Latency- and Error-Tolerant Redundant Execution

by

Gordon B. Bell

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN-MADISON

2007

Abstract

This thesis is motivated by two growing problems in computer system design: that off-chip memory accesses dominate overall performance, and that soft (or transient) errors increasingly corrupt data and cause functionally incorrect execution. We propose *skewed redundancy* as a solution to both of these problems by executing applications on multiple processor cores available in an emerging class of chip multiprocessor systems.

Redundant program execution has long been used to detect and correct errors in microprocessors; however, the vast majority of proposals to date add significant checking overhead and suffer from considerable slowdown. This work makes the observation that not all instructions contribute equally to overall performance, and allowing at least one core to skip those that are among the most expensive to execute (i.e. last-level cache misses) can enable cores to become skewed from each other by thousands of instructions. Such “skewing” has a data prefetching benefit by overlapping many independent cache misses and providing significant speedups of up to 2X over non-redundant execution while still providing 100% soft error coverage within our sphere of replication.

In order to compensate for the lost coverage resulting from non-redundantly executing cache misses and their dependent instructions, we leverage existing temporal redundancy techniques that replicate miss-dependent computation within a single pipeline. Finally, we show that our design naturally extends beyond two cores to provide further performance gains while adding error recovery capabilities as well.

Acknowledgements

Although completing a Ph.D. is largely an individual journey, it is rarely possible without the help of many people along the way. First and foremost, I would like to thank my wonderful wife, Tara, for her endless support and patience during my long tenure as a graduate student. She has made tremendous personal and professional sacrifices so that I could complete this degree, and tolerated my long hours and occasional setbacks with minimal complaints. I am happy to join her as “the other Dr. Bell”, and I look forward to starting the next chapter of our life together.

My parents helped me realize the fundamental importance of a solid education, and I am grateful for the freedom and resources they gave me to pursue my interests from an early age. Their encouraging “pep talks” and understanding of missed holidays and family get-togethers made life as a grad student less difficult, in spite of the fact that they have only a vague notion of what computer architecture actually is.

My advisor, Professor Mikko Lipasti, has indelibly shaped my career, both during graduate school and as I move forward in my role as an industrial computer architect. I could not have made it this far without his unconditional support, availability, and eternal optimism. His critical thinking and deep understanding of computer architecture has helped me grow as a researcher and engineer, and I look forward to collaborating with him as a colleague.

The University of Wisconsin computer architecture group has provided a top-notch environment to develop my engineering and research skills, from its insightful and ambitious graduate students to its world-class faculty members. I particularly benefitted from my friendships with Mikko’s prior Ph.D. students: Trey Cain, Kevin Lepak, and

Ilhyun Kim. I will fondly remember many arduous hours together working on course projects, hacking simulators, and participating in many stimulating discussions. I would especially like to thank Trey and Kevin for their tremendous effort in developing PHARMsim into an exceptional research tool, without which I could not have collected the data that appears in this thesis.

Prior to attending the University of Wisconsin, Peter Kogge and Jay Brockman at the University of Notre Dame cemented my early interest in computer architecture and encouraged me to continue my education in graduate school. By welcoming me to participate in their research as an undergraduate, they helped me transition from simply completing course work to developing and evaluating new solutions to engineering problems.

Stephen Stevens provided me with a wonderful opportunity at IBM by hiring me into his group as a summer intern, and was instrumental in my subsequent hiring as a permanent employee. During my internship, my mentor Rick Eickemeyer was incredibly unselfish with his time and helped foster my interest in IBM and product development in general. I am also grateful to my current manager, Ken Vu, for his understanding and flexibility in the overlapping of my final year as a graduate student and my first year as an IBMer.

Computer architecture research often requires large quantities and varieties of computer hardware, and administering these machines is rarely an easy task. Bruce Orchard has gone far beyond the call of duty by providing exceptional and timely support to our research group. I think Kevin Lepak most accurately summarized Bruce's efforts by saying that he would let us do whatever was necessary to get the job done, and yet was always willing to help us put back the broken pieces.

Finally, I thank my thesis committee members for their helpful feedback and interest on earlier versions of this research, and their general support throughout my time at UW.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The von Neumann architecture permeates virtually every aspect of modern computer system design, from basic hand-held calculators to the state-of-the-art supercomputers. Its distinguishing feature is its separation of processing and control elements from a centralized memory used to store instructions and data [96]. In the early years of computer systems this centralized memory could be accessed relatively quickly and was capable of supplying data as fast as processors could consume it. However as high-speed CMOS logic performance eclipsed DRAM fabrication technology, memory took increasingly more processor clock cycles to access. This mismatch between processor and memory speeds became known as the von Neumann bottleneck, and represents the single most significant barrier to increasing single-thread performance. Mitigating this bottleneck is one of the primary goals of this thesis.

At the same time, a combination of decreasing feature size, decreasing supply voltages, high processor frequency, and other factors makes each successive processor generation increasingly susceptible to transient errors caused by stray cosmic rays and alpha particles [101]. Transient (or “soft”) errors occur when a high-energy particle strikes and inverts the logical state of a transistor. They have historically affected memory devices, which can be protected through information redundancy provided by parity or error correcting codes (ECC). However recent studies project that logic and flip-flop circuits may soon be equally susceptible to soft errors as well [78][18][39]. The development of a design that can detecting and tolerate soft errors is a second, parallel goal of this thesis.

Both of these two areas have proven to be popular areas of recent computer systems research and have solicited considerable prior work. In fact, despite their fundamentally different goals (improving performance through reducing memory stalls and tolerating soft errors), they often employ remarkably similar solutions -- namely, redundant execution of a single program thread on multiple processing elements. Running an application on multiple processors can improve performance by overlapping a larger number of long-latency memory accesses, and it can detect errors by replicating computation on physically distinct hardware and comparing results. The cost of allocating multiple processor cores to a single application thread has been significantly reduced by an emerging class of *chip multiprocessors* systems, further adding to the popularity of such approaches.

Despite their similar mechanisms, there has been very little work that considers merging these two goals into a unified technique that achieves both ends. Worse, each goal is often at the expense of the other -- that is, adding error detection capabilities often impedes performance by adding resource contention and introducing checking overhead, and spreading computation across more processing elements increases susceptibility to soft errors.

The objective of this thesis is to utilize redundant execution to simultaneously provide increased single-thread performance as well as error detection. It is successful in this regard, obtaining significant speedups over existing error detection techniques as well as over non-redundant execution. We provide details regarding our proposed implementation of *skewed redundancy* and highlight our solutions to many of the challenging correctness requirements involved. The next two sections elaborate on each of these two design goals, and are followed by a summary of the research contributions made by this work.

Exponential growth in transistor density (commonly referred to as “Moore’s Law” [56]) enabled processor designers to explore alternative microarchitectures to overcome the relatively increasing cost to access main memory (i.e. the “memory-wall”, as it came to be known). On-chip L1 and later L2 SRAM caches provided fast access to memory locations that exhibited temporal and spatial locality. While they decreased the frequency of off-chip DRAM accesses they did nothing to diminish their penalty. This was addressed in the 1980’s and 1990’s with the advent of dynamically scheduled (a.k.a. out-of-order) designs that speculatively executed future instructions while waiting for older memory accesses to complete. Because such designs still typically retire instructions in their original program order, they need to buffer pre-executed instructions until it can be guaranteed that committing their results is safe.

As processor cycle time improvements continued to outpace corresponding reductions in memory latency, this buffering became considerable. Increasingly deeper pipelines exacerbated this effect in two ways: by increasing processor frequency they further penalized cache misses to main memory, and expanding the number of concurrent in-flight instructions resulted in the need for additional buffering resources.

Unfortunately, due to a variety of circuit design constraints, such instruction buffering is neither free nor limitless. In order to stay within successively shrinking clock periods, data structures need to be small and quickly accessible. This puts limitations on the capacity of register files, reorder buffers, memory operation queues, and other related hardware essential to dynamic scheduling. Furthermore, higher transistor density, faster clock rates, and increased static energy leakage have tightened power budgets, making

complex and power-inefficient out-of-order structures less appealing. Consequently, today these structures are nowhere near large enough to completely hide the penalty of a main memory access resulting from a cache miss.

Figure 1-1 provides historical data illustrating the mismatch in improvement rates between processor cores and commodity DRAM memory (adopted from Hennessy and Patterson's fourth edition of *Computer Architecture: A Quantitative Approach* [31]). One series of data points graphs normalized performance of microprocessors shipped over nearly the past three decades. Another shows memory performance assuming a fixed 9% rate of performance improvement per year. Two important conclusions can be drawn from this figure. First, processor performance improved substantially during period between 1984 and 2003, doubling approximately every 18 months. Memory technology improved at only 9% annually (effectively doubling every 10 years) during the same period, leading to a large performance gap between the two. Second, the slope of the trajectory that processor performance tracked decreased to about 20% per year around 2003, largely due to limitations on clock frequency and structure sizes imposed by excessive power consumption. These two trends leave with us today a large processor-memory gap that current microarchitectures are unable to cope with, coupled with an inability to reap the diminishing returns of provided by processor performance alone.

Future trends in systems and microarchitectures will make this problem worse. CMOS logic speeds continue to outpace reductions in DRAM latency. And although Moore's Law continues to enable larger transistor budgets, any attempts to increase processor core size must now compete with designs that utilize excess transistors for additional cores. This has lead to a shift to smaller, simpler cores that are comparatively worse

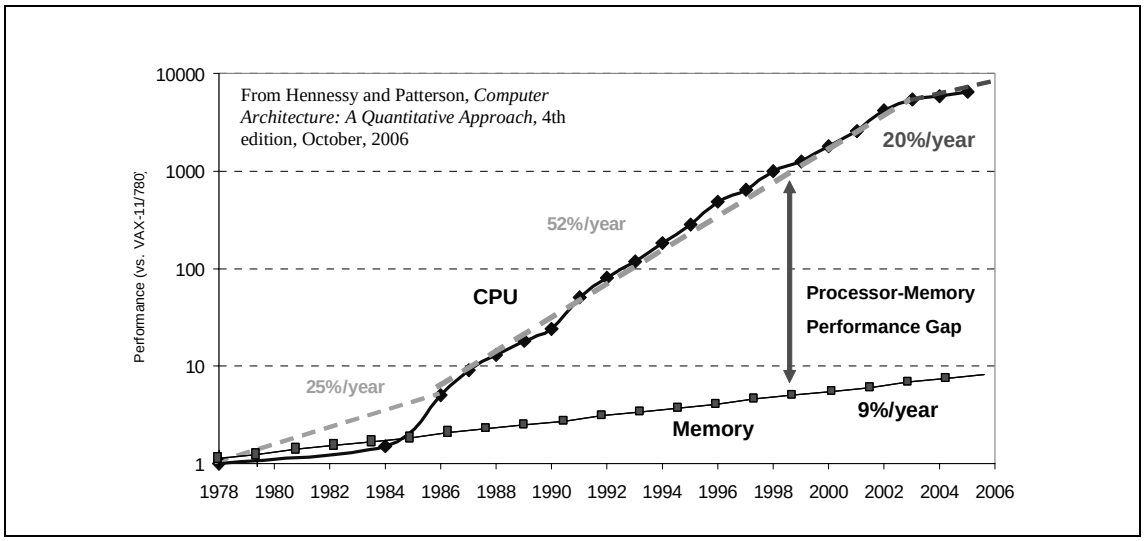

FIGURE 1-1. The Memory Wall. CPU and memory performance over the past 30 years. Note logarithmic scale.

at overlapping memory accesses with other useful work. For example, the Sun UltraSPARC T2 processor provides eight in-order cores that support eight threads each, for a total of 64 simultaneous threads on a single chip. Process technology is expected reach 14nm by 2020 [4], at which point a chip may be able to contain up to 100 cores.

To date, software has not been particularly successful in leveraging the additional processor cores that are now available from all major microprocessor manufacturers. Multithreaded programming is a difficult and error-prone task and despite decades of research and development, there is still a dearth of adequate programming tools and language support to aid general-purpose programmers. This has consequently led system designers to look toward innovative microarchitectures to utilize these new resources.

1.2 Soft Errors

The increased prevalence of soft errors has had a direct effect on processor design, particularly for server products targeting high-availability markets. For example, IBM's recent Power6 processor core implements ECC protection in all levels of cache, parity

protection in nearly all SRAMs, register files and control logic, and additional architecture-level redundancy features such as instruction retry, processor migration, and I/O bit line repair [42].

This increasing vulnerability has similarly propelled error detection and mitigation techniques to the forefront of computer architecture research and has solicited a considerable number of recent publications. Redundant multithreading (RMT) processors execute an identical copy of a program on an unused hardware thread or processor core [72][69]. When they were proposed existing multiprocessors spanned several chips, and some implemented lock-step execution with instruction-by-instruction checking across separate chips [53][77][79]. High inter-chip communication costs made such checking expensive, and these products incurred a significant slowdown. Consequently their use was limited almost exclusively to niche high-availability applications such as banking and stock exchanges. RMT leveraged the decreased communication latency inherent in simultaneous multithreaded processors (SMTs) and chip multiprocessors (CMPs) to demonstrate that error detection could be implemented with a reduced performance penalty and little or no software support. Pushing error detection techniques towards mainstream computing is appealing for several reasons:

- Computers are becoming more ubiquitous and are increasingly used for mission and life critical applications that cannot tolerate hardware failures.
- Circuit technology scaling increases the likelihood of errors due to cosmic rays and alpha particle strikes.
- Higher levels of integration enables chip designs with many cores (each possibly multithreaded), therefore reducing the cost to redundantly execute applications.

Figure 1-2 illustrates the basic concept of RMT. Critical hardware resources within the *sphere of replication* exploit redundancy to guarantee correct execution. Inputs are replicated as they enter and outputs are verified before they exit. Resources outside of the sphere of replication do not employ redundant execution, and must rely on other techniques to provide error detection coverage (for example, through information redundancy implemented by parity or error-correcting codes).

Output comparison in RMT is primarily performed by checking store addresses and data. Once both threads have executed and verified the same instance of a dynamic store, the store is released from the sphere of replication to irrevocably update system memory. Checking only stores reduces checking overhead and enables some amount of "slack" to exist between the threads (unlike true lock-step execution that does not commit *any* instruction until it has been verified), however there will be times when one thread must stall while waiting for verification of one of its stores by the other thread. This results in increased execution time compared to the unreplicated case. Additional slowdown can result from contention between the threads for shared resources such as physical registers, issue window slots, reorder buffer entries, load/store queue entries, execution units, cache bandwidth, and other hardware structures.

