UnSync: A Soft Error Resilient Redundant CMP Architecture

by

Fei Hong

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved March 2011 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Rida Bazzi
Georgios Fainekos

ARIZONA STATE UNIVERSITY

May 2011

ABSTRACT

Reducing device dimensions, increasing transistor densities, and smaller timing windows, expose the vulnerability of processors to soft errors induced by charge carrying particles. Since these factors are inevitable in the advancement of processor technology, the industry has been forced to improve reliability on general purpose Chip Multiprocessors (CMPs). With the availability of increased hardware resources, redundancy based techniques are the most promising methods to eradicate soft error failures in CMP systems. This work proposes a novel customizable and redundant CMP architecture (UnSync) that utilizes hardware based detection mechanisms (most of which are readily available in the processor), to reduce overheads during error free executions. In the presence of errors (which are infrequent), the *always forward execution* enabled recovery mechanism provides for resilience in the system. The inherent nature of UnSync architecture framework supports customization of the redundancy, and thereby provides means to achieve possible performance-reliability trade-offs in many-core systems. This work designs a detailed RTL model of UnSync architecture and performs hardware synthesis to compare the hardware (power/area) overheads incurred. It then compares the same with those of the Reunion technique, a state-of-the-art redundant multi-core architecture. This work also performs cycle-accurate simulations over a wide range of SPEC2000, and MiBench benchmarks to evaluate the performance efficiency achieved over that of the Reunion architecture. Experimental results show that, UnSync architecture reduces power consumption by 34.5% and improves performance by up to 20% with 13.3% less area overhead, when compared to Reunion architecture for the same level of reliability achieved.

TABLE OF CONTENTS

Page

LIST OF TABLES iii

LIST OF FIGURES iv

CHAPTER

1 INTRODUCTION 1

2 RELATED WORK 4

3 OUR APPROACH: UNSYNC 7

Architecture Overview 7

Features of the UnSync Architecture 9

Implementation Details of UnSync 12

4 REUNION ANALYSIS 14

Issues with Reunion Architecture Implementation 14

5 EXPERIMENTAL SETUP 18

6 EXPERIMENTAL EVALUATION 20

Lower Area and Power Overheads in UnSync 20

Negligible Performance Overhead in UnSync 23

UnSync Performs better Across SER Rates 25

Customizable Redundancy with Less Complexity and Larger ROEC 26

7 CONCLUSION 30

REFERENCES 31

LIST OF TABLES

Table Page

1 Simulated Baseline CMP Parameters 18

2 Hardware Overhead Comparison 20

3 Comparison of Projected Die Sizes of Existing CMP Processors 22

4 Benchmarks with Different Reliability Requirements 27

LIST OF FIGURES

Figure Page

1 UnSync Architecture 7

2 Potential Problem of Write-back L1 Cache 12

3 The Core Pipeline Architecture in Reunion 14

4 Performance Overhead from Serialization 23

5 Performance Overhead from Increased Occupancy of ROB 24

6 Performance of UnSync with different CB sizes 26

7 Performance of Customizable UnSync with Different Redundancy 28

1 INTRODUCTION

The boons of technology scaling come with several consequences. We can build chips that have billions of transistors, but since we pack many of those transistors tightly together, the increased power density, diminishing node capacitances, and reduced noise margins make these transistors unreliable. To counter the power density problem, chip designers have resorted to multi-core architectures; which provide a way to continue improving performance, without commensurate increase in the power consumption. As a result, CMPs have become popular, e.g., Intel core 2 duo, Intel core i7, AMD Opteron, IBM Cell processor, etc. However, the reliability problem continues to grow. Transistors are becoming so small and fragile that a stray charge or high energy particle can cause current pulses on a transistor and toggle the logic value of the gate. This phenomenon, of radiation induced transient error, is referred to as “Soft Error”. The problem is that while high energy neutrons (100KeV - 1GeV from cosmic background) have caused soft errors for a long time, now low energy neutron particles (10meV - 1eV) also cause soft errors [29]. This effect is multiplied with the fact that there are many more low-energy particles, than those at higher energies [12]. Soft errors have already been attributed to cause large fiscal damages, e.g., Sun blamed soft errors for the crash of their million-dollar line SUN flagship servers in November 2000 [18]. At the current technology node, a soft error may occur in a high-end server once every 170 hours, but it is expected to increase exponentially with technology scaling and reach alarming levels of once-per-day! [14]

Chip Multiprocessors or CMPs are inherently good for reliability due to the availability of many cores, on which redundant computations can be performed for error detection, and/or correction. Many redundancy based techniques, at various levels of design space abstraction, based on Dual Modular Redundancy (DMR) [35], Triple Modular Redundancy (TMR) [33], and checkpointing [7] have been proposed to enable error detection and correction in CMPs. Reunion [31] is one of the most promising redundancy based multi-core architectures that achieves error resilience with low performance overhead. In Reunion, a hash of a set of instructions called *fingerprint* is generated at regular intervals and compared between redundant cores executing the same thread. The retired instructions are committed to their respective ARF (Architectural Register File) or memory *iff* the *fingerprints* are found to match; if not, the execution is resumed from the previous correct position. Though efficient in its design to detect and recover from errors, the Reunion methodology suffers from issues during implementation:

1. *significant changes to the core design.* Reunion implementation requires adding a new pipeline stage in the processor, resulting in increased design complexity, and high overheads of power and area.

2. *performance overheads due to serializing instructions.* Instructions that must finish execution before the processor can start executing new instructions, such as traps, memory barriers, and non-idempotent instructions; require the cores to synchronize, and therefore cause performance degradation. Furthermore, increased ROB (Re-order Buffer) occupancy during such scenarios, lead to significant performance degradation.

3. *ignorance of efficient hardware mechanisms for error detection.* Error detection in Reunion is implemented by comparing the results of the two cores, ignoring all the advances and possibilities of efficient error detection schemes in hardware, e.g., parity bits, DMR, etc.

In this thesis, we propose a novel redundancy based multi-core architecture for CMPs: *UnSync*. In this, already existing and readily available power efficient hardware error-detection mechanisms, to provide effective protection against infrequent soft errors, are used. UnSync consists of two identical cores executing the same thread; each core is modified with power/area efficient hardware only error-detection mechanisms, for every sequential element in the processor core. In the event of an error detected in one of the cores, execution in both the cores are stalled. The architectural state from the error-free core is copied onto the erroneous core to resume correct execution on both the cores, ensuring *always forward execution*, i.e., it does not go back for re-execution. Some popular redundancy based techniques, involve loose/tight lock-stepped executions on the cores, or synchronizing the execution on both the cores by memory accesses. On the contrary, our

UnSync architecture disassociates the two redundant cores during error-free execution as much as possible, thereby allowing for maximum possible performance in the absence of errors. The main justification for synchronizing the redundant cores, is to ensure that recovery overhead is minimized by re-execution when an error is detected; by copying the architectural state of the error-free core onto the erroneous core, both the cores resume execution from the instruction last stopped on the error-free core. Neither of the cores are required to re-execute instructions, and the amount of execution re-traced (on the erroneous core, to which the architectural state was copied), only depends on the difference in execution speeds of the two cores. Our recovery mechanism has a higher overhead, compared to popular redundancy based techniques for CMP. By reducing the performance overheads during error free execution, and given the fact that errors are infrequent, UnSync achieves better performance at lower power/area overheads and lesser design complexity. Unlike Reunion, the performance of UnSync is not affected by serializing instructions, or increased pipeline occupancy, since there is no inter-core communication. In addition, since every individual core is identical in its hardware architecture, the number and pairs of redundant cores in the multi-core system can be configured by the user, based on reliability and performance requirements.

To evaluate and compare the effectiveness of UnSync (compared to Reunion), we developed a multi-core power, performance and area estimation setup. We model the multi-core architecture and estimate program cycle times using the M5 simulator [4]. To estimate the power and area overhead of the core and pipeline architecture, we implemented changes required for both the *UnSync* and *Reunion* in a 5 stage pipelined RTL implementation of the MIPS [11] architecture. We synthesize and also place and route (PNR) the RTL models, using the Cadence Encounter [5] to evaluate the on-chip hardware overheads. The L1 cache area and power, is estimated using CACTI [22] models. Our experimental results show that the UnSync architecture achieves up to 20% improved performance with 14.6% reduced area and 34.5% lower power overhead as compared to the Reunion architecture.

