Y. and Chanson, S. T., "Optimizing static job scheduling in a network of heterogeneous computers," *Intl. Conf. Parallel Processing*, pp. 373–382, 2001.

[98] X. Ma, J. Huang, and F. Lombardi, "A model for computing and energy dissipation of molecular QCA devices and circuits," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 3, no. 4, pp. 18:1–18:30, 2008.

[99] A. Nakajima, T. Futatsugi, K. Kosemura, T. Fukano, and N. Yokoyama, "Room temperature operation of Si single-electron memory with self-aligned floating dot gate," *Applied Physics Letters*, vol. 70, no. 13, pp. 1742-1744, 1997.

[100] A. DeHon, "Nanowire-based programmable architecture," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 1, no. 2, pp. 109–162, 2005.

[101] T. Wang, M. Ben-Naser, Y. Guo, and C. A. Moritz, "Wire-streaming processor on 2-D nanowire fabrics," *NSTI (Nano Science and Technology Institute) Nanotech*, 2005.

[102] Y. Chen, G.-Y. Jung, D. A. A. Ohlberg, X. Li, D. R. Stewart, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, and R. S. Williams, "Nanoscale molecular-switch crossbar circuits," *Nanotechnology*, vol. 14, no. 4, pp. 462–468, 2003.

[103] A. DeHon, S. C. Goldstein, P. J. Kuekes, P. Lincoln, "Nonphotolithographic nanoscale memory density prospects," *IEEE Trans. Nanotechnology*, vol. 4, no. 2, pp. 215–228, 2005.

[104] M.-H. Lee, Y. K. Kim, and Y.-H. Choi, "A defect-tolerant memory architecture for molecular electronics," *IEEE Trans. Nanotechnology*, vol. 3, no. 1, pp. 152–157, 2004.

[105] I. Koren, Z. Koren and C. H. Stapper, "A statistical study of defect maps of large area VLSI IC's," *IEEE Trans. VLSI Systems*, vol. 2, no. 2, pp. 249–256, 1994.

[106] C. M. Jeffery and R. J. O. Figueiredo, "Hierarchical Fault Tolerance for Nanoscale Memories," *IEEE Trans. Nanotechnology*, vol. 5, no. 4, pp. 407–414, 2006.

[107] F. Sun, L. Feng, and T. Zhang, "Run-time data-dependent defect tolerance for hybrid CMOS/nanodevices digital memories," *IEEE Trans. Nanotechnology*, vol. 6, no. 3, pp. 341–351, 2007.

[108] H. Naeimi, A. DeHon, "Fault secure encoder and decoder for Memory applications," *IEEE Intl. Symp. Defect and Fault-Tolerance in VLSI Systems*, pp. 409–417, 2007.

[109] P.J. Kuekes, W. Robinett, G. Seroussi, and R. Stanley Williams, "Defect-tolerant demultiplexers for nano-electronics constructed from error-correcting codes," *Applied Physics A: Materials Science & Processing*, vol. 80, no. 6, pp. 1161–1164, 2005.

[110] S.Williams and P.Kuekes, "Demultiplexer for a molecular wire crossbar network," *U.S. Patent* 6 256 767, July 3, 2001.

[111] A. DeHon, P. Lincoln, and J. Savage, "Stochastic assembly of sublithographic nanoscale interfaces," *IEEE Trans. Nanotechnology*, vol. 2, no. 3, pp. 165–174, 2003.

[112] M. Purser, "Introduction to error-correcting codes," *Artech House Publishers*, 1994.

[113] J. Edler and M. D. Hill, "Dinero IV trace-driven uniprocessor cache simulator," at *http://www.cs.wisc.edu/∼markhill/DineroIV/.*

[114] A. Milenkovic and M. Milenkovic, "Exploiting streams in instruction and data address trace compression," *IEEE Annual Workshop on Workload Characterization*, pp. 99-107, 2003.

[115] T. Calin, M. Nicolaidis, and R. Velazco, "Upset hardened memory design for submicron CMOS technology," *IEEE Trans. Nuclear Science*, vol. 43, pp. 2874-2878, 1996.

[116] G. Memik, M. T. Kandemir, and O. Ozturk, "Increasing Register File Immunity to Transient Errors," *Conf. Design, Automation and Test in Europe*, pp. 586-591, 2005.

[117] K. Chakraborty, S. Kulkami, M. Bhattacharya, P. Mazumder, and A. Gupta, "A physical design tool for built-in self-repairable RAMs," *IEEE Trans. VLSI*, vol. 9, pp. 352-364, 2001.

[118] R. Fellman, et.al., "Design and evaluation of an architecture for a digital signal processor for instrumentation applications," *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. 38, pp. 537-546, 1990.

[119] F. Wrobel, J.-M. Palau, M.-C. Calvet, O. Bersillon, and H. Duarte, "Simulation of nucleon-induced nuclear reactions in a simplified SRAM structure: scaling effects on SEU and MBU cross sections," *IEEE Trans. Nuclear Science*, pp. 48(6):1946-1952, 2001.

[120] S. S. Mukherjee, J. Emer, S. K. Reinhardt, "The soft error problem: an architectural perspective," *Proc. Intl. Symp. High-Performance Computer Architecture*, pp. 243-247, 2005.

[121] P. Hazucha, T. Karnik, "Characterization of soft errors caused by single event upsets in CMOS processes," *IEEE Trans. Dependable and Secure Computing*, vol 1, pp. 128-143, 2004.

[122] N. Seifert, X. Zhu, and L. W. Massengill, "Impact of scaling on soft-error rates in commercial microprocessors," *IEEE Trans. Nuclear Science*, vol 49, pp. 3100-3106, 2002.

[123] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. Device and Materials Reliability*, vol. 5, no. 3, pp. 305-316, 2005.

[124] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: an architectural perspective," *Intl. Symp. High-Performance Computer Architecture*, pp. 243-247, 2005.

[125] M. Horiguchi, "Redundancy techniques for high-density DRAMS," *IEEE Innovative Systems Silicon*, pp. 22-29, Oct. 1997.

[126] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji, "Adapting cache line size to application behavior," *Intl. Conf. Supercomputing*, pp. 145-154, 1999.

[127] P. Pujara and A. Aggarwal, "Increasing the cache efficiency by eliminating noise," *Intl. Symp. High Performance Computer Architecture*, pp. 145-154, 2006.

[128] K. Inoue, K. Kai, and K. Murakami, "Dynamically variable line-size cache exploiting high on-chip memory bandwidth of merged DRAM/logic LSIs," *Intl. Symp. High Performance Computer Architecture*, pp. 218-222, 1999.

[129] R. Subramanian, Y. Smaragdakis, and G. H. Loh, "Adaptive caches: effective shaping of cache behavior to workloads," *Intl. Symp. Microarchitecture*, pp. 385-396, 2006.

[130] S. Belfiore, L. Crisa, M. Grangetto, E. Magli, and G. Olmo, "Robust and edge-preserving video error concealment by coarse-to-fine block replenishment," *Intl. Conf. Acoustics, Speech, and Signal Processing*, vol. 4, pp. 3281-3284, 2002.