1.3 Skewed Redundancy

The primary goal of this thesis is to develop and evaluate methods to increase the slack between redundant threads in order to improve performance while still providing 100% error detection coverage within the sphere of replication. This represents a significant departure from prior work that attempts to merely minimize the slowdown caused by

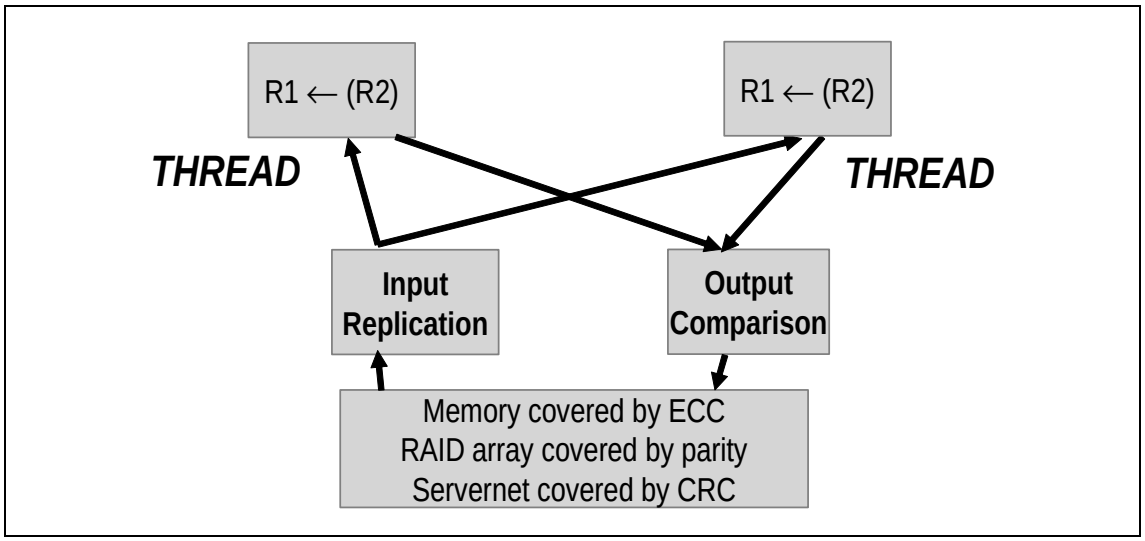

FIGURE 1-2. Chip-level Redundant Threading (CRT). Adapted from Reinhardt et al. [69].

redundant execution.

Even with no checking overhead, no resource contention, and no restrictions on the amount of slack permitted between threads, the performance of RMT is limited to that of unreplicated execution. While the trailing thread can benefit from the early execution of instructions by the leading thread (such as prefetched cache misses and precomputed branch outcomes), this communication is only one way -- the leading thread cannot benefit from the trailing thread's execution. Therefore each thread executes at approximately the same rate, and both threads will identically stall on long-latency events such as cache misses. The threads' similar performance prevents any significant decoupling of their execution from the other.

Skewed redundancy breaks this coupling, and allows one thread to ignore expensive L2 cache misses and rely on a redundant thread to wait for the miss to complete and to execute dependent computation. It utilizes a previously proposed technique [22] to mark the result of the load miss as invalid, and propagates information regarding dependence on that load through the register file and cache hierarchy. This enables that thread to

continue executing and uncovering future cache misses while the redundant thread is stalled. Figure 1-3 depicts two redundant threads executing the same instruction stream. It illustrates that each of them has become “skewed” with respect to the other, resulting in a single large virtual instruction window capable of overlapping many concurrent cache misses.

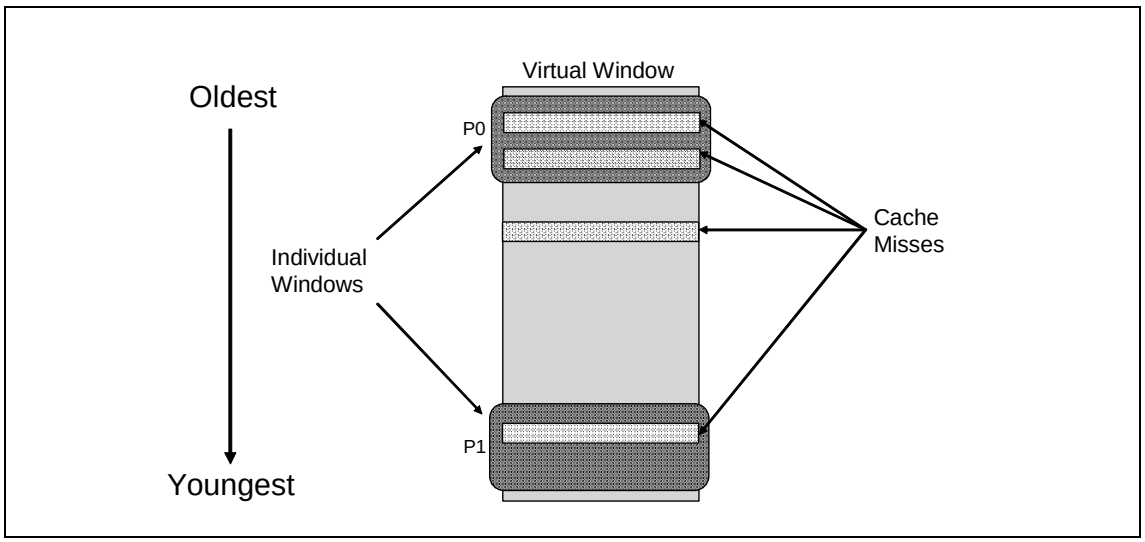

FIGURE 1-3. A Large Virtual Instruction Window. Two redundant threads that have become skewed with respect to each other. Each overlaps instructions with its local instruction window, while cache misses are distributed across both cores and form a large virtual window

Instructions skipped by either thread no longer exhibit spatial redundancy, and must therefore rely on an alternate approach to provide 100% soft error coverage. This research shows that a second technique that exploits temporal redundancy by replicating instructions within a thread is particularly well-suited to fill this gap in coverage.

1.4 Thesis Contributions

The research presented in this thesis makes the following contributions:

- **Decoupling of redundant threads:** We propose a mechanism to allow execution of redundant threads to become “skewed” with respect to each other, preventing

stalls in one thread from impeding progress by the other.

- **Development of "skewing" mechanism:** We develop a mechanism to take advantage of the decoupling between redundant threads to enable one to continue executing instructions while another is stalled waiting for a cache miss.
- **Elimination of input replication:** We propose a simple technique to address the "input incoherence" problem without the need for an explicit load-value queue (LVQ) to satisfy load requests of the redundant thread. Other work has contended that the LVQ adds significant complexity to existing designs [80]. We also show how this technique can be used to enforce multiprocessor load-to-load coherence.
- **Selective application of redundant instruction issue:** Skipping expensive cache misses allows threads to become skewed with respect to each other, but sacrifices redundancy and soft-error coverage. We selectively apply an older technique that replicates instructions *within* a single thread, and show that it can compensate for this loss in coverage without adversely impacting performance.

1.5 Thesis Organization

This thesis is organized as follows: Chapter 2 presents a detailed overview of existing approaches that achieve each of our two goals in isolation: enabling large instruction windows, and soft error mitigation techniques. Chapter 3 discusses the experimental methodology used to collect the data that appears in this thesis, including PHARMsim, the PowerPC-based full-system timing simulator utilized in this work. Chapter 4 develops the correctness requirements involved in partitioning a single program thread across multiple processing elements, and also provides details on our proposed implementation to enforce

these requirements. Chapter 5 extends our implementation to additionally incorporate error checking and recovery. A detailed experimental evaluation of skewed redundancy is presented in Chapter 6. Finally, Chapter 7 concludes this thesis and present additional avenues for future research.

Chapter 2

Related Work

This chapter is divided into three sections. The first section addresses proposals that improve single-thread performance through large *virtual instruction windows* designed to uncover and overlap additional cache misses. It places particular emphasis on work that implements such windows with additional processor cores. The second section considers proposals that use coarse-grain redundant execution to detect and tolerate soft errors. It discusses some seminal publications in this area as well as reviews subsequent work that aimed to minimize checking overhead. The third section discusses the relatively few proposals that combine both of these techniques to simultaneously achieve error detection and increased performance. The research presented in this thesis falls into this third category.

2.1 Performance

2.1.1 Motivation: The Demand for Large Windows

To completely hide the latency to access main memory on a cache miss, an out-of-order processor must have sufficient buffering resources to fetch and execute younger instructions while the miss is pending. Typically, this implies that each instruction consumes a reorder buffer entry, physical result register, and load/store queue entry (in the case of a memory operation) from the time it issues until it retires. These resources are collectively referred to as the *instruction window,* and place an upper bound on the maximum number of in flight instructions that the processor can concurrently track and execute.

Exhausting these resources before the miss returns will result in a machine stall until the miss completes and instruction retirement continues.

Table 2-1 summarizes the out-of-order capabilities of several currently-shipping processors. The first column indicates the number of processor clock cycles required to service a last-level cache miss assuming a typical 100ns DRAM read time. The second column indicates the maximum number of instructions that can be issued in a single cycle. The product of these two parameters indicates the maximum number of subsequent instructions the processor is capable of reaching while waiting on the miss. Instruction windows smaller than this may result in lost execution opportunities and machine stalls. The final column shows the actual number of instructions that can co-exist in the out-of-order window. In all cases there is a significant mismatch between the number of instructions a processor has the time and execution bandwidth to reach, and the number of instructions that it has the capability to buffer. Consequently, applications with large working sets and poor locality tend to spend considerable time stalled on cache misses.

Table 2-1: Instruction Execution Bandwidth and Window Size of Modern Microprocessors.

	Memory latency (cycles / 100ns)	Superscalar width (instructions / cycle)	Instruction window (instructions)
Intel Core (3GHz)	300	3 uops[1]	96
AMD Hammer/K8 (2.5 GHz)	250	3 uops	72
IBM Power4/5	220	8	100
IBM Power6	470	7	in-order
Sun UltraSPARC T2	140	1	in-order

1. The Intel Core microarchitecture can send up to three mico-ops to the ROB per cycle. The actual amount of work this corresponds to depends on the particular x86 to micro-op translation, as well as its effectiveness of fusing multiple micro-ops together into a single instruction.

To maximize the number of processor cores that can be placed on a single chip,

designers strive to optimize the area and power efficiency of each individual core. Several current and forthcoming products have therefore abandoned the complexities of out-of-order execution and adopted simple in-order cores. We point these out in the right-most column. While such designs typically tolerate execution stalls by exploiting parallelism across multiple threads, rather than across instructions within a thread, each individual thread suffers enormously from cache misses due to its inability to overlap *any* younger instructions. Finally, we point out that some of the data in this chart is simplified for the sake of clarity -- for example, dispatch and issue restrictions may limit the number of instruction of a given type or of a given thread. However the general conclusion that threads are incapable of fully tolerating cache misses with subsequent computation is clear.

Clearly modern processors have insufficient dynamic scheduling resources to fully hide main memory accesses. A reasonable question might ask why designers simply do not scale these resources to accommodate enough instructions to prevent stalls. As deeper pipelines enable faster processor frequencies, the number of logic levels that can be traversed in a single pipeline stages decreases [34]. Many out-of-order structures are inherently complex due to content-associative indexing as well as higher degrees of multiporting required by wider superscalar widths and multithreading. Therefore further increasing their size would likely require either extending the clock period or pipelining their access. Neither of these alternatives is particularly appealing.

2.1.2 Extending Instruction Windows

Now that we have motivated the desire to support large instruction windows and highlighted the difficulty in building them, we present a brief survey of some recent

research and discuss their approaches to overcome the memory wall.

The problem of finite sized instruction windows in the face of a growing processor-memory gap is a substantial problem in modern computer architecture and has garnered considerable attention. Novel schemes such as early reclamation and reference counting of critical instruction window resources have been proposed in a number of studies in order to achieve a large virtual window without the physical design challenges of a large physical window [9][20][21][52][88]. However, dedicating significant additional design complexity and/or substantial hardware resources to the sole task of tracking a large number of instructions from a single thread is difficult to justify, particularly since future designs are likely to be limited by thermal and power considerations. At the same time, the increasing prevalence of thread-level parallelism has led to the development of chip multiprocessors. Any attempt to allocate die area or design time for single-thread performance enhancements like large (virtual or physical) instruction windows must now compete with a demand for additional processor cores on the same die, since those additional cores provide attractive levels of performance for many important workloads.

Researchers have therefore turned towards utilizing CMP cores to accelerate single-threaded programs. However the challenges of maintaining a single virtual instruction window across multiple processing elements are numerous. Prior proposals have leveraged compiler support to break sequential programs into speculative threads, and to use varying levels of hardware support to guarantee correct sequential execution semantics for those threads, ranging from fairly simple coherence protocol extensions in the thread-level speculation work [89][44][30] to extensive support for resolving both register and memory dependences in the Multiscalar proposal [85]. We advocate an approach that maintains

binary compatibility and trades computation for communication and complexity: rather than attempting to precisely partition a program into completely disjoint threads, we partition the work only when cache misses cause a reasonably-sized instruction window to fill up and stop making forward progress. To enable this approach, we redundantly execute most instructions on all processing elements to minimize communication and synchronization. This simplifies the tasks of maintaining precise exceptions, enables a very simple algorithm for partitioning work, and allows us to utilize existing CMP resources to extract much higher single-thread performance and error detection capabilities out of a single chip.

Other work has proposed *leader/follower* architectures, where a program is redundantly executed on two threads: a leading thread that executes only a subset of the instruction stream, and a trailing thread that verifies results computed by the leader. The leader's performance is improved by skipping certain instructions, and the trailing thread's performance is improved from prefetches and potentially branch prediction hints generated by the leader.

Two primary instruction-skipping mechanisms have been proposed to accelerate the leader. Slipstream processors [91] omit instructions predicted to have no effect on program outcome (dubbed "ineffectual" [73]) and rely on the follower to verify its predictions. Examples of such instructions include silent stores [48], dynamically dead code [12], and backwards slice instructions of predictable branches. While this technique can achieve modest speedup from the aggregate removal of ineffectual instructions, the instructions themselves are not inherently expensive to execute.