2 RELATED WORK

Error resilient redundant processor designs must solve two key problems: maintaining identical instruction streams and detecting divergent executions, on the redundant cores. Mainframes, which have provided fault tolerance for decades, solve these problems by tightly lock-stepping two executions [30]. Lock-step ensures both processors observe identical load values, cache invalidations, and external interrupts. While conceptually simple, lock-step becomes an increasing burden as device scaling continues [19]. As technology scaling continues to concern the performance and power consumption overheads, multi-core designs are being investigated to keep up with Moore's Law [20]. The increase in the integration of a number of processor cores on a single chip makes the chip more dense in area and hence making them more vulnerable to reliability threats such as soft errors. On the other hand, CMPs inherently provide replicated hardware resources which can be exploited for error detection and recovery. A number of proposals [1, 9, 31, 32] have attempted to take advantage of the inherent replication of cores in CMPs to provide fault tolerance by pairing cores and checking their execution results.

Redundant multi-threading (RMT) [27] is a modified and efficient SMT (Simultaneous Multithreading) processor architecture where only store addresses and values are checked to detect soft errors. It uses a Load Value Queue (LVQ) to provide consistent replication, on redundant threads, of load values. Output comparison is performed by a store comparator (SC) that buffers completed but not yet committed stores. Once a store has been successfully verified, it is eligible to commit in each thread, and a single instance of the store is released to the memory hierarchy. The RMT technique was later extended to map redundant threads onto separate processor cores in a CMP, rather than separate hardware threads in an SMT. Chip-level redundant threading (CRT) [21] solved one source of resource contention while exacerbating another. Execution on separate cores eliminated contention by providing each thread its own private set of resources. In the work by Gomaa et.al [9], the comparison of load values is restricted to be based on the data-dependence chains in the executing threads, to reduce on performance overheads in comparison. However, CMP cores are not as tightly integrated as SMT threads, and the additional physical separation increases the round-trip store verification latency. This subsequently increases the average store execution time, which reduces any gains reaped from additional hardware resources, and results in a net slowdown. Fingerprinting [32] is a checkpointing scheme designed to minimize hardware changes to commodity hardware. Processor pairs identify errors by comparing cryptographic signatures that summarize architecture state updates. Mismatches trigger a rollback to a known good checkpoint; successful comparisons free prior checkpoints. Such techniques can be implemented cheaply, however they rely on heavy-weight checkpointing mechanisms that capture all of system states (including memory) and increase error detection latency. In this thesis, we adapt the benefits achieved from core-level redundancy, and propose a method to reduce the hardware overheads (memory storage, comparators, etc.) and also reduce inter-core communication (load values, fingerprint, etc.)to ensure resilience in the system.

To increase the resource usage and flexibility of CMPs, Gupta et al. [10], develop a redundancy based technique at a finer granularity. It connects the pipeline stages through routers and enables sharing of resources across cores. In Dynamic Dual Modular Redundancy (DDMR) [8], linking of cores for the purpose of redundant processing is dynamically done. Dynamic linking achieves two main benefits: 1) reliability is unaffected by defective cores; and 2) cores with similar throughput may be paired for the purpose of running redundant threads at similar speeds. Although having the benefit of flexibility and better resource usage, these techniques suffer from increased design complexity and high performance overhead. In addition, the larger hardware overheads and design complexity involved, limit their applicability when the number of cores in the system increases beyond hundreds.

Smolens et al. [31], in their work overcome the issues in design complexity and overheads in the load-value queue (LVQ) technique, and propose to relax strict input replication by allowing the redundant thread to issue loads directly to the memory system. However, to deal with input incoherence resulting from multiprocessor data races, it identifies cases where redundant loads receive updated values from other processors in the system. Such mismatches are essentially treated as transient errors – both threads re-issue their respective load and re-check the load values. The mismatch in load values is handled by issuing the load a third time via synchronizing memory requests that eliminate input incoherence for the requested cache line. Reunion [31] is one of the state-of-the-art CMP based redundant techniques. In order to synchronize between redundant core pairs and reduce bandwidth of comparison, Reunion proposes comparison of *fingerprints* between vocal and mute cores. However, our intensive analysis and experiments show that Reunion incurs high overheads in terms of area, performance, and power. In this thesis, we highlight in detail the design issues and hardware overheads involved in the implementation of Reunion on a many-core processor setup (at Section 4).

Error free execution, minimal error detection overhead (hardware or software) and possible customization of redundancy, are the three ideal expectations of a reliable many-core system. Our proposal, UnSync, satisfies these three ideal expectations: i) UnSync reduces error detection overheads in terms of area, performance, and power by exploiting readily-available hardware based error detection mechanisms; especially during error free executions. ii) UnSync further has simple and easy provisions to implement customizable redundancy and realize performance-reliability trade-offs in scalable many-core systems.

3 OUR APPROACH: UNSYNC

3.1 Architecture Overview

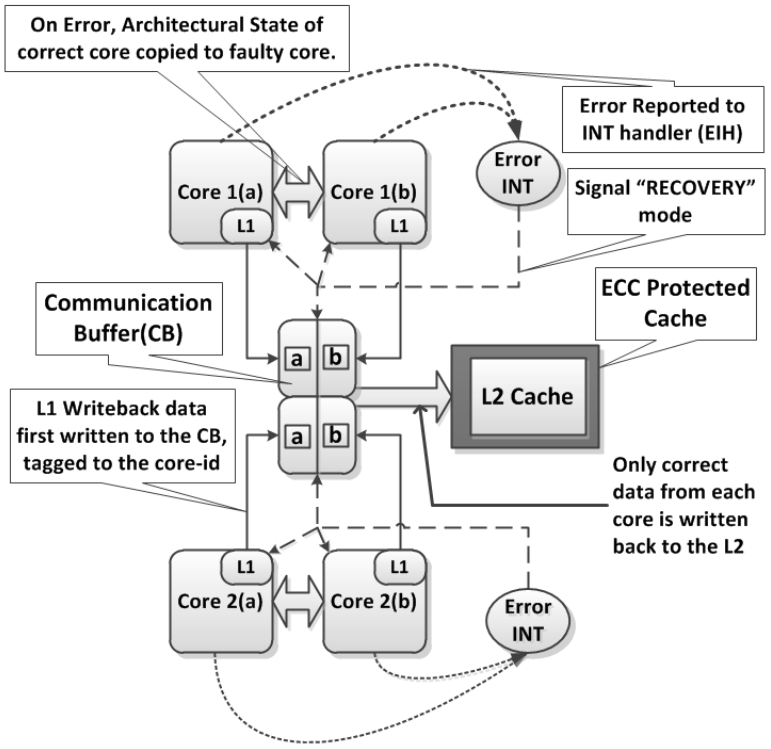
On Error, Architectural State of correct core copied to faulty core.
Error Reported to INT handler (EIH)
Error INT
Signal "RECOVERY" mode
Core 1(a)
L1
Core 1(b)
L1
Communication Buffer(CB)
ECC Protected Cache
a
b
L2 Cache
L1 Writeback data first written to the CB, tagged to the core-id
a
b
Only correct data from each core is written back to the L2
L1
Core 2(a)
L1
Core 2(b)
Error INT

Figure 1: **UnSync Architecture: The L1 of each redundant core writes into a CB, that acts as a secondary write buffer. Only one of the redundant copies, from the CB-pair (a, b), is written into the L2 cache. An error detected in any of the cores signals "RECOVERY" through the EIH, to the corresponding core-pair and the CB.**

Figure 1 describes an overview of the UnSync architecture. It shows two core-pairs of our two-way redundant UnSync architecture with their inter-core and intra-core communication links. Each core in this architecture, is configured with an on-core write-through L1 cache, and off-core shared ECC protected L2 cache. As part of the hardware based error-detection machinery, the L1 cache contains a parity-bit on each cache line to detect 1-bit errors. Similarly, the core architecture blocks are fitted with error-detection circuitry, details of which will be discussed in detail later in this section. The hardware error-detection blocks are connected to an *Error Interrupt Handler*(EIH) for each core-pair, to signal recovery in the even an error is detected. Data committed into the L1 cache, from each core of a core-pair executing identical threads of the program, is first written into a *Communi-*

cation Buffer (CB). From here, one copy of the data is passed on, to be written-back in the protected L2 cache.