Later leader/follower proposals targeted those instructions that are among the most

expensive to execute: last-level cache misses [100][91][8][24][64]. They allow the leader to skip cache misses and their dependent operations by using a technique first proposed by *runahead execution* [22][60], in which cache misses are identified, discarded, and propagated through register and memory dependences. This allows the leader to advance much further ahead and generate more effective prefetches. The trailing thread becomes responsible for identifying cases where discarded load misses caused the leader to diverge from the correct program control path, at which point it squashes and restarts the leader.

Our work is fundamentally different than these approaches. *Skewed redundancy* follows a similar model of selectively skipping cache misses and dependent instructions, however it does so *non-speculatively.* Non-speculative instruction commit is desirable because it enables soft-error detection by checking each instruction stream against the other. Checking against speculative instructions is problematic and can lead to silent data corruption in the case where an error and misspeculation produce identical incorrect results.

2.2 Error Detection and Tolerance

One method to detect errors is to introduce redundancy into a design. Because a single event upset will either affect the original instance or the replicated instance (but not both), comparing these two forms with each other can catch single-bit errors. This section highlights three types of replication that appears in existing literature: information redundancy, space redundancy and time redundancy.

2.2.1 Information Redundancy

Information redundancy stores an additional copy of state to be protected, often in

compressed form. Common implementations include parity (to detect errors) and Hamming codes (to detect and correct errors). Because it requires some space and computational overhead, it is most commonly implemented in memory structures where its cost can be amortized over large arrays. It is not usually implemented in processor datapaths; instead, space and time redundancy are alternatively used for this purpose.

Recent research has examined the use of information redundancy to verify the correct operation of simple in-order cores. Argus [54] separately checks four high-level invariants of von Neumann architectures: correct control flow, dataflow, computation, and memory access. Similar to other forms of information redundancy, it compresses information into compact signatures (e.g. control and data flow signatures) that it independently checks off of the critical execution path. Although Argus is effective at detecting a large fraction of errors at a low overhead, it cannot achieve performance speedups over unreplicated execution.

2.2.2 Space Redundancy

Space redundancy concurrently performs identical computations on multiple hardware, and detects errors by comparing independent results. A popular and conceptually simple implementation of it is lock-step execution [53][77][79], which duplicates the entire processor pipeline and compares each instruction result before it commits and updates architected state. Spatial redundancy has also been widely used in lower-level circuit implementations, where a signal fans out to multiple independent logic gates and is transparently checked for errors [1].

Despite successful commercial implementations, spatial redundancy often comes at a significant cost. Not only does it require additional dedicated hardware to perform the

redundant computations (either multiple processors in the case of lock-step execution or larger cores to accommodate duplicate logic paths in circuit redundancy), but it can also degrade performance (by either increasing instruction latencies to perform a round-trip check with another processor, or by increasing clock periods to perform circuit-level signal comparisons).

2.2.3 Time Redundancy

Time redundancy techniques address this additional hardware overhead by re-using the same hardware to redundantly compute a result. Increasing transistor density has enabled aggressive microarchitectures capable of executing multiple instructions per clock cycle. However conventional programs often exhibit limited data parallelism and are ill-suited to extract high-performance from superscalar hardware [97][6]. Much research has focused on exploiting this inefficiency by utilizing spare processor cycles to re-execute instructions and provide soft error tolerance.

Sohi et al. applied Recomputing with Shifted Operands (RESO) [84] to pipelined functional units and was among the first to demonstrate that error detection can be provided with reasonable overhead. This technique was later applied to superscalar processors by Ray et al. [67] and extended to include soft error recovery as well as detection.

O3RS [55] aimed to reduce the penalty of executing additional instructions by sharing issue queue and register update unit (RUU) entries between replicated and non-replicated instructions. Other work has similarly observed that redundant instructions do not require explicit issue queue storage [74][35]. The *SHREC* microarchitecture [82] adopted a similar approach but added a dedicated in-order issue queue for the redundant thread to allow it to stager its execution with the main thread. Redundant instructions use

input operands generated by the main thread, and therefore do not require the flexible scheduling provided by dynamic issue. The Ditto processor [46] also followed this model, but exploited the lower checking overhead of short-latency instructions and proposed different verification mechanisms for short and long-latency instructions.

Time redundancy has also been applied at the circuit level to provide soft-error tolerance. Nocolaidis et al. embedded a delay element between combinational logic and pipeline latches to duplicately latch data at different times [62]. However because this redundancy occurred within a single clock cycle, adding such delays are likely to increase clock period of modern highly-pipelined designs.

Most of these techniques provided *instruction-level replication* by duplicating individual instructions at the issue or execute pipeline stages. Subsequent research exploited emerging designs centered on *simultaneous multithreading* [94] and *chip multiprocessing* [63], which allowed program replication at much coarser granularities. CMPs shifted error detection techniques away from the previous focus of purely time-redundant (where re-execution occurs on the same hardware at different times) to purely space-redundant (where re-execution occurs on physically separate hardware). SMT provides a unique hybrid approach that uses both time and space redundancy. Wide-issue superscalar designs common in SMT can schedule redundant instructions on the same functional unit at different times, or on separate functional units. This new model of error detection is the foundation of this thesis, and we discuss its implementation in a following chapter.

2.2.4 Redundant Multithreading

Utilizing SMT processors for error detection was first proposed by Saxena and McCluskey [75]. It was further developed and evaluated by Rotenberg as AR-SMT [72],

which redundantly executes a single software thread on two hardware threads: an “active”, or A-thread, and a “redundant”, or R-thread. Committed results from the A-thread are passed to the R-thread via a *delay buffer* where they are verified. The delay buffer additionally helps to speed up execution of the R-thread by prefetching cache misses and forwarding performance hints (such as computed branch outcomes).

Redundant multithreading (RMT) [69] generalized AR-SMT’s approach to both SMT and CMP processors and demonstrated that only checking store addresses and values is sufficient to detect soft-error. Output comparison is performed by a *store comparator* (SC) that buffers completed but not yet committed stores. The first thread to reach a specific dynamic store inserts the store address and data into the SC. After the second thread executes the same dynamic store it similarly accesses the SC and compares its store address and data. Once a store has been successfully verified it is eligible to commit in each thread, and a single instance of the store is released from the sphere of replication to the memory hierarchy.

2.2.5 Simultaneous and Redundant Threading

RMT was originally proposed in the context of simultaneous multithreading (initially called *simultaneous and redundant threading*, or SRT [69]). It outperformed previous implementations of lock-stepped execution for two reasons. First, only checking stores decoupled the execution of each thread from the other and enabled a small amount of “slack” to exist between them. This avoided stalling every instruction until it could be verified. Second, executing both threads within an SMT dramatically reduced the store verification latency compared to separate processors on physically separate chips.

Despite improved performance over existing techniques, SRT still suffers signifi-

cant slowdown compared to unreplicated execution. The redundant thread places additional pressure on critical shared microarchitectural resources such as the issue window, reorder buffer, load/store queue, physical register file, fetch unit, and execution units. This increased contention alone is responsible for a reported average slowdown of around 26%. Additional resource pressure from extending all store lifetimes (to an average of 39 cycles) further degrades performance to 30%.

2.2.6 Chip-Level Redundant Threading

The SRT paradigm 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) [58] 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. However CMP cores are not as tightly integrated as SMT threads, and the additional physical separation increased the round-trip store verification latency and subsequently increased the average store lifetime from 39 to 69 cycles. This latency dominated any gains reaped from additional hardware resources and resulted in a net slowdown compared to SRT. Nevertheless, spreading computation across multiple cores provides a higher level of redundancy and reduces the risk that a soft error will adversely affect both instances of a single instruction, and therefore remains a viable alternative to SRT. Its authors show that some of this additional overhead can be mitigated by "cross-coupling" logical threads in multithreaded workloads. This technique pairs leading and trailing threads of different applications on the same processor, and is beneficial because each trailing thread generally consumes less resources due to prefectching and branch predictor hints generated by its corresponding leading thread.

Other designs utilize specialized checking processors instead of general-purpose processing cores. *Watchdog processors* [51] are simple coprocessors designed to detect errors by monitoring a system. DIVA [5] is one example of this model, and uses a simple in-order checker to verify the execution of an aggressive out-of-order superscalar processor. It propose to detect and correct design faults (in addition to soft errors), which it asserts are increasingly frequent as processors gain complexity in the face of limited design resources and aggressive design times [29]. However despite their simplicity, such processors nevertheless require added design and verification effort, consume additional die area, and offer limited flexibility. It is therefore often preferable to map redundant execution onto highly-replicated and under-utilized processor threads or cores.

2.2.7 Improving RMT

A large body of subsequent work has focused on improving the original RMT proposal, and can be divided into three broad categories: work that adds error recovery capabilities to RMT, work that aims to reduce the checking overhead of RMT, and work that simplifies the hardware implementation of RMT.

2.2.7.1 Error Recovery

Because both SRT and CRT only detect an error after it has propagated to a store, they cannot easily dynamically correct the error and continue. *Simultaneous and Redundantly Threaded processors with Recovery* (SRTR) [95] extend SRT to provide error recovery by checking all instructions instead of only stores. In order to reduce this additional overhead, it proposes to pipeline instruction verification by checking instructions as they complete rather than when they commit. SRTR only slightly degrades SRT performance, however it adds additional complexity to support speculative instruction checks.

SRTR is effective because it can generally hide instruction verification latency within an instruction's complete-to-commit time. The physical separation of threads in CRT increases the round trip verification latency and limits inter-core bandwidth; therefore SRTR cannot be applied to CMPs without significant slowdown. *Chip-level Redundantly Threaded multiprocessors with Recover* (CRTR) [26] addresses this problem in two ways. First, by checking only the final instruction in register dependence chains it reduces the checking frequency and required bandwidth between cores. Second, it decreases the checking penalty by allowing the leading thread to commit non-store instructions before they are verified. If the trailing thread detects an error it interrupts execution and copies its uncorrupted register state to the leading thread.

2.2.7.2 Reducing overhead

Other work has focused on reducing the overhead of redundant multithreading. Smolens et al. classified sources of resource contention between threads [82] and identified specific bottlenecks through a survey of existing error-tolerant microarchitectures. *Opportunistic Transient-Fault Detection* [27] employs partial explicit redundancy (PER) by applying conventional instruction replication during periods of low ILP, while providing the main thread unrestricted access to execution resources during periods of high ILP at the cost reduced soft-error coverage. To compensate for this reduced coverage, it proposes a light-weight error checking mechanism based on *instruction re-use* [83] for periods of high-ILP. While this technique can provide higher performance than the original SRT proposal, it suffers from reduced error coverage because instruction re-use cannot detect errors in front-end pipeline stages.

Microarchitecture-Based Introspection (MBI) [65] reduces resource contention by

executing redundant thread instructions while the main thread is stalled on L2 cache misses. During high-ILP periods the main thread operates in *performance mode* and commits unverified instructions into a *backlog buffer*. When a cache miss occurs or the buffer fills the redundant thread processes the buffer contents. This effectively applies a form of course-grained multithreading [90] to error detection techniques; baseline microarchitectures that already support simultaneous multithreading can likely benefit from the finer-grained resource sharing that it provides. Another key difference between MBI and other SRT work is that stores in SRT update the L1 only after they are successfully verified. Errors in MBI stores are not detected until they are processed from the backlog buffer, possibly thousands of cycles after the main thread committed the erroneous store and updated the L1. This complicates recovery by requiring memory state to be checkpointed in addition to register state. An alternative proposed by the authors is to augment the backlog buffer with store values and addresses and incrementally roll back memory state when an error is detected.

2.2.7.3 Implementation

Mukherjee et al. provided a detailed discussion of the hardware requirements to implement SRT into existing SMT processors [58], and acknowledged several subtle design issues not initially considered. Since then, some researchers have focused on simplifying the SRT model to ease integration into existing designs. Reunion [80] points to the load-value queue (LVQ) as one source of complexity, and proposes to relax strict input replication by allowing the redundant thread to issue loads directly to the memory system. 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. This will result in identical load values the vast majority of the time; the remaining cases are handled by issuing the loads a third time via conservative synchronizing memory requests that eliminate input incoherence for the requested cache line. The authors report an average 5% performance penalty compared to strict input replication. The rarity of remote invalidates matching in flight load addresses has been previously noted elsewhere [14].

Fingerprinting [81] is a checkpointing scheme designed to minimize changes to commodity hardware. Processor pairs identify errors by comparing cryptographic signatures that summarize architected 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 state (including memory) and increase error detection latency over purely microarchitecture-based techniques. Finally, hybrid hardware-software designs have been proposed to further reduce error detection implementation costs [70].

2.2.8 Backward Error Recovery

Of the techniques discussed so far, most of those that can correct errors use forward error recovery (e.g. lock-stepped execution, SRTR, CRTR, etc.). This requires that they can precisely identify erroneous instructions and correct or re-execute them before they commit and update architected state. Other work relaxes this time frame, and permits errors to be detected after they commit. Since committing bad values can corrupt the machine state, these techniques rely on backward error recovery to restore state from periodic checkpoints taken throughout a program's execution [28][87][36][7].

A good example of a backwards error recovery design is SafetyNet [86], which, targets shared-memory multiprocessor systems and supports globally-consistent checkpoints. This enables it to detect errors that occur outside of the processor cores (e.g. failed interconnection switches or dropped coherence messages); in this sense, it provides a complementary sphere of replication to skewed redundancy, which focuses on errors within the core pipelines. A comprehensive design could include both of these orthogonal techniques to further extend error coverage.

2.3 Multiple Cores for Both Error Detection and High-performance

Very few proposals attempt to simultaneously achieve the combined goals of both increased performance and soft-error detection. The authors of dual-core execution (DCE) recently extended their original design to include error checking capabilities [50], but there are several important distinctions between their proposal and ours. Because DCE's back core does not explicitly fetch instructions from memory and instead consumes them from the front core via a *result queue*, its sphere of replication does not encompass the instruction fetch unit. They propose to protect the front core's fetch unit with ECC. While this may detect errors in the fetch datapath, it is less clear if this can be used to detect transient errors that occur within the fetch logic as well. All instructions and their results pass through the result queue: this structure therefore needs to be large enough to accommodate all instructions within the virtual window (other than those actively executing in either core), and it needs to be fast and multi-ported because instructions are read and written at the peak instruction bandwidth of the machine. Skewed redundant cores use dedicated fetch units, adding a degree of error coverage and avoiding the need to buffer instructions

across cores. Furthermore, the number of instruction results that are buffered across cores is also minimized. Chapter 6 shows that our *global reorder buffer* with as few as 32 entries is sufficient to enable virtual windows spanning thousands of instructions, reducing the space needed for buffering results by two decimal orders of magnitude. Another significant difference is that the DCE front core commits instructions speculatively and thus requires a separate runahead cache for passing results through memory dependences and high performance. Speculatively committing instructions also reduces error detection coverage because an error in the back core may be masked by a mispredicted branch or invalid store address in the front core. Finally, DCE is limited by design to two cores, while our approach generalizes elegantly to three or more cores to enable triple-modular or even greater levels of redundancy. Slipstream processors were also extended to detect soft errors [68], however this proposal provides less than 100% coverage.