The working of the UnSync architecture can be best studied by observing its phases of operation:

(a) Error-free Mode: The two identical cores in a core-pair execute the same thread of the application, where each core performs memory accesses on the shared L2 cache as independent cores. Data written into the L1-cache of a core, as it leaves the core (as in a write-through cache), is written into a non-coalescing CB, one for each core in the core-pair; as described in Figure 1. In the CB, each updated entry is tagged with its corresponding instruction address. As and when the L1-L2 data bus is free (available for data transfer), the latest entry, that has completed execution on both the CB is selected; and one copy of all the CB entries, earlier to this, are written into the L2 cache. This process ensures that, when processed data leaves the cores to be updated into the lower-level memory, both the cores have completed a particular state in the execution; and that since no error was detected during this time, the two copies are correct.

(b) Error-detection: Error detection in each core of the UnSync architecture, is enforced by the use of hardware-only error-detection blocks. The L1 cache, register file and the queuing structures are enabled with 1-bit parity based detection; owing to the fact that data write (parity generation) and read (parity verification) have a minimum of 1 cycle time difference. On the other hand, for the architecture blocks like the program counter and pipeline registers, where data is read/written on every cycle, parity-based detection cannot be employed; therefore Dual Mode Redundancy (DMR) based error detection is enabled. If any of these detection blocks determine an error in the data, on either core, an interrupt is transmitted to the EIH for that core-pair; which then performs error recovery. The interconnect between the core and the EIH is described by the dotted arrows in Figure 1.

(c) Recovery Mode: Once the EIH, receives an error interrupt, it signals "RECOVERY" to both the cores and CB of the corresponding core-pair. In this mode, the following procedure implements our "*always forward execution*" recovery mechanism:

1. Program execution on both the cores of the core-pair is stopped.

2. The pipeline of the erroneous core is flushed, so as to reset the pipeline registers.
3. The architectural state (register file, PC, etc.), and the content of the L1 cache, of the error-free core is copied onto that of the erroneous core (the core in which error was detected). This operation is performed by specific subroutines using the shared L2 cache.
4. Data transfer from the CB to the L2 cache is stopped. Only the transfers currently in flight are completed.
5. The content of the CB, corresponding to the erroneous core, is overwritten by data from the error-free core.
6. Once the architectural state, program counter, L1 cache contents and the CB content of both the erroneous core has been overwritten, both the cores resume execution of the program from the same program counter state as that was copied from the error-free core.

3.2 Features of the UnSync Architecture

Here we discuss some of the important features of UnSync that ensure its novel efficient error resilience.

3.2.1 Power-Efficient Error Detection in Hardware

The power and chip-area benefits of the UnSync architecture, hinges on the use of hardware based error-detection mechanisms in each of the cores, in a redundant multi-core setup. Single Error Correction and Double Error Detection (SECDED) a cache detection and correction mechanism incurs an overhead of 22% cache area, owing to tree of XOR gates, the depth of which increases super-linear to the number of bits covered by the ECC [26]. In addition, the ECC generation and verification logic requires more than one cycle to complete, which also has an impact on the cycle time of the processor. On the other hand, 1-bit parity based error detection mechanism requires only a negligible ($< 1\%$) power and area overhead [26]. In addition, the series of XOR gates don't impose a significant delay in its processing, and therefore can complete its operation in one cycle. It should be noted that, though the probability of an energy particle strike is uniform throughout the processor core, sequential elements which store data (even if it is for one cycle) are the most vulnerable architectural blocks [23]. TMR based detection and correction techniques for sequential elements, incur an overhead of 200% in power, while DMR based detection only technique requires only around 6% power overheads [8, 24].

Having identified that error detection on sequential elements and storage elements, can ensure error resilience in processors, an efficient choice of the right detection mechanism has to be made for each architecture block. In UnSync we choose either parity-bit or DMR based error detection methods. Owing to, the time latency requirements of the parity-bit generation and verification, data storage elements like: Load Store Queue (LSQ), Translation Lookaside Buffer (TLB), register file and the L1 cache data are enabled with parity-bit error detection. All the other sequential elements (e.g., pipeline register, `PC`) which have accesses on every cycle of processor execution; the 1 cycle latency of parity-bit technique is unacceptable, and therefore DMR based error-detection is employed. The use of DMR is reduced as much as possible, because of the hardware area and power overheads involved in every access.

Since only error-detection methods are employed within the core, to ensure error resilience, the redundancy of CMPs is used. By introducing a novel hardware only error reporting scheme, through the use of the EIH network in the UnSync architecture, area/power-efficient error resilience is achieved

3.2.2 Need for Synchronization Among the Cores is Eliminated

In popular redundancy based techniques for error resilience, the execution on the cores (or within a core) is synchronized by either lock-stepping [30] or through memory accesses [27], because: i) error detection is implemented by comparing the execution outputs or by comparing the memory accessed among the redundant threads (on the same or different cores), and ii) when an error is detected, both the cores can be directed to re-execute a set of instructions from a previously identified error-free position (e.g., checkpoint [32]). However, as suggested by the name of our architecture - *UnSync*, the need to synchronize execution among the redundant cores is eliminated. This is made possible by, i) hardware based error detection mechanisms that eliminate the need to compare redundant executions, and ii) the "*always forward execution*" based recovery mechanism ensures that the cores resume from the last executed position of the correct core and no re-execution is required in either cores.

When an error is detected on a core, execution on both the cores is stopped and architectural state from the error-free core is copied onto the erroneous core. While resuming the processor, the execution sequence is altered on only the erroneous core (since `PC` is copied from error-free core). The error-free core resumes from where it was stopped. Another aspect of this technique is that the amount of instructions re-traced (if any), by the erroneous core depends on the difference in the execution speeds between the two cores. In the case that the erroneous core was executing at a slower speed, during recovery, execution on this core is forwarded. The absence of re-executions in the recovery mechanism, provides some compensation to the overhead involved in the architectural state and L1 cache content copy from one core to the other.

3.2.3 Customizable Resilience and Redundancy

As the number of cores on a many-core architecture are scaled, the availability of cores and the requirement for soft error tolerance grow simultaneously. The Cisco Metro chip with 188 cores and 4 redundant cores per die, pose as an ideal example of defect tolerance in a many core architecture with core level redundancy. The overall system throughput is thus given by the maximum number of non-redundant tasks that can be executed (36 in the case of the Cisco Metro chip) on the processor. However, if the level of redundant cores executing a thread on the processor can be customized dynamically, it is possible to increase or decrease the system throughput in accordance with the reliability requirements and the current Soft Error Rate (SER). With the rapid increase in the number of cores on even general purpose processors, the need for user-configurable means to realize performance-reliability trade-offs become evident. In UnSync, no synchronization or data comparison exists between the two redundant cores; only an instruction completion check is performed at the CB, to write a single copy of data from the CB to the L2, when the data bus is available.

Since there does not exist any hardwired synchronization between the two cores, the CB and EIH are the only architecture blocks creating a virtual paring of the tow adjacent cores. This architecture thus provides easy means to configure the CB and EIH to decouple the two adjacent cores, thereby allowing the two cores to execute independent threads without error resilience on each. As a result, UnSync can be easily customized for non-redundant execution (statically or dynamically), and therefore a trade-off between reliability and performance can be achieved.

3.3 Implementation Details of UnSync

3.3.1 Need for Write-through L1 Cache configuration

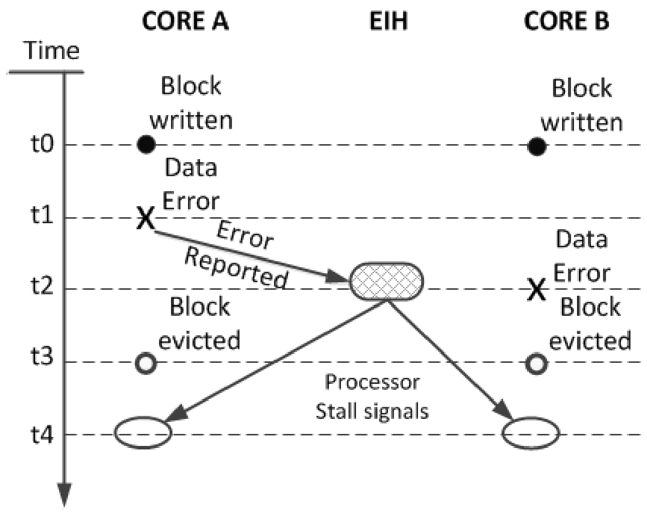

Figure 2: **Once an error is detected in a core, non-zero cycles are incurred in the communication to the EIH and subsequent RECOVERY signaling to stall the processor and perform cache copy. In write-back cache configuration, corrupt data (if any) in core B could be written into the memory or copied over to core A during recovery.**

Most architectures used in high reliability applications, use the L1 cache in write-through mode. This is because in the write-through configuration, a copy of the updated cache data always exists in the lower-level memory. Therefore, if an error is detected during read operation on a cache-line, the cache-line can be invalidated, and the correct updated cache line can be loaded from the memory. Here, we discuss the need for the write-through cache configuration and its importance in the UnSync architecture. If the UnSync core was configured using a write-back cache, an error on a cache-line may leave us in a irrecoverable state. Figure 2 describes such a scenario and thereby demonstrates the importance of the write-through cache in UnSync. Figure 2 shows two redundant cores with write-back caches, running an identical thread, and the events in the caches. At time $t0$, a data cache block is written (on both cores) and therefore the cache-line is deemed dirty. At time $t1$, an error is detected in core A, the error is reported to EIH. The EIH, on receipt of the error signal, transmits processor stall signals (RECOVERY) to both the cores, which requires a non-zero number of cycles for both the processor pipelines to stall. Based on information received by the EIH, core A is known to be erroneous, and has to be replaced with contents from core B. Meanwhile, an error may strike on a dirty cache block in core B, which would be detected only when read during the recovery process. Since the cache is a write-back cache, and the erroneous cache-line in core B is dirty it does not have an updated copy anywhere else in the system. This scenario thus makes it impossible to recover from errors on the cores. On the other hand, if the cache configuration was a write-through cache, as and when the block was updated, a copy is written into the L2 cache (or memory). On a similar situation as in Figure 2, it is possible to simply invalidate both the cache lines and treat the same as cache-misses as the correct updated copy of the cache blocks are available in the ECC protected L2 cache.

4 REUNION ANALYSIS

4.1 Issues with Reunion Architecture Implementation

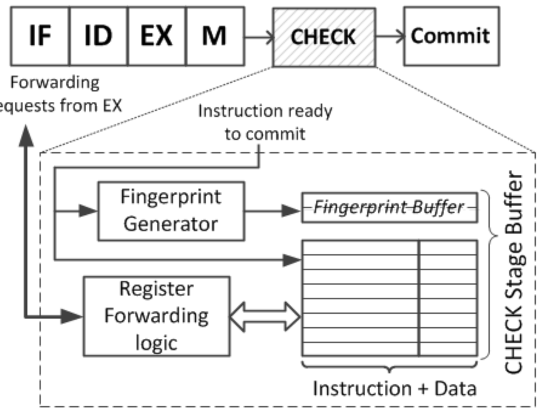

Figure 3: **The core pipeline architecture in Reunion is described, and the internal components of the additional pipeline stage (CHECK) is shown in detail.**

The *Reunion* paper is sketchy in the details of its hardware implementation. The authors, focus on the technique and architecture, without careful consideration to the increase in hardware complexity when implemented on a many-core system. Our meticulous analysis here, shows that Reunion implementation incurs significant hardware power, area and performance overheads.

4.1.1 Additional pipeline stage: "*CHECK*"

A key architecture block in the implementation of Reunion, is the additional pipeline stage: CHECK, highlighted in Figure 3. The function of this additional stage is to, **1)** generate the *fingerprint* – hash of the instruction and output-data of a set of instructions (*fingerprint interval* (FI)), **2)** send and receive the *fingerprints* to compare the same between the vocal and mute cores, **3)** temporarily store instruction and output data, before committing to the architectural register file and memory, in a buffer – *CHECK Stage Buffer*(CSB). From our hardware synthesis experiments (at the 65nm technology node, and details of setup in Section 5), we observe that the CHECK stage alone contributes significantly to the overall hardware overhead in the Reunion implementation: compared to the "Execute" pipeline stage, of the baseline MIPS core, it occupies 75% of chip-area and consumes 40% of per-access energy. The micro-architecture components that constitute the CHECK stage, described in Figure 3 are: the hashing circuitry – *Fingerprint Generator*, *CHECK Stage Buffer*, and their allied circuitry.

4.1.2 Fingerprint Generator

According to the authors in [31], the fingerprint generator logic is composed of a two stage, parallel, 16-bit Cyclic Redundancy Checking (CRC) logic [3,6,32]. On each The *fingerprint* is generated in parallel to the mechanism that stores instruction and data to the buffers. The *Fingerprint Generator* is composed of 238 gates [3], and this combinational logic cone is on the critical path during timing analysis of the CHECK stage. This adds to the complexity of its design and also the power-performance trade-offs realized in its implementation.

4.1.3 *CHECK* Stage Buffer

At the end of the "Memory" stage, in the Reunion pipeline, the data to be written to the memory is not written but stored temporarily along with the instruction in the CSB; as the corresponding *fingerprint* has to be verified before being Committed. Similarly, the data to be committed to the register files is also stored in the buffer. Considering the data bus speeds in [28], we observe that a minimum of 6 cycles is required to communicate the *fingerprint*, comparing the values, and deriving a result. While the *fingerprint* is being compared, the pipeline continues to executes instructions that will form the next *fingerprint*, and thus these instructions and output-data also have to be stored in the buffer. Since at any point in time, two *fingerprints* exist (one in the process of comparison, and the other due to parallel pipeline execution), an additional *fingerprint* buffer is required (described in Figure 3). For a FI of 10 (the minimum indicated in [31]), a total of 17 buffer entries are required, each of size 66-bits; forming a total of 17×66 = 1122 bits buffer.

In the buffer, each entry requires one exclusive write port and three exclusive read ports to allow buffering and parallel instructions commit, before and after *fingerprint* comparisons. During hardware implementation (at the 65nm technology node), we observe that the cell that forms each bit of the CSB is $10.40\mu m^2$ which is $1.3\times$ the size of a register file cell ($7.80\mu m^2$); because of the additional read port in the buffer. The chip-area occupied by the CSB is thus $1.46\times$ that of a 32 entry register file (32×32-bits). The authors in [31] indicate the possibility of increasing the FI (around $1\ 50$) to reduce the communication overhead, without any discernible difference in system performance. An increase in the FI involves an increase in the size of the CSB and the allied circuitry. We observe through synthesis, that for a FI of 50, the CSB alone occupies a chip-area of $39125\mu m^2$, which is 91% the size of the MIPS core without the cache ($42818\mu m^2$).

4.1.4 Register Forwarding Logic

Another consequence of buffering instructions and their output data before committing to their respective architectural states, is the possible pipeline stalls due to data starvation while executing data-dependent instructions. For example, if a register is updated by an instruction that is part of the *fingerprint* generated and currently under comparison; and a succeeding instruction reads from that register, the (W-R) data-dependency realized requires the pipeline to stall until the register file is committed. Since the output data is buffered in the CSB, a *Register Forwarding logic* can be employed to forward the stored updated register data to the instruction in the execution pipeline. In the Reunion implementation, such a forwarding mechanism is essential to maintain the minimal performance loss incurred in parallelizing the *fingerprint* based error detection process. On analysis of the hardware implementation (after place and route of synthesized blocks, as described in Section 5), we observe the forwarding logic and the datapaths between the buffer and the processor pipeline, adds to the overall hardware overhead [2].