Chapter 3

Experimental Methodology

This chapter details the experimental methodology used to collect the data that appears in this thesis. It begins with a description of the simulation environment on top of which we implemented skewed redundancy. Next we present relevant microarchitectural parameters of our baseline machine model, as well as those specific to skewed redundancy. We conclude this chapter with information regarding the applications used to benchmark our implementation, followed by a brief discussion and data related to our baseline hardware prefetching mechanism.

3.1 Simulation Environment

All of the data presented in this dissertation was collected with an architecture-level simulator built on top of PHARMsim, an execution-driven full-system timing model of a system that implements the full PowerPC instruction set [14]. PHARMsim was developed by the PHARM research group at the University of Wisconsin, and inherits substantial portions from both the PowerPC port of SimOS [71] and SimpleMP [66], which is itself originates from SimpleScalar [11]. PHARMsim models all aspects of PowerPC multiprocessor systems, including cache-coherent shared memory, out-of-order superscalar processors, functionally correct execution of all user and system-level instructions, asynchronous interrupts, and all aspects of address translation including hardware page table walks and page faults. It boots and runs the AIX 4.3 operating system.

We implemented skewed redundancy and its related hardware structures on top of

this infrastructure as they have been described in this thesis. This required a number of substantial modifications and enhancements. PHARMsim was originally developed to model a symmetric multiprocessor (SMP) where each processor has a private L1, L2, and L3 cache. Because we propose skewed redundancy in the context of a chip multiprocessor (CMP) with private L1s and a shared L2 and L3 cache, the first step in its implementation involved allowing processor cores to share portions of the cache hierarchy. Normally this would involve shifting the coherence point from the shared L3 to the private L1 caches. However, because only a single program thread executes at any time and stores originate from a single source (the *global store queue,* or GSQ), the L1s do not need to be kept coherent with each other. In fact, coherent L1 caches would be detrimental to performance -- since cores execute identical instructions at nearly the same time, their stores would continually invalidate cache lines in other L1s and substantially increase the L1 miss rates.

The second step involved changing the way in which simulator checkpoints are mapped onto processors. PHARMsim uses multiprocessor checkpoints created by SimOS-PPC that encapsulate the entire architected state of a system including shared memory and processor registers. It creates a one-to-one mapping of checkpointed processors to simulated processors -- for example, a 4-processor checkpoint contains the state of four separate processors, which are restored to a 4-processor simulated machine. From an operating system's point of view, skewed redundancy appears as a uniprocessor; therefore, a single uniprocessor checkpoint needs to be restored to multiple cores, each of which begins fetching instructions from the same starting instruction address.

The third and final step forces multiple cores to execute the same program while still maintaining the appearance of a single high-performance and error-tolerant unipro-

cessor. This is not trivial in an execution-driven simulator such as PHARMsim that implements value-passing through registers and memory, execution of system instructions and OS code, instruction cracking, address translation, and a multitude of other challenging artifacts that are present in real systems and architectures. Maintaining this image is further complicated by the fact that each processor only executes a subset of the entire program -- all cores discard some cache misses and their dependent operations. Our implementation mirrors what actual hardware would do: a single program thread is mapped onto multiple cores with instruction windows of finite capacities; as those cores commit store instructions they are inserted into a *global store queue* (described in the following chapter) and eventually released to the shared L2 in program order. L1 cache misses need to snoop the GSQ to identify any pending stores before accessing the shared L2. Cores advance ahead of each other by discarding last-level cache misses and proceed without the correct load value.

Implementing our simulator in a way that mimics real hardware provides an additional degree of fidelity in the accuracy of our design and performance results. For example, if a bug in the simulator caused a core to continue executing without adhering to the synchronization conditions presented in Chapter 4.3, it would likely veer off the correct execution path and eventually be detected when it mismatches another core's global store queue (GSQ) or global reorder buffer (GROB) entries. Or if memory dependencies were mis-identified through the private L1s, shared GSQ, and shared L2/L3, a load would receive an incorrect value and change the program's execution. These examples and others occurred in the process of implementing this research; had we used a trace-based or analytic model as our infrastructure, these bugs may have never been detected and resulted in

inaccurate results or an incomplete design. As a final enhancement, to provide further confidence in the functional correctness of our design, PHARMsim supports the use of an additional “functional-only” checking processor that compares the result of every instruction to detect mismatches. We utilized this checker to verify that all simulations presented in this thesis followed the same execution path as the functional-only model.

3.2 Machine Model

PHARMsim supports a large number of configurable microarchitectural parameters that control the timing of processors and their memory requests. Table 3-1 presents the baseline machine model that skewed redundancy is built upon.

Table 3-1: Machine Configuration.

Attribute	Value
Processor Clock	3.0 GHz
Pipeline Depth	12 stages
Fetch Queue Size	16
Branch Predictor	combined bimodal (16k entry)/gshare (16k entry) with selector (16k entry); 64 entry RAS; 8k entry 4-way BTB
Decode/Issue/Commit Width	4
Issue Window Size	16
Reorder Buffer Size	64
Load/Store Queue Size	32
Memory Dependence Predictor	4k entry Alpha-style dependence predictor [41]
Functional Units (latency)	4 INT ALU (1) 2 INT MUL/DIV (4/4) 2 FP MULT/DIV (4/4) 4 L1D memory ports
L1 Caches (private)	Instruction: 32KB, 2-way, 32B lines (1 cycle) Data: 32KB, 2-way, 32B lines (1 cycle)
L2 Cache (shared)	256KB 8-way unified, 32B lines (15 cycles)
L3 Cache (shared)	1MB 8-way (50 cycles)
TLB	2k entry, 2-way, instruction + data hardware page table walker 4KB pages
Memory	256MB 300 cycles/100ns best-case latency exclusive prefetching (stores prefetch at issue) stride-based prefetcher modeled after Power4
Skewed Redundancy (latency)	128 entry GROB (10) 128 entry GSQ (15) additional pipeline commit stage to insert store into GSQ

It is representative of contemporary CMP systems available today; we also believe it will be a close match to future systems as well. Historically, higher transistor density enabled by decreasing feature sizes has lead to more aggressive processor cores in each successive technology generation. However, dramatic increases in power consumption have resulted in diminishing returns on single-thread performance, and have generated a trend toward designs that favor more cores over more complex cores. For this reason we contend that future microarchitectures will not be significantly more complex than they are today. In fact, one could argue for the reverse trend -- that designs are returning towards simple in-order cores, many of which can fit onto a single chip. This design philosophy is exemplified in two currently-shipping systems composed of highly-threaded in-order cores: the Sun UltraSPARC T1/T2 (a.k.a Niagara 1/2), and IBM Power6.

For this reason, Chapter 6 also evaluates skewed redundancy in the context of a CMP design composed of in-order cores. Conceptually, such a design is appealing -- many small power-efficient cores can be placed on a single chip to exploit thread-level parallelism; when high single-thread performance is important skewed redundancy can provide a form of out-of-order execution to exploit memory and instruction-level parallelism. The in-order experiments in Chapter 6.6 utilize cores with the same parameters as those presented in Table 3-1 with the additional constraint that instruction issue occurs in program order.

The final row in the table describes simulation parameters specific to skewed redundancy and describe the capacity and latency of the hardware structures proposed in this work. Chapter 6.2 presents sensitivity studies examining the effects of these parameters on overall system performance.

3.3 Benchmarks

The SPEC CPU2000 Integer and Floating-Point benchmark suites provide a collection of standardized programs and inputs intended to represent real-world applications and programming constructs [92]. We use this benchmark set as our primary metric to evaluate skewed redundancy. Table 3-2 lists the SPEC CPU2000 benchmarks and their descriptions[1]. It also indicates their baseline performance in *instructions committed per cycle* (IPC) for both machines models we consider: a moderately-aggressive out-of-order core and a simple in-order core. All performance data in this thesis is normalized to these baseline IPCs. This helps clarify the performance graphs and provides an easy way to display speedups.

Table 3-2: Benchmark Descriptions and Baseline IPC.

Benchmark	Language	OO IPC	IO IPC	Comments
SPECINT2000				
gzip	C	1.26	0.639	Compression
vpr	C	0.680	0.347	FPGA circuit placement and routing
gcc	C	1.27	0.561	C programming language compiler
mcf	C	0.0522	0.0469	Combinatorial optimization
crafty	C	1.87	0.670	Game playing: chess
parser	C	0.956	0.452	Word processing
eon	C++	1.61	0.578	Computer visualization
perlbmk	C	1.38	0.479	PERL programming language
gap	C	1.03	0.410	Group theory: interpreter
vortex	C	1.35	0.461	Object-oriented database
bzip2	C	0.780	0.416	Compression
SPECFP2000				
wupwise	F77	0.971	0.465	Physics / Quantum chromodynamics
swim	F77	0.115	0.0826	Shallow water modeling
mgrid	F77	0.663	0.360	Multi-grid solver: 3D potential field
applu	F77	0.359	0.260	Parabolic / Elliptic partial differential equations
art	C	0.204	0.0686	Image recognition / neural networks
equake	C	0.246	0.167	Seismic wave propagation simulation

1. Three of the SPEC CPU2000 are missing from this set due to difficulties we encountered while running them inside of our simulator: *twolf*, *galgel*, and *facerec*.

Table 3-2: Benchmark Descriptions and Baseline IPC.

ammp	C	1.81	0.559	Computational chemistry
lucas	F90	0.231	0.196	Number theory / primality testing
fma3d	F90	1.97	0.674	Finite-element crash simulation
sixtrack	F77	2.21	0.557	High energy nuclear physics accelerator design
apsi	F77	1.28	0.494	Meteorology: pollutant distribution

Benchmarks written in the C programming language were compiled for the PowerPC ISA with peak optimization by the IBM xlc optimizing C compiler. *eon* is the only benchmark with C++ source and uses g++ version 2.95.2. The subset of SPEC floating-point applications written in Fortran were compiled using the IBM xlf optimizing Fortran compiler. All benchmarks use the SPEC reference input sets and are fast-forwarded 10 billion instructions before timing analysis is performed on the following 200 million instructions. We utilize the cache checkpointing support that has been added to PHARMsim to eliminate cold cache misses when timing simulation is initiated.

3.4 Speedup From the Baseline Hardware Prefetcher

Skewed redundancy improves single-thread performance by enabling very accurate prefetching without the need for large physical instruction windows. Many other prefetching mechanisms have been proposed that can provide similar benefit. One such technique uses a simple hardware state machine that observes cache traffic and attempts to extrapolate access patterns of a program's memory requests. After identifying a pattern, it issues prefetches to preemptively start bringing anticipated lines into the caches.

Our baseline machine configuration utilizes a hardware prefetcher modeled after the Power4 design [93]. The prefetcher detects streaming accesses that sequentially touch adjacent cache blocks in either ascending or descending direction. Active streams are allocated into an eight-entry stream buffer which prefetches the next five cache blocks into the

L2 cache, and the next adjacent block into the L1 cache. Prefetch streams terminate on 4KB page boundaries.

Figure 3-1 graphs the speedup from enabling the hardware prefetcher in the out-of-order and in-order baseline models over systems with no prefetching. Each bar is separately normalized to the baseline performance of that machine model (e.g. “io” indicates prefetching speedup over the baseline in-order model and “oo” indicates speedup over the baseline out-of-order model). This does not provide a direct comparison of in-order verses out-of-order performance, but rather shows the performance contribution of the prefetcher relative to its respective baseline microarchitecture.

We present this data to show that the prefetching benefit of skewed redundancy is not just “low-hanging fruit” that is easily captured by conventional prefetchers implemented in current systems. We observe that the prefetching speedup ranges from 0% (for benchmarks without strided access patterns) to over 2X (for benchmarks with strided access patterns). Because simple unit-stride access patterns are easily captured by skewed redundancy, our reported speedups would be significantly higher on a baseline configuration with no hardware prefetching support.

Finally, we note that in certain cases prefetching helps the in-order model more, while in others it helps the out-of-order model more. Prefetching can provide better speedups in the in-order model because the baseline microarchitecture is ill-suited to overlap independent cache misses. Out-of-order execution, on the other hand, already obtains some of the benefit of prefetching by speculatively issuing subsequent independent misses. Prefetching can provide better speedups in the out-of-order core in applications with significant ILP. With frequent cache miss stalls, Amdahl’s Law prevents dynamic

instruction issue from providing significant speedup. When misses are eliminated by prefetching, the core's instruction throughput can better match the peak execution bandwidth of the machine. When cache misses are eliminated from the in-order core, however, the in-order issue mechanism can become the limiting factor in achieving high performance.