In this, the forwarding logic datapaths add to the total wiring length of the Reunion implementation; which is 34% more than that of the baseline MIPS core. In addition, it should be noted here that these datapaths add to the total load capacitance of the connected blocks and thereby increase the energy consumed per access. Since the blocks attached to these datapaths (“ALU” and “CSB” blocks) are accessed on every cycle, this causes an overall increase in the power consumption of the processor. The observed phenomenon will only multiply with the increase in the issue-width of the processor. As discussed above, an increase in the FI will cause an increase in the size of the CSB. As a consequence to the increased buffer size, the number of datapaths is also increased, which would further increase area and power overhead estimated earlier.

4.1.5 Performance overheads

The *fingerprint* based error detection mechanism realizes two kinds of indirect hardware overheads. Firstly, pipeline occupancy increases from instructions in *CHECK* stage, occupying additional Reorder Buffer (ROB) capacity in the speculative window. For workloads that benefit from large instruction windows, this decreases opportunity to exploit memory level parallelism or perform speculative execution. Secondly, the serializing instructions (i.e., traps, memory barriers, etc.) cause the entire pipeline to stall in the event of dependent instruction executions, till the *fingerprint* including the serializing instruction is verified. These pipeline stalls cause a cascading effect on the ROB occupancy and thereby affect the performance of the system.

5 EXPERIMENTAL SETUP

Parameter	Configuration
Processor Cores	4 logical cores, Alpha 21264 2GHz, 5-stage pipeline; out-of-order 4-wide fetch/issue/commit
Issue Queue	64
L1 Cache	32KB split I/D, 2-way, 10 MSHRs 2 cycle access latency, 64-byte/line
Shared L2 Cache	4MB, 8-way, 64-byte/line 20-cycle access latency, 20 MSHRs
I-TLB	48 entries, 2-way
D-TLB	64 entries, 2-way
Memory	3GB, 64-bit wide, 400 cycles access latency

Table 1: **Simulated Baseline CMP Parameters**

To evaluate and compare the effectiveness of UnSync, in comparison with Reunion, we develop a multi-core setup to estimate power, performance and area. The cycle-accurate M5 multi-core simulator [4] is modified to model the UnSync and Reunion implementation on a 4 core processor. The specifications of each core is tabulated in Table 1. The simulator is instrumented to obtain application statistics of cycle-time and the cycle-delays of each architecture block. We model accurately, 1) the faster core's delay due to stalls during the execution of serializing instructions and increased ROB occupancy in the Reunion architecture, and 2) the stalls caused when the CB is full and the bus is busy. We experiment over benchmarks from SPEC2000 and MiBench.

To estimate the power and area overheads incurred, we perform hardware synthesis (using the Cadence Encounter [5]) on an RTL implementation of the the MIPS [11] processor. We implement both the UnSync and Reunion architecture on the baseline MIPS core. In our analysis since each core is identical and executes redundantly, we compare and contrast the area and power of only a single core in three configurations. The hardware components added to the MIPS core, for the Reunion implementation are: *fingerprint* size=16bits, *fingerprint interval*=10 instructions and the CHECK Stage Buffer=17entries each of 66bits. In the case of UnSync, the L1 cache is in Write-through configuration with 10 entries in the Communication Buffer, for each core. We synthesize the three core models for the same frequency of 300MHz at 65nm technology. In order to accurately evaluate the impact of datapaths and interconnects in the processor implementations, we perform place-and-route (PNR) at the nominal density of 0.49. For an accurate analysis of cache area and power, we estimate the parameters using CACTI cache simulator [22]. We scale cache line size (and total cache-size accordingly) to derive power and area values for the cache including parity-bit and SECDED [16] error protection mechanisms.

6 EXPERIMENTAL EVALUATION

6.1 Lower Area and Power Overheads in UnSync

6.1.1 Error detection mechanisms incur lower overheads

Here, we summarize the area and power overheads, of the two redundant processor architecture implementations (UnSync and Reunion), in comparison with the baseline MIPS core. The hardware parameters discussed here are that of a single core after PNR and therefore demonstrate with greater accuracy the actual power and area specifications of the chip, in each configuration, after fabrication.

Parameter	**Basic MIPS**	**Reunion**	**UnSync**
Chip-Area Overhead			
Core (μm^2)	98558	144005	115945
L1 Cache (mm^2)	0.1934	0.2086	0.1939
CB (mm^2)	N/A	N/A	0.00387
Total Area (μm^2)	291958	352605	313715
Overhead (%)	N/A	20.77	7.45
Power Overhead			
Core (W)	1.153	2.038	1.635
L1 Cache (mW)	38.35	42.15	38.45
CB (mW)	N/A	N/A	0.77258
Total Power (W)	1.19	2.08	1.67
Overhead (%)	N/A	74.79	40.34

Table 2: **Hardware Overhead Comparison**

In Table 2, we observe that the UnSync architecture core has a hardware area overhead of only 7.45% (compared to the MIPS core) and occupies 13.32% lesser chip-area when compared to the Reunion implementation. Two significant architectural blocks constitute the area overhead of the Reunion: i) *the additional CHECK stage* (46% in core area) – which is composed of *Fingerprint Generator*, *CHECK Stage Buffer* and the datapath in the register forwarding logic, and *the SECDED protected L1 cache* (7.85% in cache area) – which is composed of additional storage bits (8 *check* bits for every 64 bit data chunk), and ECC generation and verification circuitry, when compared to the MIPS baseline processor. On the other hand, the UnSync implementation is composed of only 17.6% increased core-area and 0.2% increased cache area (1 parity bit for a 256 bit cache-line). In UnSync, the hardware detection blocks added to all the sequential elements in the core, are mostly composed of combinational logic, which can be synthesized optimally for tighter chip-area by the default configurations of the design compiler. On the other hand, the hardware components that contribute to the Reunion area overhead are composed of storage elements, which are mostly regular array structures, that demonstrate lesser flexibility in circuit optimization, for area and power. It should be noted here that, the only additional storage block added to the UnSync architecture is the CB, which has a negligible area overhead.

On similar lines, we observe in Table 2, that the power consumption of the Reunion implementation is a staggering 75% more than the baseline MIPS core. This is mostly due to accesses to the power consuming CHECK stage components (hashing logic and buffer array structure), which consumes 76.8% more core power compared to the MIPS core. In addition, the SECDED generation and verification on every cache access, constitutes around 10% more cache power consumption than the baseline MIPS cache. On the other hand, the hardware detection blocks added to the processor core, only cause around 42% increased power consumption; while the parity bit protection technique employed, does incurs an insignificant power overhead (0.2%). It should be noted here that, the only additional storage structure in UnSync (CB), adds negligible power consumption overheads. We thus demonstrate here through accurate hardware synthesis experiments that the UnSync implementation is of significantly reduced chip-area and low power consumption. Currently the above discussion, is based on worst case analysis with basic design optimizations at the design compiler and power analysis tools. Any circuit level optimization on the detection techniques, or circuit implementations, will only further reduce the overheads incurred.

6.1.2 Projected Scaling of Area and Power Overheads to Many Core Processors

In order to facilitate the design choice of an error-resilience methodology in a many-core system, we compare the die size projections of Reunion and UnSync implementations. For this, the per-core overhead parameters are scaled to existing many-core processors (Table 3), where the original die sizes are obtained from [15]. The overall chip area scales up linearly as the number of cores increase, but the same under the error-resilient architecture

Parameters	**Intel** [13] **Polaris**	**Tilera** [34] **Tile64**	**NVIDIA** [25] **GeForce**
Technology node	65nm	90nm	90nm
No. of Cores:n	80	64	128
Per-core Area (mm^2)	2.5	3.6	3
Original Die Area (mm^2)	275	330	470
Reunion Die Area (mm^2)	316.54	377.85	549.76
UnSync Die Area (mm^2)	289.9	347.16	498.61
Relative difference (mm^2) $DA_{Reunion} - DA_{UnSync}$	**26.64**	**30.69**	**51.15**

Table 3: **Comparison of Projected Die Sizes (DS) of existing many-core processors, in two error-resilient implementations (Reunion and Unsync). While choosing an error-resilient many-core implementation, the last row indicates the difference in die-area between the two choices: UnSync and Reunion.**

implementations does not follow the same trend. To project the die size parameters, we extract the Core Area Overhead–CAO (per core), for each configuration, from Table 2. Since the area overhead of each implementation is observed at the per core level, the increase in area per core (CA_{inc}), is given by $CA_{inc} = n \times CA \times CAO$; where CA is original area of a single core, and n the number of cores, in the processor. The projected Die Area–DA of the processor, in an implementation, is thus given by:
$DA = CA_{inc} + DA_{orig}$; denoted by $DA_{Reunion}$ for Reunion and DA_{UnSync} for UnSync implementations.

The last row of Table 3 denotes the difference in the die areas ($DA_{Reunion} - DA_{UnSync}$) between the two error-resilient implementations of the many-core processor. We use this parameter during design, to determine the right redundancy based mechanism to employ in the many core processor architecture considered. From the projections in Table 3, we observe that:

1. with the increase in the number of cores in the processor, the difference between the die areas of the two implementations increases in a non-linear fashion. In the case of the Intel polaris and NVIDIA GeForce processors, for only around 50% increase in the number of cores, the difference in the die-areas increases by around $2\times$. The higher per-core area overhead of the Reunion implementation (0.2077), as compared

to that of UnSync (0.0745), is the reason for this behavior.

2. the original per-core area is another key factor that governs the total Die Area parameters. In the case of the Tilera processor with 64 cores the per-core area is $3.6mm^2$; which is larger than that of the NVIDIA processor with 128 cores, in the same technology node. In Tilera, the difference between the die areas of the two implementations, is relatively large, when compared to that of the NVIDIA processor.

From the two observations made above, we can deduce that in the design of many-core processors with large number of cores, or when the per-core area is reduced, the UnSync implementation demonstrates reduced power overheads, lower overall die-area, and improved performance.

6.2 Negligible Performance Overhead in UnSync

6.2.1 No Performance Penalty from Serialization

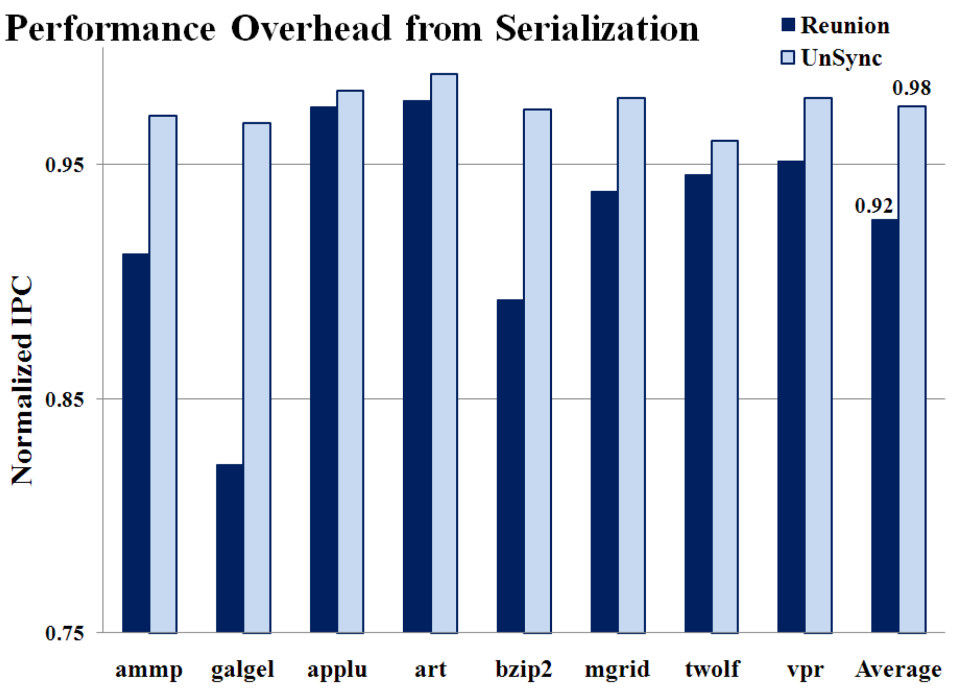

Figure 4: **Reunion is affected by serializing instructions, while UnSync is not. The set of smaller bars on the left, demonstrate performance overheads incurred.**

To show the impact of serializing instructions, we analyze the performance variation on benchmarks which have serializing instructions. The FI in the Reunion implementation determines the granularity or frequency of synchronization and therefore we consider the baseline value for the interval as 10 instructions (smaller the better for Reunion). As we demonstrate here, the UnSync implementation is not affected by serializing instructions, we consider the baseline architecture as described in Section 5. We observe from the results in Figure 4 that, the Reunion implementation incurs an average of 8% performance overhead due to serializing instructions. Applications: *bzip2*, *ammp* and *galgel* suffers from more than 10% overhead because they have more serializing instructions, which are 2%, 1.7% and 1% of total instructions respectively. However, *galgel* also suffers from increased ROB occupancy, and therefore has the maximum overhead. On the other hand, UnSync demonstrates a consistently negligible variation (around 2%) in performance.

6.2.2 No Performance Impact due to ROB occupancy

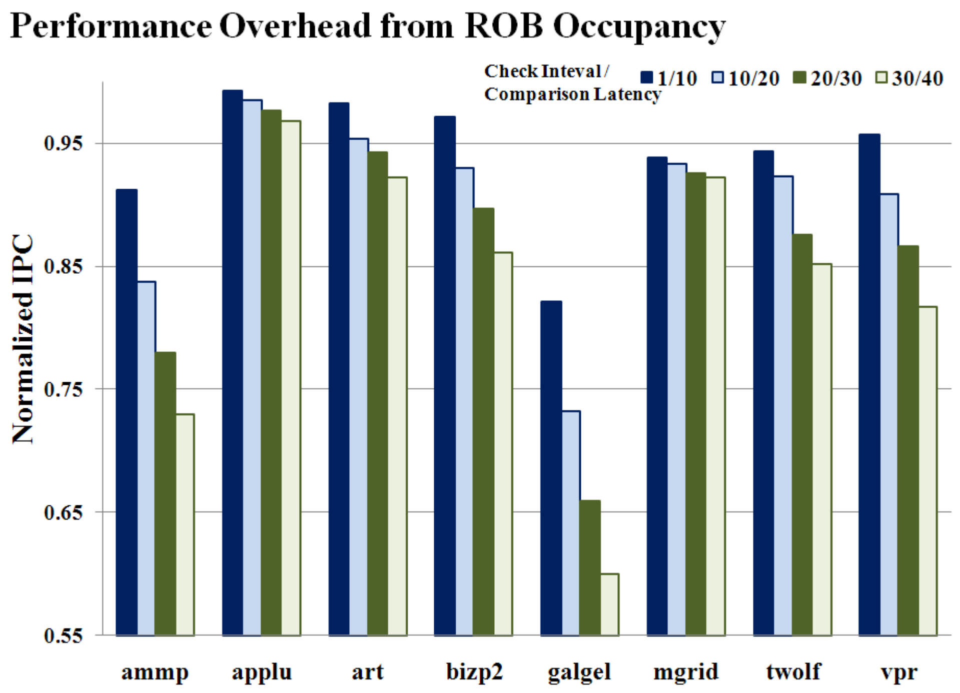

Figure 5: **Reunion suffers from increased ROB occupancy due to the CHECK stage, and varies with the *fingerprint interval* and *comparison latency* parameters.**

In Figure 5 we show how the FI and comparison latency can affect the performance of Reunion. *Fingerprint interval* is the granularity of checking, determined by the number of instructions included in a generated *fingerprint*; while the *comparison latency* is defined as the total time required to generate, transfer, and compare the *fingerprint*. Larger FI has the benefit of less frequent communication and thus less power overhead, but at the cost of increased ROB occupancy; and therefore longer comparison latency since there are more instructions staying longer in the CHECK pipeline stage. Figure 5 shows the performance of Reunion at different comparison latencies and FIs. We start at the FI of 1 instruction and comparison latency of 10 cycles, and then continuously increase them. We can see that *ammp* and *galgel* are greatly affected by the length of the FI and comparison latencies, because the program quickly saturates the ROB. At the FI of 30 instructions and comparison latency of 40 cycles, on an average, the performance decreased by 27% and 41% respectively. In contrast, in UnSync with no synchronization or inter-core comparisons, it is not affected by the increased ROB occupancy since instructions are not held in ROB for an additional time.