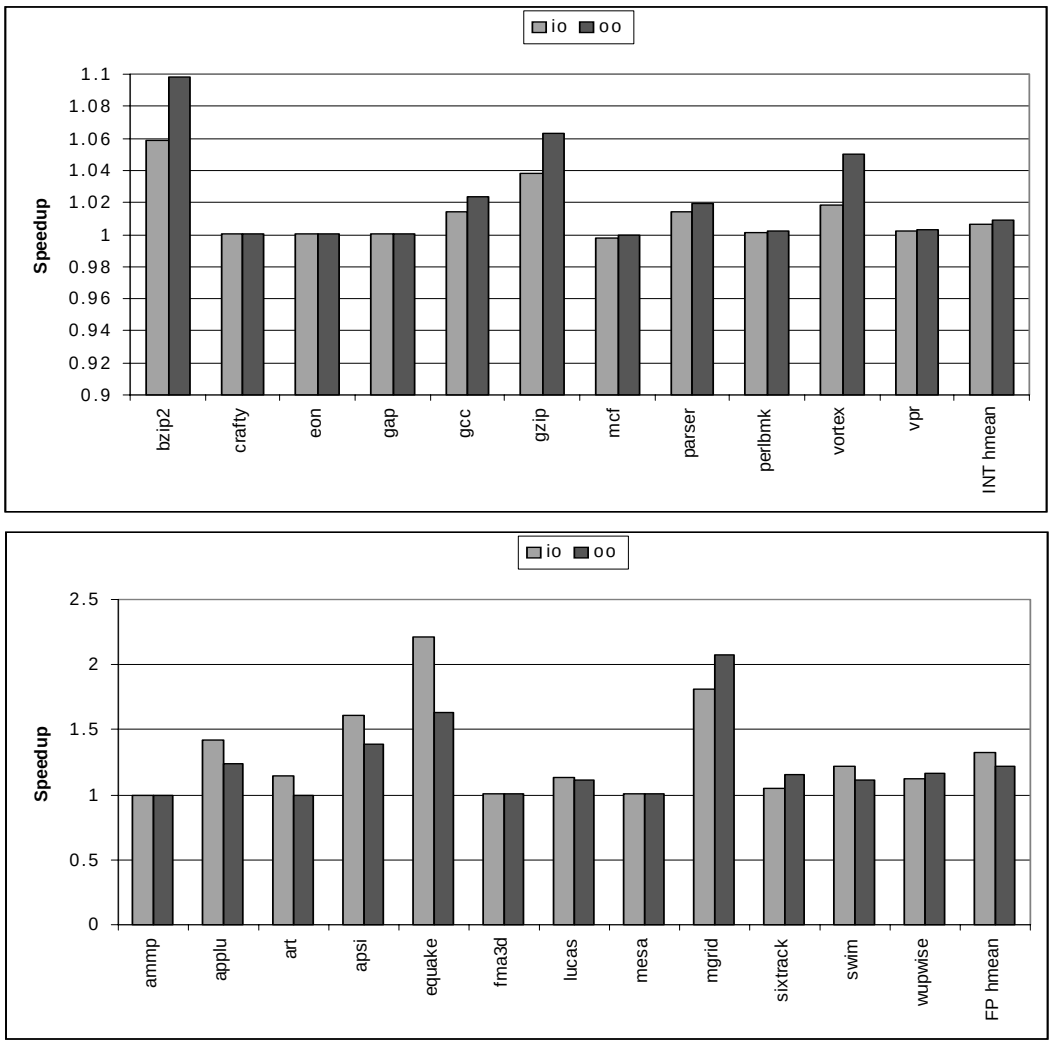

FIGURE 3-1. Speedup from Power4-style hardware prefetcher. "io" indicates speedup over the baseline in-order model. "oo" indicates speedup over the baseline out-of-order model. Note different y-axis scales.

Chapter 4

Increasing Performance

This chapter proposes the fundamental mechanisms employed by skewed redundancy and explains how they can be used to boost single-thread performance. Although detailed performance results are presented in a subsequent dedicated chapter, we include a spectrum of characterization data here that motivates our design decisions. The following chapter demonstrates how these performance techniques can be readily adapted to detect and recover from soft-errors as well.

4.1 Data Prefetching

Chapter 2.1.1 motivated the importance of large instruction windows in achieving high single-thread performance by describing how they enable many instructions to begin execution while waiting for cache misses to complete. Starting execution as early as possible is particularly important for long-latency instructions, such as future cache misses. Chapter 2.1.1 also provided some reasons why large instruction windows are difficult to implement in modern microprocessors. An alternative method to overlap the latency of younger misses is to *prefetch* them into the local cache(s). In this case, by the time a load enters the instruction window and issues to the memory system, its data is already enroute to the cache and the load instruction completes in less than the full memory access penalty.

The ability to overlap multiple outstanding cache misses is often referred to as *memory level parallelism* (MLP) [25], and is the principal goal of all prefetching mechanisms. An accurate prefetcher can exploit most or all the MLP of a large instruction win-

dow, but at a higher efficiency. Without explicit prefetching, a miss begins after it enters the scheduling window and issues to the memory system. It consumes precious instruction window resources (e.g. result register, load queue entry, reorder buffer entry, etc.) until the miss completes, which can take hundreds of clock cycles given the high-frequencies common today. A successful prefetch, on the other hand, predicts the miss address long before the processor fetches the instruction that requires it with the hope that the data will already be cache-resident by the time it is requested. Unlike demand misses, tracking the outstanding prefetch does not require allocating resources within the processor core -- it only requires a small amount of state to indicate the final destination of the miss (often called a *miss status holding registers,* or MSHR [45]). This enables a large number of concurrent misses to be overlapped without large physical instruction windows.

A variety of techniques have been developed to accurately predict which memory addresses will be accessed in the near future. Many architectures include explicit instructions that compilers can use to pass hints of which addresses are likely to be accessed soon. Alternatively, hardware itself can directly issue prefetches based on the history of misses it observes. For example, stride-prefetchers look for sequences of code that step through memory in fixed increments. Once they identify a particular pattern, they can predict when a given cache line is likely to be needed based on the stride length and relative access latencies of different levels of the cache hierarchy [93]. More sophisticated predictors have generalized this approach to identify arbitrary access patterns [38][61]). Finally, other work has focused on constructing specialized software threads than can be concurrently executed on multithreading hardware [15][19][49][102].

An alternative to predictive or analytic prefetching methods is to simply specula-

tively execute the miss-independent instructions while waiting for the miss to complete. This is the approach of *runahead execution* [22][60], which makes the observation that an effective predictor of future memory operations is those operations themselves. When an last-level cache miss is detected in runahead execution, the processor checkpoints its register state and immediately completes the load and marks its destination register with a special INV bit to indicate that it does not contain correct data. Any younger instruction that reads a register marked INV inherits that bit and also immediately completes. Previous work has shown that the forward slice of miss-dependent instructions is typically small [47][88][40], and therefore a potentially large number of independent instructions can be executed under the miss shadow. When the original miss completes, the pipeline is squashed, architected register state is restored from the checkpoint, and execution is restarted at the miss that caused the transition into runahead mode. Any independent misses on the correct control path that were uncovered during the runahead period are at least partially prefetched.

The combination of relatively high memory access times, wide superscalar designs, and few miss-dependent instructions gives runahead execution a long reach into the future instruction stream. Its primary shortcoming, however, is that this reach is bounded by memory latency -- when the miss returns the runahead thread is killed and its state is discarded.

This work leverages the additional hardware contexts available in current and future microprocessors to maintain this state after the original miss completes. We target additional cores in chip multiprocessors, however the ideas presented in this thesis are also applicable to additional threads in simultaneous multithreading processors. We initially

describe a design that utilizes two cores, and later show how a third core can be added to increase performance and provide error recovery. In some sense, skewed redundancy can be thought of as runahead execution implemented on an additional processor core, rather than time-multiplexed on the same core. However there are several important differences:

- **No explicit runahead periods:** each core essentially operates in runahead period continuously. There is no checkpointing on cache misses or restoring from checkpoints on miss completion. If one core has advanced ahead of another by discarding cache misses, it can maintain that lead after the miss returns.
- **Communication between cores:** After the main thread copies its state to the runahead thread in runahead execution, no further communication occurs. Skewed redundancy allows instruction results to be passed between cores. This can be important when redundant threads become significantly skewed with respect to each other, and passing relatively few values between them can prevent expensive squashes that cause leading cores to restart execution hundreds or thousands of instructions earlier.
- **Non-speculative instruction commit:** Each core commits instructions non-speculatively despite the fact that it may have incomplete architected state[1]. This reduces the number of squashes and enables high-coverage soft-error detection and recovery schemes.

This chapter presents our proposal for *skewed redundancy.* It begins with an outline of the basic mechanism by which it improves performance. The remainder of the chapter describes the cases where communication between cores is required, and details

1. In the common case cores commit instructions non-speculatively. In rare cases, a core will be squashed and restarted. Details regarding such events are presented later in this chapter.

the relatively minor microarchitecture changes required to conventional designs. The following chapter extends this framework and highlights how skewed redundancy can be additionally used to detect and recover from soft errors.

4.2 Basic Principles

Figure 4-1 depicts an example system configuration typical of modern chip multiprocessor systems. It consists of several processor cores that each contain private write-through L1 instruction and data caches. An on-chip interconnect runs between the L1s and a unified (instruction plus data) L2 cache shared by all cores. The L2 is backed by a shared L3 cache, followed by off-chip system memory. Skewed redundancy combines two or more cores together to form a single virtual processor that achieves higher single-thread performance than either core on its own, in addition to soft-error detection capabilities. We use this design as a starting point for our proposal.

4.2.1 Load Misses

Skewed redundant execution begins when a program thread is initialized and is transparently mapped by hardware onto separate processing cores. Both cores fetch from the same starting instruction address and begin identically executing the program. At some point, a last-level cache miss will become the oldest instruction in one core's window. A conventional machine would continue fetching and executing instructions until it exhausts its buffering resources and stalls. Under skewed redundancy, the core does not wait for the data to return from memory, and instead immediately marks the load as completed. To indicate that the load destination register contains a bogus value, it marks it INV ('invalid') with a special bit added for this purpose. Dependent instructions that read an

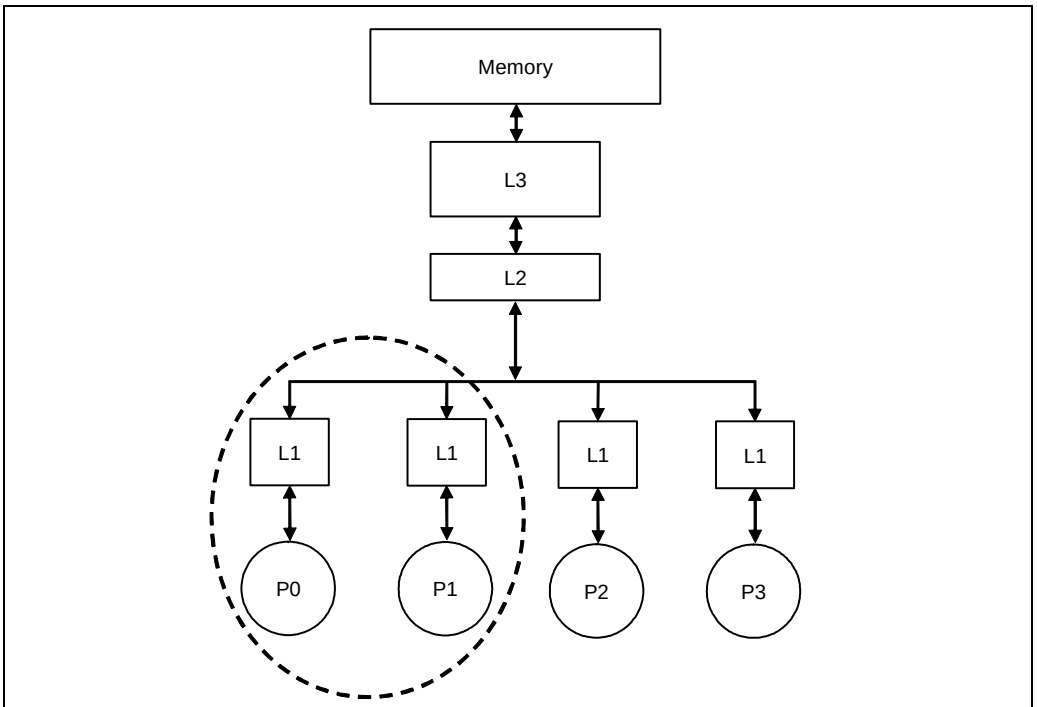

FIGURE 4-1. Example CMP System Configuration. Cores have private L1 caches and share a unified L2 and L3 cache. The dashed line indicates that core P0 and P1 are combined into a single skewed redundant core that has superior single-thread performance and error detection than either core alone

INV register inherit that state and propagate it to their output register and likewise immediately complete. Store queue entries are similarly augmented with an INV bit to allow dependences to propagate through memory to younger loads. We discuss the details of communicating INV state through other levels of the cache hierarchy in Chapter 4.3.7.

Because this starting design includes three levels of cache, only L3 cache misses access main memory and therefore only they are skipped. If L3 hits (i.e. L2 misses) are sufficiently expensive, it can also be advantageous to allow cores to skip them as well. Whether or not this will lead to higher performance depends on a variety of factors, such as the relative memory latencies of each cache level, inter-core communication latency, and the ability of the underlying microarchitecture to hide long-latency events through extracting instruction and memory-level parallelism from applications. Chapter 6 explores these issues and provides performance results; in explaining the mechanisms of our design

in throughout this chapter, we assume that only L3 misses are discarded.

When a miss is skipped by a processor core, it will not cause a stall, and that core can advance and uncover future independent cache misses. In most cases, the core will never require the skipped value, and all instructions that depend on it will also be skipped and executed by the stalled core. However certain cases exist that require the skipped value in order for execution to continue. These cases require inter-core communication, and will be discussed in this chapter.

Note that we do not statically assign one core to always skip last-level cache misses and another core to always wait for those misses. We instead adopt a general policy that simply allows the first core to reach a miss to discard it, and requires the trailing core to wait for it. This allows the core that is currently farther ahead in the global instruction stream at the time the miss is discovered to continue to lead and reach younger misses earlier. We also point out that once a core discards a miss, it is not guaranteed to always lead. Variable memory latency, resource contention, slightly different branch predictor behavior across cores, and inter-core register communication can result in different performance of each core. This can cause cores to advance ahead of each other at different times, and provides an advantage to dynamically partition misses. The hardware mechanisms to implement this partitioning policy will be discussed later in this chapter.

4.2.2 Store Misses

Prior work has shown that, in addition to load cache misses, store misses can also cause machine stalls and limit performance [16][98][48], particularly in systems that implement sequential consistency. Unlike weaker consistency models that support write buffering, stores in a sequentially consistent system cannot retire from a machine until

they are ordered with respect to all other processors in the system. Long-latency store cache misses, therefore, will prevent younger instructions from committing until they complete.

Our proposal for skewed redundancy permits processor cores to discard store misses in the same manner that they discard load cache misses. Only one core needs to wait until the cache block data is retrieved from memory, and the remaining cores can mark the store as completed and continue executing younger instructions. Chapter 4.3.7 provides details describing how younger loads to the same address identify that a prior store was skipped, and Chapter 4.5.4 describes how sequential consistency can still be enforced despite this relaxation of store ordering.

4.3 Inter-core Synchronization

Conceptually, the skewed redundancy model exploits a hierarchy to realize a large virtual instruction window. At the lower level of the hierarchy, conventional and well-understood techniques are used to enable out-of-order execution, maintain precise semantics, and guarantee correct memory ordering within a single processor. At the higher level, a collection of techniques must be employed to ensure the same correctness constraints across each of the processing elements. None of these techniques fundamentally differ from existing techniques for supporting instruction-level parallelism, but they merit discussion, particularly since this paradigm trades local computation for communication and synchronization, hence enabling reasonably-sized and relatively simple structures and mechanisms for inter-element communication.

Dynamically scheduled processors enable multiple instructions to concurrently

execute by using a reorder buffer (ROB) whose purpose is two-fold: it buffers speculatively executed instructions until they can update architected state in-order, and it provides a mechanism to pass speculative results between these instructions. Skewed redundancy exploits parallelism by executing instructions across multiple processor cores. It can be viewed as implementing a *virtual instruction window* consisting of all instructions spanning the oldest and youngest instructions across all cores. This section proposes a mechanism to provide equivalent buffering and communication functionality across the separate cores that constitute the virtual instruction window.

A monolithic instruction window would need to buffer all unique in-flight instructions in the virtual window, which Chapter 6.3 will show can span thousands of instructions in skewed redundancy. However because most buffering and forwarding can be satisfied through the smaller individual ROBs, we only need a mechanism to handle the cases where this functionality cannot be locally provided. Therefore the only instructions that need to be buffered beyond those already contained in individual instruction windows are the forward instruction slices that depend on cache misses to main memory, which previous work has shown to be quite small [47][88]. We propose containing slice instructions in a shared *Global Reorder Buffer* (GROB) to facilitate inter-core communication and precise exception recovery. While the GROB only needs to contain enough entries to accommodate the forward slice of the miss, we will show that the actual number of entries that participate in synchronization between processing elements is far fewer.

The remainder of this section describes the rare cases where cores must communicate, and describes an implementation to accomplish such synchronization. Specifically it presents four requirements under which synchronization is necessary: register dataflow

joins between threads that are executing privately on separate processing elements; maintaining precise state for branch recovery and exception handling; and, finally, maintaining a coherent and consistent view of memory. It also describes how a Global Reorder Buffer can detect and implement these synchronization requirements

4.3.1 Global Structures

The Global Reorder Buffer (GROB) functions similar to a conventional ROB in that it buffers and forwards speculative results before they update program state. It is organized as a circular FIFO queue where each entry corresponds to a unique slice instruction that is private to a single core and contains that instruction's output register value as well as a monotonically increasing instruction sequence number used for relative age comparisons. When a processor core commits a slice instruction that it exclusively executed it copies the result into the GROB at the index specified by its *GROB tail*. When a processor commits a slice instruction that another core executed, it does not access the GROB, but updates its architected register file (ARF) to indicate that the output register value can be found at its current GROB tail in the event that a later instruction requires it. In either case it finally increments its GROB tail modulo the GROB size to point to the next entry. The fact that all processors observe the same instruction stream and identify the same slice instructions enables them to update their GROB pointers locally and consistently, albeit at different times. This enables each core to locate the storage location of all registers it has marked invalid.

The GROB tracks the index of the last entry accessed by each core. The oldest among these indexes represents the *global tail*; all instructions corresponding to entries older than it are guaranteed to have been committed in all cores. After all local GROB tails

have advanced past a GROB entry it is reclaimed and its output value is copied into the *Global Architectural Register File* (GARF). Conceptually, the GARF contains one entry for each architected register whose last definition prior to the oldest instruction across both cores depends on a miss. It provides a mechanism to retrieve the results of slice instructions after they have been de-allocated from the GROB. Each GROB entry also contains an instruction sequence number to indicate if the source register value resides in the GROB or the GARF. A sequence number newer than that of the requesting instruction indicates that GROB entry has been recycled and re-allocated to a younger instruction and that the correct value has been retired to the GARF. Their operation is illustrated by example in the following section.

The capacity of the GROB potentially limits the number of miss slice instructions in the virtual window. Most instructions do not depend on misses, and only on a small degree of buffering is generally required to allow the cores to continue executing instructions and prefetching independent misses. However, if sized too small it can fill, at which point no cores will be able to commit additional slice instructions. Chapter 6.2.1 examines the relationship between GROB size and overall system performance, and shows that a capacity of as little as 32 entries is sufficient to prevent the vast majority of stalls.

On the other hand, the number of entries that the GARF needs to contain is bounded by the number of architected registers (typically less than 100 in most modern RISC architectures). A straightforward implementation might therefore size it equal to the architected register file (ARF) in each core. However it can likely be made considerably smaller for two reasons. First, as we will discuss shortly, system registers are never marked invalid; therefore the GARF only needs to contain entries corresponding to gen-

eral purpose registers (GPRs). Second, the relative infrequency of slice instructions means that few registers are likely to be marked invalid at any time. We can therefore map all architected registers onto a *smaller* pool of physical registers, similar to the way in which conventional microarchitectures map architected registers onto a *larger* pool of physical registers for dynamic scheduling. In this case, if the GARF is fully utilized the trailing core must not commit any slice instructions with destination registers that have not already allocated a GARF entry. Otherwise the GROB would not be able to insert a new value into the GARF when it de-allocates its oldest entry. We re-visit this issue and provide corresponding data in Chapter 4.4.4.

There is a clear analogy between the hierarchical GROB and GARF presented in this work and a conventional ROB and ARF that appear in existing microarchitectures. We point out here that some dynamically scheduled processors substitute a single physical register file with a rename table for a ROB that buffers instruction results (e.g. MIPS R10000 [99] and Intel Pentium 4 [33]). Skewed redundancy could alternatively implement buffering of global in-flight miss slice results through a similar unified physical register file and rename table; we describe it in terms of the GROB/GARF because it is conceptually simpler.

4.3.2 Miss Partitioning

Allowing the first core that reaches a miss to discard it helps sustain the size of the virtual instruction window and increases prefetching effectiveness. Here we describe how a particular core determines if it can safely skip a miss, or if it needs to wait for the data to return from memory.

Because the GROB tracks each core's location within the global instruction

stream, we rely on it to make this decision. When an L3 miss reaches a local ROB head, it presents its local GROB tail index to the GROB. The GROB compares this index to the known tails of the remaining cores. If at least one tail points to an older entry, then the core can discard the miss and update its tail. On the other hand, if all remaining tails point to younger entries, then this core is the last to reach the miss and must wait for it to complete. This requires paying a round-trip latency to the GROB and back to determine if a miss can be discarded. However this additional delay does not significantly impact performance -- either a core will wait for the miss to complete, in which case the additional delay can be overlapped with the miss penalty, or it will not wait for this miss and will discard it, in which case it has traded a last-level cache miss (300 cycles in our model) for a relatively cheap GROB read (10 cycles in our model).

Partitioning misses among two cores in this manner is straightforward -- the first core to reach the miss skips it and the second core waits for it. However, methods to efficiently partition misses among more than two cores are less clear. Chapter 4.9 explores these issues and proposes a simple and effective heuristic for skewed redundant systems with three cores.

4.3.3 Register Dataflow

Now that we have a mechanism to allocate GROB entries to miss slice instructions, this and the following sections present conditions that require GROB entries to be read.

4.3.3.1 Identifying Communication Conditions

Localizing computation dependent upon cache misses removes the primary culprit of full-window stalls from other instruction windows. The core responsible for handling

the miss waits for data to return from memory and exclusively executes dependent instructions. However, some instructions' source operands can reside in different slices executing on different cores (for example an add that sums two values in memory, each of which missed in the cache and was serviced by a different core). In this case no core has both operands and therefore some form of communication is required.

Figure 4-2 depicts a simple example. Loads to the addresses in registers r3 and r9 miss the L3 cache and are handled by processor cores P0 and P1, respectively. Neither core can execute the final write to r4 because each is missing one source operand. Either P1 needs to communicate r5 to P0, or P0 needs to communicate r4 to P1.

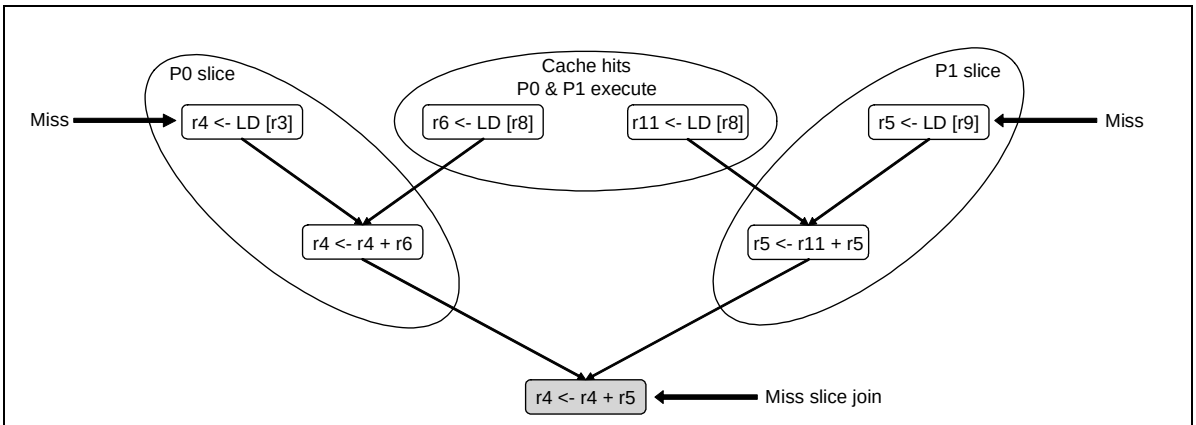

FIGURE 4-2. Miss Slice Join. Addresses in r3 and r9 miss the L3 cache and are handled by cores P0 and P1, respectively. Neither core has both operands for the final write to r4; therefore inter-core communication is required.

A variety of possibilities exist for determining which core will wait for the missing operand and which will discard the instruction and poison its output register. One option might assign the core that is last to produce its local operand responsibility for executing the slice join instruction; this could minimize the number of stall cycles suffered by either core. Another option might make the core furthest back in the global instruction stream responsible for executing the instruction; this could maximize the probability that the leader is able to extend the virtual instruction window and uncover future cache misses

without stalling. However both of these policies require the cores to collaborate and make a *joint decision*. We instead adopt a simple policy that designates the producer of the left operand responsible for retrieving the right operand from the GROB and executing the join instruction. This enables each core to make a *local decision* and avoids the need for centralized decision logic. This chapter later shows that such communication is rare, and optimizing it is not particularly important to performance.

Writes to some registers create an implicit dependency to all younger instructions; therefore we cannot use the data dependence graph to identify all cases where communication is required. Many system registers fall into this category, such as those that control address translation modes, exception modes, and other global machine state. When a core updates a system register with private data (e.g. PowerPC's *Move to Machine State Register* instruction), it is necessary to propagate the new system register value to the other cores. We handle this case by forcing cores to retrieve invalid source operands of system register update instructions. This ensures that system registers always have correct and up-to-date values in all cores. Chapter 4.4.4 shows that updates to machine state with invalid data are rare.

4.3.3.2 Communication Mechanism

The previous section discussed the first of four situations that require inter-core register communication. Now we turn our attention to identifying this first situation, as well as the communication mechanism used to handle it.

In addition to an INV bit, each register is associated with a P (“Private”) bit to indicate dependence on a miss executed by that core. Because private register values are normally marked invalid in another core, instructions that source private registers must make

their results available by writing them to the GROB. The INV and P bit are mutually exclusive; a register can have either set or none (denoted “Global” state). Instructions with two source operands can therefore have the nine states listed in Table 4-1. For a given combination of source states, each row indicates the resulting destination register’s final state, whether the instruction is locally executed, and whether a GROB read or write is necessary. If a processor core has both operands locally available (G or P state), it executes the instruction. If either operand is marked INV, the core discards the instruction and continues executing, with the exception of the case in which the left operand is marked P and the right operand is marked INV (depicted in Figure 4-2 and row seven in Table 4-1). In this case no core has both inputs and the one with the left operand needs to retrieve the right operand value as follows: when the instruction reaches the local ROB head it looks up the right source operand in its ARF. Because this value depends on a miss it will be marked invalid and will not have a corresponding value. In its place will be the GROB index where the other core will eventually place the missing value. The core issues a GROB read that will block until the data is written and made available by the other core. At that point it has both source operands and can execute the instruction.

Table 4-1: Miss Slice Joins -- Register States.

Left Op	Right Op	Dest	Execute?	Read GROB?	Write GROB?
INV	INV	INV	N	N	N
INV	G	INV	N	N	N
INV	P	INV	N	N	N
G	INV	INV	N	N	N
G	G	G	Y	N	N
G	P	P	Y	N	Y
P	INV	P	Y	Y	Y
P	G	P	Y	N	Y
P	P	P	Y	N	Y

It is also possible that the other core has already written the missing value to the GROB, both cores have committed the producing instruction, and the GROB entry has

been freed and re-allocated to a younger instruction. This can be identified by the fact that the instruction sequence number recorded in the source GROB entry is numerically larger than the instruction sequence number of the local instruction issuing the read. At this point, the core requiring the operand reads the GARF entry corresponding to that logical register.

Note that we defer GROB reads until an instruction reaches the ROB head and becomes the oldest in its local window. This simplifies our communication mechanism because each core's GROB tail pointer provides the only method to access the GROB and is only updated at instruction commit. Renaming the local GROB tails to allow speculative read accesses adds considerable complexity because all older slice instruction may not yet have been identified. Because GROB reads are rare and depend on long-latency cache misses, we do not explore this option.

Table 4-1 emphasizes skewed redundancy's ability to trade communication for computation. Instructions with at least one private source are conventionally executed and their results are copied to the GROB. The copy operation is largely off of the critical execution path because the result will simultaneously reside in the local register file and can be used by subsequent dependent instructions in that core. In most cases dependent instructions will be discarded by the other core, and that GROB entry will never be read.