6.2.3 Larger CB Size Eliminates Performance Bottleneck

In UnSync, a core with a full CB has to stall and wait for the latest instruction to complete execution on the other CB for comparison and thereby eviction to L2. Therefore, if an application has a large number of write operations, it will affect the performance of system by stalling the cores when the CB is full. Figure 6 shows the performance of UnSync across different CB sizes. We can see that when the CB size is small, the performance decreases; whereas larger CB sizes (2KB and 4KB) completely eliminates the resource occupancy bottleneck, and UnSync has almost identical performance with that of the baseline CMP architecture.

6.3 UnSync Performs better across SER rates

In order to evaluate the two multi-core architectures for their reliability, we extrapolated the average IPC (over the set of benchmarks experimented) for a range of SER rates. For this, we used the exponential ratio of SERs between the technology nodes 180nm (1000 FIT) and 130nm ($100,000$ FIT), and extrapolated the same to obtain the SER for 90nm technology. From experiments performed by iRoc technologies, we observe that the SER

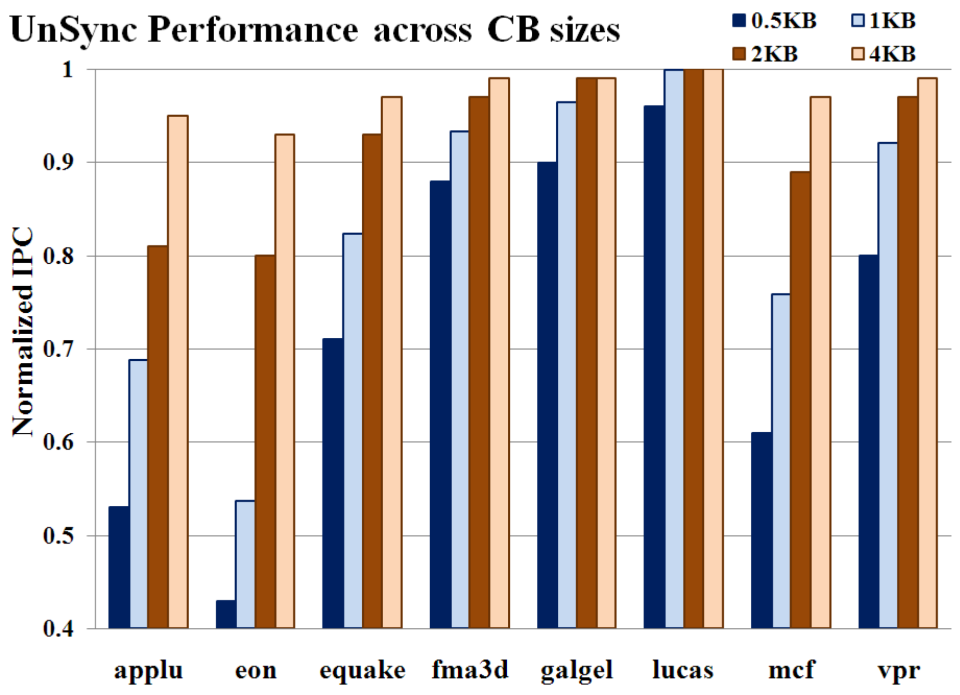

Figure 6: **Larger CB size eliminates the resource occupancy bottleneck and demonstrates better performance.**

rate for technology nodes from 65nm and beyond are more or less saturated and don't follow the same exponential ratio trend. For our analysis, we consider the SER at the 90nm node (2.89×10^{-17} per instruction) [17]. Our projected results of IPC for both the Reunion and UnSync processor architectures does not vary with change in the SER rate from 10^{-7} to 10^{-17} (or lower), thereby demonstrating that the UnSync processor architecture even in the presence of soft errors performs better than the Reunion architecture. A hypothetical analysis, to determine the "break-even" SER that equates the two processors' performance numbers, reveals that when the SER reaches 1.29×10^{-3}, the two processor's will perform alike. Therefore, for all practical purposes, we can safely say that our UnSync technique, performs with 20% better performance with or without soft errors as compared to Reunion.

6.4 Customizable Redundancy with less complexity and larger ROEC

6.4.1 UnSync provides customizable redundancy

To show the performance of our customizable UnSync implementation, over different redundancy configurations, we choose eight benchmarks from SPEC2000 and MiBench. We

Group_0 (Reliability critical)	sha, gsm_encode, gsm_decode, adpcm_endode, adpcm_decode
Group_1 (Reliability non-critical)	facerec, ammp, art, galgel, apsi

	Number of cores in redundant execution	Number of cores in non-redundant execution
setting_0	2	6
setting_1	4	4
setting_2	6	2
setting_3	8	0

Table 4: **UPPER: Benchmarks with different reliability requirements are categorized as group_0, group_1.**
LOWER: Settings for different number of cores used in redundant and non-redundant execution.

divide them into two groups according to our reliability requirement (upper table in Table 4). For example, we require reliable execution for security benchmarks (*sha*) and telecommunication benchmarks (*gsm* and *adpcm*). For the other benchmarks a few errors will not change the results significantly (*facerec, ammp, galgel* and *apsi*), and we can use non-redundant execution for those benchmarks to get higher throughput. Here, we note that there is no strict rule that decides if a workload is reliability critical or not. Choosing the application for reliable redundant execution is user dependent. The lower table in Table 4 describes the different configurations in which the 8 core processor can be set, with varying degrees of redundancy (or error resilience). If the number of cores available for redundant execution are not enough for executing the reliability critical benchmarks in parallel, we run those benchmarks in a round robin fashion.

We measure the performance of different settings using aggregate IPC. Figure 7 shows the system throughput for different settings. For example, in setting_0, 2 cores are used to redundantly execute 4 reliability critical benchmarks in a round robin manner; while 4 out of the remaining 6 cores are used for non-redundant execution of the remaining 4 non-critical applications. Here, two cores that are configured to be used in non-redundant mode are idle and not fully utilized. In Figure 7, we observe that the throughput increases in setting_1 (when 4 cores are used in redundant execution). In setting_2, we use 6 cores for redundant execution, and 2 cores for non-redundantly executing the remaining 4 non-critical benchmarks. A drop in throughput is observed here in Figure 7. When we use all the cores for redundant execution in setting_3, those reliability non-critical benchmarks also run in the redundant mode, and therefore we have the lowest throughput. Setting_3 is similar to the Reunion technique, since all the cores are used for redundant execution.

The design of the UnSync architecture, ensures that all the cores are identical and that pairing of redundant cores is at the conceptual level and only the CB. Only the CBs of adjacent cores can communicate between each other, to only determine completion of an instruction and then perform L2 write. Such a design also allows for quick and easy dis-association of the two cores dynamically through means of BIOS instructions or processor configuration settings. The UnSync architecture, possesses the flexibility to allow dynamic customization of the number and nature of core-level redundancy employed.