Figure 4-3 provides an example. It shows two processors as well as their interaction with the additional hardware structures described above. Processor core P1 first issues the initial load to address [r8], which misses the last-level cache. P1 does not wait for the miss to complete and commits the load and marks its destination register (r1) INV. Because P0 is the last core to reach that miss, it must wait for the data to return from mem-

ory. It marks r1's P bit, which is then inherited by the subsequent dependent instruction and propagated to r2. When P0 eventually commits the load it copies the result from r1 into GROB[0]. It then increments its GROB tail from 0 to 1. When it commits the next instruction it likewise copies r2 into GROB[1] and increments its GROB tail to 2. The third instruction does not depend on a miss and commits normally without accessing the GROB or incremented its local GROB tail. The fourth instruction misses the L3 and is assigned to P1. P0 discards the miss, marks r4 INV, updates the entry for r4 in its ARF to indicate that GROB[2] will have the missing value, and increments its GROB tail to 3. The fifth instruction inherits r4's INV bit and similarly marks r5 as occupying GROB[3] before incrementing the GROB tail to 4. When the sixth instruction reaches the ROB head P0 finds its right operand (r5) in INV state and its left operand (r1) in P state. This corresponds to the seventh row in Table 4-1, and requires that P0 retrieve r5 and exclusively execute the instruction. P0 looks up r5 in its architected register file (ARF), which indicates that r5 will reside at GROB[3]. It issues a blocking read to that entry and when r5 is supplied by the GROB it is copied into P0's ARF in shared state. The instruction now has both source operands and can execute.

P1 operates similarly. Because P0 is handling the initial load miss to [r8], P1 marks r1 INV. The second instruction propagates the INV bit to r2. Neither of these instructions consume execution bandwidth nor do they access the GROB, however they will increment P1's GROB tail as they commit. The third instruction is miss-independent and executes normally. The fourth instruction is an L3 load miss that P1 will handle; therefore it sets the P bit for r4 in its ARF. The P bit in r4 is then propagated to r5 in the next instruction. The sixth instruction is a miss slice join that P0 is responsible for executing (because it has the

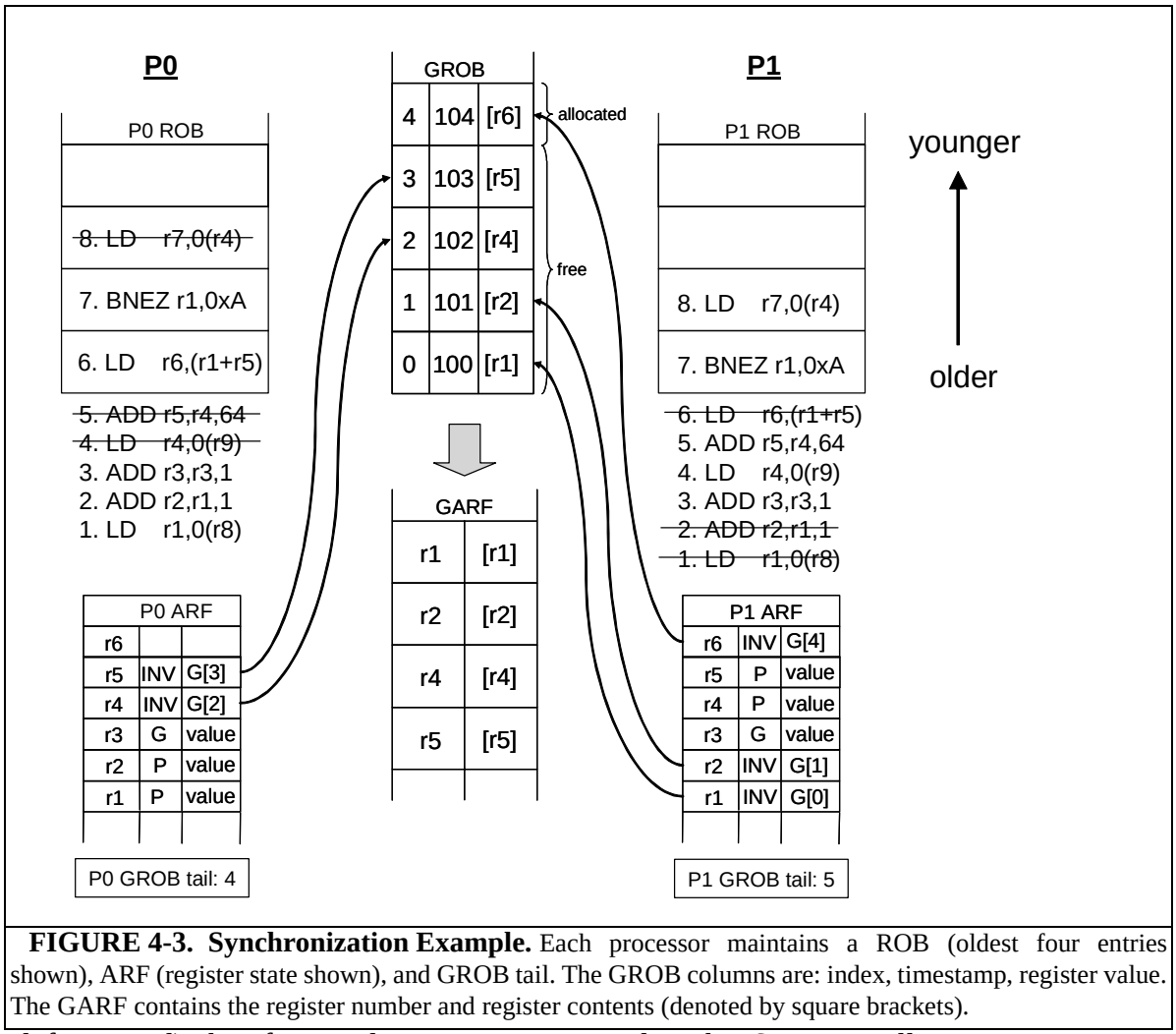

FIGURE 4-3. Synchronization Example. Each processor maintains a ROB (oldest four entries shown), ARF (register state shown), and GROB tail. The GROB columns are: index, timestamp, register value. The GARF contains the register number and register contents (denoted by square brackets).

left operand), therefore P1 does not execute it and marks r6 INV. Finally P1 increments its GROB tail from 4 to 5 to point to the GROB entry that it should write to next.

At this point P0 and P1's GROB tail pointers are 4 and 5, respectively. Because they have both advanced past entries 0-3, these first four GROB entries can be allocated to later slice instructions. When a GROB entry is overwritten its output value is copied to the GARF. For example, P1 may continue committing instructions and allocating GROB entries for their miss-dependent results. Its GROB tail will eventually wrap around the circular queue and entry 3 will be overwritten. If P0 reads GROB[3] and determines that its sequence number is newer than the load join instruction, indicating that the entry has been

recycled, it instead retrieves r5 from the GARF. Instruction sequence numbers are incremented concurrently with GROB tail increments, and can be limited to LOG_2(GROB size) + 1 bits by utilizing a technique such as that proposed by [37] to deal with counter overflows.

4.3.4 Control Flow

While individual processor cores can discard instructions that contribute towards the global dataflow of a program, similarly discarding branches will result in ambiguous control flow. If a branch direction or target depends on a last-level cache miss, then the processor waiting for the miss needs to communicate the branch outcome to the remaining processors in the system. The other processors therefore need to wait until the branch is resolved before they can commit it. When a branch with an invalid source operand reaches a local ROB head, it retrieves the operand value from the GROB/GARF in the same way it would for a miss slice join operand. In the example presented in Figure 4-3, P1 executes a branch with a source operand (r1) marked INV. It can predict the branch outcome and continue fetching and executing instructions within its local window, however it cannot commit the branch until its prediction has been verified. It obtains the GROB index of the last producer of r1 from the ARF, and reads that entry from the GROB. It compares the instruction sequence number of GROB[0] with that of the branch to determine if the GROB output is valid or if it instead needs to read the value from the GARF. Even though the branch falls within the miss slice, it does not write to a general-purpose register and therefore will not allocate a GROB entry when it commits.

It may initially seem that stalling miss-dependent branches will limit parallelism and impact performance. However other work has shown that branches with miss-depen-

dent sources are more likely to be mispredicted, and that those mispredictions are likely to occur shortly after the branch [40]. Our own similar results are presented in Table 4-2. Miss-dependent branches are much more likely to be mispredicted in most of the benchmarks we studied -- up to 6 times more likely in one case (*ammp*). Therefore even in the presence of a mechanism to speculatively commit branches with invalid sources, little additional useful work will likely be exposed due to their relatively high mis-prediction rate. In addition to complicating misprediction recovery, speculatively committing wrong-path instructions can lead to reduced soft error coverage, since the result of an erroneous instruction may match that of a wrong-path instruction committed by another core.

Table 4-2: Overall and Miss-dependent Branch Prediction Rates.

Application	Correct Predictions	Correct Slice Predictions	Difference
bzip2	0.92	0.89	1.4x
crafty	0.94	0.82	3x
eon	0.85	0.76	1.6x
gap	0.94	0.96	0.6x
gcc	0.97	0.96	1.3x
gzip	0.97	0.96	1.3x
mcf	0.97	0.98	0.6x
parser	0.96	0.88	2x
perlbmk	0.93	0.82	2.6x
vortex	0.99	0.97	3x
vpr	0.91	0.91	-
ammp	0.99	0.93	6x
applu	0.97	0.95	1.7x
art	0.97	0.97	-
apsi	0.93	0.90	1.4x
equake	0.95	0.86	2.8x
fma3d	0.96	0.98	2x
lucas	0.99	0.99	-
mesa	0.92	0.99	0.13x
mgrid	0.97	0.93	2.3x
sixtrack	0.94	0.88	2x
swim	0.99	0.99	-
wupwise	0.96	0.97	0.8x

4.3.5 Maintaining Precise State

Committing an instruction updates the architected state of the processor and de-allocates any resources it consumed during execution. The danger in committing instruc-

tions out of program order is that doing so implies that no older definitions of that logical register will be needed, which may not be true in the event of an exception. Although individual cores in skewed redundancy commit instructions in-order with respect to their local instruction windows, commits occur out-of-order with respect to the global instruction stream. When an exception is raised some registers may depend on a miss handled by another core and are marked invalid. Furthermore, that other core may not have the missing registers if it discarded the excepting instruction and, in the course of committing subsequent instructions, overwrote those registers.

Correct exception recovery in skewed redundancy is enabled by the fact that all slice register values are first copied to the GROB, and then to the GARF when they reach the global commit point. Consider again the example in Figure 4-3. If the load from [r4] in the eighth instruction causes a page fault when it reaches P1's ROB head, precise exception semantics mandate that all older instructions but no younger instructions have committed their results. However P1's register state is incomplete because r1, r2, and r6 are marked invalid in its ARF. Because the load address depends on a miss handled by P1, P0 will not observe the exception and will continue committing instructions.

Before P1 branches to the page fault exception handler routine it scans its ARF to identify its missing register values. It looks up invalid registers in its ARF to determine their locations and reads those values from the GROB or GARF. At that point it has a valid set of registers that it copies to the remaining cores and restarts their execution at the excepting load's program counter address. Execution exceptions are generally infrequent, and the additional latency to synchronize state between cores does not significantly impact performance.

As slice instructions commit they copy their results to the GROB for possible consumption by dependent instructions executing on another core. An alternative to copying results of producing instructions is to copy sources of consuming instructions. Consider the following two instructions:

```
r4 <- load        // cache miss; P0 handles
beq r4, 0xA000    // P1 needs r4 to resolve branch
```

Rather than copy the value of r4 to the current GROB tail when instruction 1 commits, P0 could identify that P1 will need r4 when it executes instruction 2, and write r4 to the current GROB tail when instruction 2 commits. This has two potential advantages. First, it eliminates one level of indirection. When instruction 2 reaches P1's ROB head, it can access the GROB directly rather than first looking up r4 in its ARF to identify its GROB location. Second, GROB entries are only written when it is known that they will be read by another core. That is, instead of buffering all slice instruction results, one could only buffer the subset of slice results that are communicated between cores.

Such a technique could work for communicating resulting from register dataflow joins, ambiguous control flow, and unknown store address (discussed in the following section) because each core knows a priori whether another core will eventually need that value. However, applying it to precise exception recovery is problematic. Consider again the exception example in Figure 4-3. If instruction 8 on P1 causes a page fault, then P1 needs to retrieve the values of r1, r2, and r6. It looks up each register name in its ARF to determine the GROB index where each resides. The alternative approach where P0 writes the required registers to the GROB only when it determines that P1 needs them is compli-

cated by: a) it would require storing an arbitrary number of values in a single GROB entry (r1, r2, and r6), and b) P0 may not execute instruction 8 (it does not in this example because the load address depends on a discarded miss) and will not identify that a page fault will occur on another core and that the values of those three registers are required. Therefore, we instead assume that miss slice results are written to the GROB when their producing instruction commits, rather then when they are required by a consuming instruction.

4.3.7 Memory Dataflow

Skewed redundancy exposes additional ILP and MLP by allowing some cores to execute and commit instructions ahead of others. However, memory operations must still appear to complete in program order and adhere to uniprocessor program semantics. This section discusses the requirements sufficient to track and honor memory-based dependencies between instructions. Conceptually, these techniques are similar to conventional methods of tracking memory dependencies employed in out-of-order uniprocessors. Most dependences are handled locally, using conventional load and store queues within a single processor core. To handle dependences across cores, we propose an intermediary FIFO structure called the *Global Store Queue* (GSQ) that resides between the private L1 caches and shared L2. Similar to a conventional store queue, the GSQ's purpose is to enforce WAW and RAW dependences.

4.3.7.1 WAW Dependences

Allowing writes to update the L2 and memory in the order that they locally commit can result in one processor committing a store before an older store to the same address on a different processor, creating a write-after-write (WAW) violation. To deal

with this problem, all stores are inserted into the GSQ when they locally commit and are released to the L2 in program order.

Similar to the GROB indexing mechanism, all cores maintain a GSQ tail index that points to the GSQ entry corresponding to their oldest uncommitted store. When a processor commits a store, it writes its address into the entry pointed to by its GSQ tail and, if locally available, the store data as well. One or both of these fields may be redundantly written by multiple cores; this can either be permitted (since the values written are identical), or a potential write-port optimization could first read the GSQ and only write a field if it does not yet exist. After a processor writes a store into the GSQ it increments its tail modulo the number of entries in the GSQ to point to the next entry. As the trailing tail advances past store entries, they are released to the shared L2 in program order. This design is depicted in Figure 4-4 and is similar to an ordered, non-coalescing store buffer [24].

4.3.7.2 RAW Dependences

In addition to applying stores to memory in program order, we must also ensure that loads receive the value written by the most recent store to that address. A conventional uniprocessor accomplishes this by comparing loads to older in-flight stores in its store queue. If an address match occurs, the load must either wait for the store to commit and update the cache, or the store can forward data directly to the load.