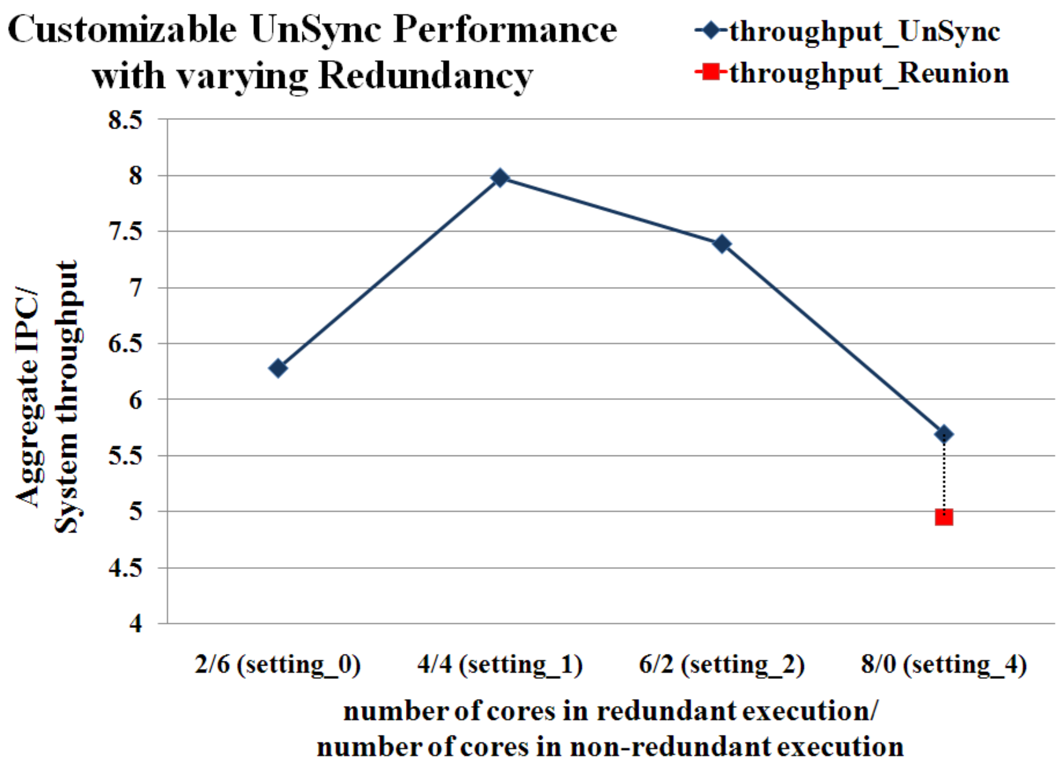

Figure 7: **Performance of customizable UnSync-Arch with different redundancy. We run benchmarks in group_0 in redundant model and group_1 in non-redundant model. Setting_0 does not fully use the cores. Setting_3 uses all cores for redundant execution**

6.4.2 Reunion has lower Region of Error Coverage

Through simulations of the multi-core system in the two configurations, we verify that both UnSync and Reunion architectures execute programs correctly in the presence of errors; though the error detection and recovery mechanisms vary in both techniques. However, the region of error coverage (ROEC) for the Reunion core is limited to the processor pipeline before the "Commit" stage, as the fingerprint verifies only the output data of the instructions after the "Execute" stage. The L1 cache in the Reunion architecture is assumed to have ECC protection and therefore not included in the ROEC. On the other hand, the UnSync architecture includes all the sequential blocks within the processor IP-core and also the L1 cache in its ROEC. We observe that the UnSync architecture achieves same level of reliability, with a larger ROEC, and at lesser hardware overheads of power and chip-area.

7 CONCLUSION

Growing technology scaling expose modern and future processors to soft error failures caused due to charge carrying particles. In this thesis, we propose UnSync– a customizable, error resilient and power efficient multi-core architecture based on core-level redundancy. Our *always forward execution* enabled recovery mechanism coupled with an efficient choice of hardware only detection techniques, reduce performance overheads in redundant designs, while also ensuring error resilience. We, compare our architecture implementation on a multi-core environment, with that of Reunion (a state-of-the-art redundant error resilient technique. Experimental results show that the UnSync achieves upto 20% improvement in performance, with 13.32% reduced area and 34.5% less power overhead when compared to that of Reunion architecture.

With the increasing parallelism in application software, and the drastic increase in the number of cores available in the CMP, our architecture opens doors to varied level of customization and programmability both at the compiler and application level to facilitate development of soft error resilient applications. Since our architecture framework is independent of the underlying architecture within the core, more efficient hardware detection techniques (multi-bit correction for cache blocks, hardened pipeline registers, efficient register file protection, etc.) can be implemented. Our architecture and its working are unaffected by such modifications.

References

[1] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. ISCA, 2007.

[2] P. S. Ahuja, D. W. Clark, and A. Rogers. The performance impact of incomplete bypassing in processor pipelines. MICRO 28, pages 36–45, 1995.

[3] G. Albertengo and R. Sisto. Parallel CRC generation. 10(5):63–71, sept. 1990.

[4] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. 26:52–60, jul. 2006.

[5] Cadence Inc. User manuals for cadence encounter tool set version 09.10-p104, 2009.

[6] K. Chakrabarty and J. Hayes. Test response compaction using multiplexed parity trees. 15(11):1399–1408, nov. 1996.

[7] Compaq Computer Corporation. Data integrity for compaq non-stop himalaya servers, 1999.

[8] A. Golander, S. Weiss, and R. Ronen. Ddmr: Dynamic and scalable dual modular redundancy with short validation intervals. *Computer Architecture Letters*, 7(2):65–68, july 2008.

[9] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. ISCA, pages 98–109, june 2003.

[10] S. Gupta, S. Feng, A. Ansari, B. Jason, and S. Mahlke. Stagenetslice: a reconfigurable microarchitecture building block for resilient CMP systems. CASES, pages 1–10, 2008.

[11] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill. Mips: A microprocessor architecture. MICRO 15, 1982.

[12] E. Ibe, H. Taniguchi, Y. Yahagi, K.-i. Shimbo, and T. Toba. Impact of scaling on neutron-induced soft error in srams from a 250 nm to a 22 nm design rule. 57(7):1527 –1538, 2010.

[13] Intel Corporation. Intel teraflops research chip.

[14] S. Kayali. Reliability considerations for advanced microelectronics. PRDC, pages 99–, Washington, DC, USA, 2000. IEEE Computer Society.

[15] S. W. Keckler, K. Olukotun, and H. P. Hofstee. *Multicore Processors and Systems*. Springer Publishing Company, Incorporated, 2009.

[16] Lattice Semiconductor Corp. ECC module reference design, 2005.

[17] K. Lee, A. Shrivastava, M. Kim, N. Dutt, and N. Venkatasubramanian. Mitigating the impact of hardware defects on multimedia applications: a cross-layer approach. MM ’08, pages 319–328, New York, NY, USA, 2008. ACM.

[18] D. Lyons. Sun screen : Soft error issue in sun enterprise servers, Nov 2000.

[19] P. Meaney, S. Swaney, P. Sanda, and L. Spainhower. IBM z990 soft error detection and recovery. 5(3):419–427, sep. 2005.

[20] K. Meng, F. Huebbers, R. Joseph, and Y. Ismail. Modeling and characterizing power variability in multicore architectures. ISPASS, 2007.

[21] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. ISCA, 2002.

[22] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to model large caches, 2009.

[23] A. A. Nair, L. K. John, and L. Eeckhout. Avf stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors. 0:125–136, 2010.

[24] A. Nieuwland, S. Jasarevic, and G. Jerin. Combinational logic soft error analysis and protection. IOLTS, 0 2006.

[25] NVIDIA Corporation. Geforce 8800, 2011.

[26] R. Phelan. Addressing soft errors in arm core-based designs. Technical report, ARM, 2003.

[27] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. ISCA, pages 25–36, 2000.

[28] A. Shrivastava, I. Issenin, and N. Dutt. Compilation techniques for energy reduction in horizontally partitioned cache architectures. CASES '05, pages 90–96, New York, NY, USA, 2005. ACM.

[29] C. Slayman. Alpha particle or neutron ser-what will dominate in future ic technology, 2010.

[30] T. J. e. a. Slegel. IBM's S/390 g5 microprocessor design. 19:12–23, March 1999.

[31] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. MICRO 39, 2006.

[32] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: bounding soft-error detection latency and bandwidth. ASPLOS-XI, 2004.

[33] D. D. Thaker, F. Impens, I. L. Chuang, R. Amirtharajah, and F. T. Chong. On using recursive tmr as a soft error mitigation technique. 2008.

[34] Tilera Corporation. Many core without boundaries, 2010.

[35] R. Vadlamani, J. Zhao, W. Burleson, and R. Tessier. Multicore soft error rate stabilization using adaptive dual modular redundancy. DATE, 2010.