Skewed redundancy must perform an analogous match across the window of in-flight stores in the GSQ. However, we exploit the inherent hierarchy in our design to streamline this process. Processors still perform conventional local lookups in their L1 data cache and local store queue; empirically, the vast majority of independent and depen-

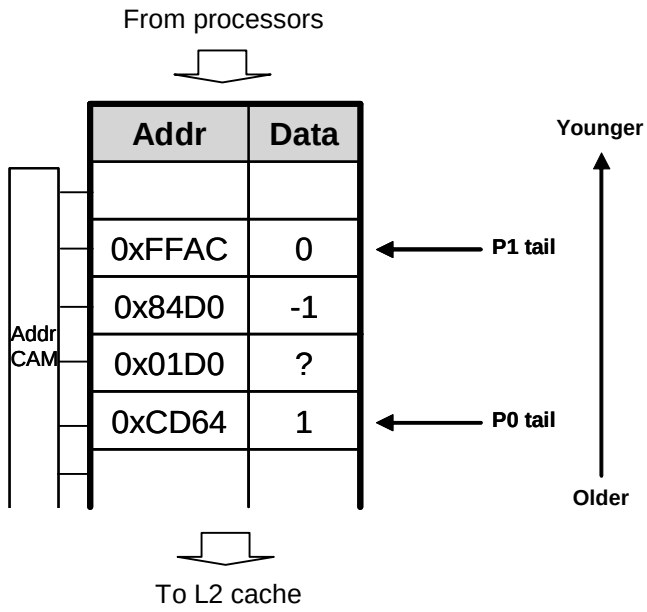

FIGURE 4-4. GSQ Structure. Entries are inserted in FIFO order at local tail pointer positions. Unavailable miss-dependent store data is skipped and written later (denoted by the question mark). Additional control logic determines when all have tails have advanced past a store, causing an entry to be reclaimed and to update the shared L2.

dent loads are handled in this fashion. That is to say, independent loads will usually hit in the L1, requiring no off-core communication. Similarly, dependent loads usually occur within the scope of the local store queue, and are resolved locally with conventional store queue forwarding, or, if the load and store are further separated, are handled by the local cache, since the vast majority of store instructions are executed on all cores, and write their results into each local cache. The only loads that must leave the core are those that miss the L1 and the local store queue. In this case, the load is sent to the L2 and the GSQ, and its address is compared against older stores in the GSQ. An address match indicates that the data in the L2 or main memory is stale and the correct value is (or will be) in the GSQ.

When a processor commits a store it writes its private L1 cache. If the store data register is marked invalid, the store can proceed, but the fact that its target location no

longer contains up-to-date data must be conveyed to younger loads. Otherwise, simply discarding the store can result in a younger load to the same cache line reading stale data from its L1. Therefore when a processor commits a store with invalid data it must invalidate any matching line from its L1. This forces younger loads to miss the L1 and search the GSQ for an address match. If the load does not match an older entry in the GSQ it can safely access the L2 cache. If the load address does match an older store, it indicates that the L2 has a stale copy and that the correct value is (or will) reside in the GSQ. This identifies that the load depends on a prior miss, and the load marks its destination register invalid and is dropped from execution. Figure 4-5 depicts this algorithm.

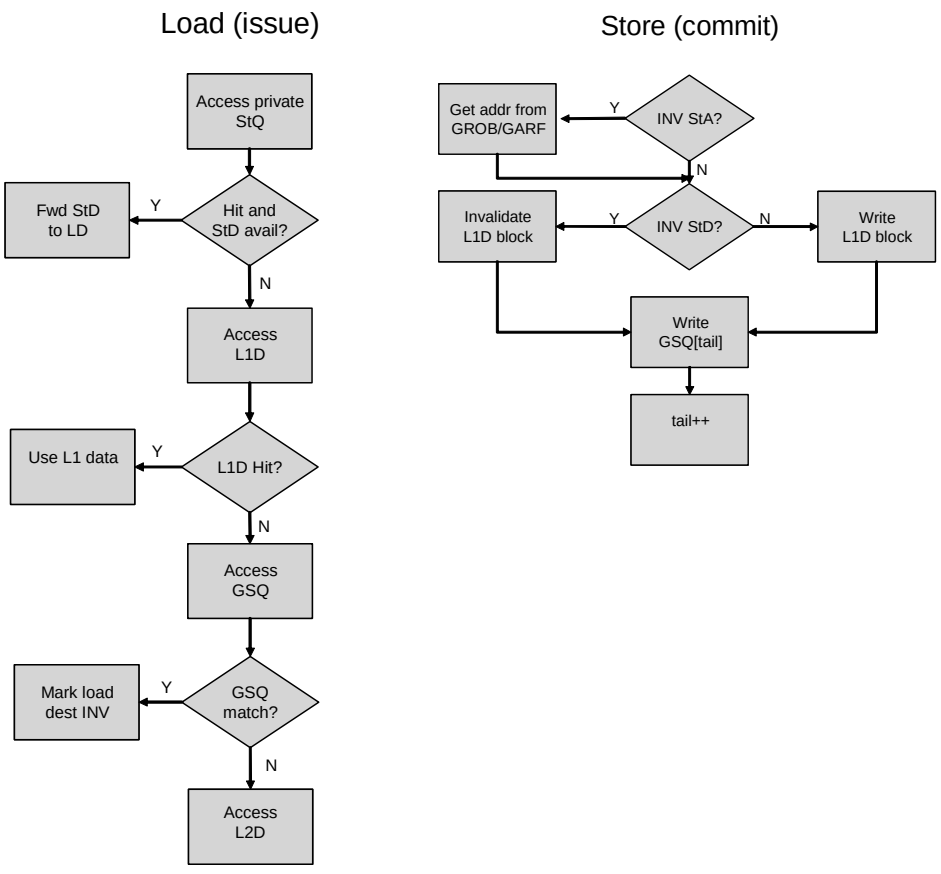

FIGURE 4-5. Handling Memory Operations. Algorithms for load and store execution.

Initially it may appear that better performance can be attained by allowing stores of poisoned data to invalidate the L1 cache at finer granularities than an entire cache line. We have investigated performing L1 invalidates for individual words within a cache line, and have found that it offers no significant performance advantage. This can be explained as follows: When false sharing is eliminated with word-level invalidates, a younger load to a different address within the same line can still hit in the L1 and receive valid data. In the case of coarser-grained line-level invalidates, on the other hand, the load will be conservatively marked invalid and discarded. Therefore the penalty of false sharing in this case is that additional registers may be marked invalid, and communication between cores may increase. In practice we have found that such additional communication is minimal and performance is not significantly affected.

4.3.7.4 Store Addresses

The previous section described how invalid store data is propagated to younger loads of the same address. However, if the store address itself depends on a miss and is marked invalid, that store cannot be correlated to younger loads. This loss of memory dependence visibility can result in loads incorrectly receiving stale data from older stores. We deal with this problem by simply forcing all processors to compute all store addresses. When a store address register is marked invalid it is retrieved from the GROB or GARF as it would for a slice join or invalid branch operand. This ensures that all loads can identify the last store to their address in program order, even if the store data is not available. In general, the fraction of store addresses that depend on load misses is typically small, and forcing all processors to generate all store addresses does not significantly impact perfor-

Memory dependence prediction [57][17] has been shown to be an effective mechanism to predict if a load will receive data from a nearby store. This technique could be leveraged here by allowing stores with unresolved addresses to commit and either stall or poison younger loads predicted to alias to those stores. This would avoid the synchronization overhead of retrieving invalid store addresses; however, it requires detection and recovery mechanisms to handle the rare cases where the prediction is incorrect. Detecting this case can be difficult, since a processor core would need to remember its past predictions of committed load addresses and compare them to store addresses computed on other cores. Recovery can also be problematic, since the state of the core that mis-speculated would need to be rolled back to a point prior to the misprediction. If waiting for store addresses resolution is prohibitive to high performance for important applications, such a heavy-weight mechanism could be employed. A similar mechanism could also be used to allow high-confidence unresolved branches to speculatively commit as well. Because these types of synchronization events are generally rare, we do not explore such alternatives in this work.

4.4 Inter-core Synchronization Results

Skewed redundancy improves performance by allowing cores to skip expensive cache misses and continue executing younger instructions. The previous sections presented several cases in which the leading core cannot progress until it retrieves miss slice values. Since these values are not likely to be available for some time, if such inter-core synchronizations are frequent it will limit the degree that cores can become skewed with

This section presents data that indicates that synchronization is not frequent, and provides motivating evidence that skewed redundancy can be a viable performance-enhancing technique. Because we do not present cycle counts here, this data is not as sensitive to microarchitectural parameters as that presented in Chapter 6; nevertheless it does depend on the size and latencies of the various levels of the memory hierarchy. Details of our machine model and simulation methodology appeared in Chapter 3.2.

4.4.1 Number of Misses and Dependent Instructions

Table 4-3 presents two statistics collected from our application set: the number of L3 misses per 1,000 committed instructions and the average number of dependent instructions per miss (i.e. the miss slice length). Working set size and locality varies considerably across the benchmarks, leading to significant differences in the L3 miss rate. Some benchmarks miss as rarely as twice per 100,000 instructions (*eon*), while others miss more than once per ten instructions (*mcf*). Clearly, applications that suffer a greater number of L3 misses stand to benefit the most from skewed redundancy. We revisit this table during our performance results discussion in Chapter 6.1.

The second column indicates the average miss slice length. We classify an instruction as miss-dependent if it consumes a register written by a load miss (either directly or transitively through intermediary instructions), or if it depends on a load that was forwarded from a store miss via the store queue. It confirms prior work [47][88][40] that shows that relatively few instructions depend on cache misses, and that significant independent work can be started while waiting for a miss to complete. Only three benchmarks have average miss slices greater than 10 instructions, and in several cases misses feed an

average of only one or two instructions. Applications with slice lengths below one instruction per miss have a significant fraction of store misses, which do not produce a register output. Because miss slice instructions are buffered in the GROB, the fact that they are few suggests that the GROB does not need to be large. The performance impact of limited GROB capacity is further explored in Chapter 6.2.1.

Table 4-3: Number of L3 Misses and Dependent Instructions.

Application	L3 misses / 1k instructions	Average Miss Slice Length
bzip2	3.5	2.1
crafty	0.065	7.2
eon	0.026	2.6
gap	1.6	24
gcc	1.2	8.6
gzip	1.0	36
mcf	124	4.7
parser	1.2	4.2
perlbmk	0.024	11
vortex	0.93	4.0
vpr	6.3	2.6
ammp	0.12	16
applu	48	4
art	62	2.5
apsi	2.7	2.9
equake	54	7
fma3d	0.093	3.8
lucas	18	1.5
mesa	0.30	1.4
mgrid	11	4.9
sixtrack	0.049	3.4
swim	57	4.7
wupwise	4.4	33

4.4.2 Private vs. Global Execution

Figure 4-6 breaks down miss slice instructions according to which core executed them in a two-core skewed redundant configuration. The bottom bar represents instructions that do not depend on a miss and can be computed directly from Table 4-3 as:

$$\text{Miss Independent Instructions} = 1 - \left(\frac{\text{L3 misses / 1k instructions}}{10} \times (\text{Avg Miss Slice Length} + 1) \right)$$

We observe that over 90% of instructions do not depend on last-level cache misses and globally execute on both processor cores in 17 of the 23 benchmarks. The remaining instructions are exclusively executed by either P0 or P1 and allocate GROB entries when they commit. Figure 4-6 confirms an earlier point made in Chapter 4.2.1: that once a core discards a cache miss and becomes the leader, it does not necessarily always lead. Although small variations in microarchitectural events (such as different prefetching and branch prediction behavior) can partially explain this behavior, the most common reason for cores swapping positions is synchronization. If a leading core needs to retrieve a miss-dependent value from the GROB, it must wait for another core to catch up and produce that value. Because GROB reads do not occur instantaneously, the core may fall behind by up to the number of cycles required for the read to complete (10 cycles in our model). The larger percentage of instructions handled by P1 can be explained by our arbitrary decision to force P1 to wait for the miss in the event both reach it in the same cycle.

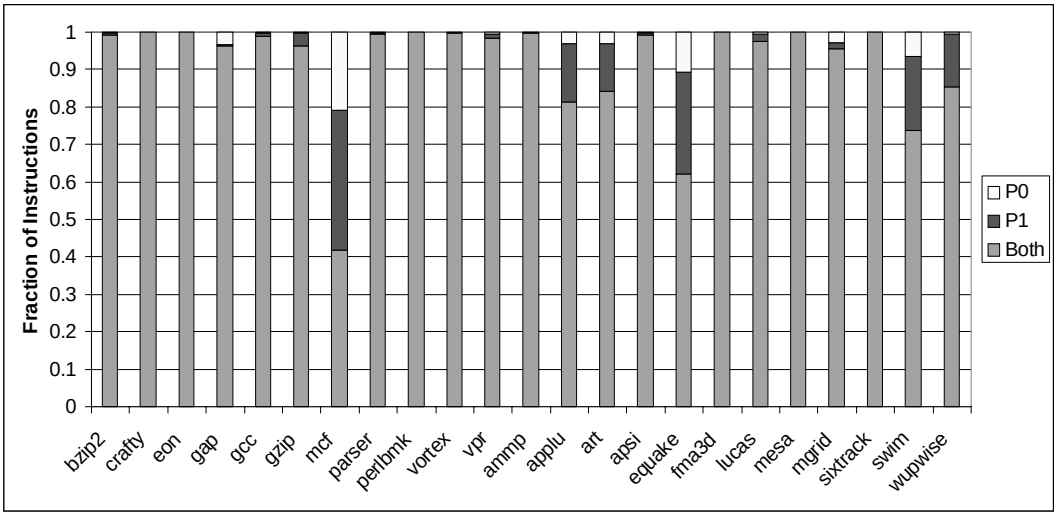

FIGURE 4-6. Fraction of instructions executed by each core. Instructions executed by P0, P1, or both P0 and P1.

4.4.3 Synchronization Frequency and Causes

Copying miss-dependent values to the GROB does not necessarily imply inter-core synchronization. Rather, results are conservatively buffered in case another core requires them according to the conditions presented in Chapter 4.3. In most cases, they will not be read and will be eventually overwritten by results produced by younger instructions.

Figure 4-7 indicates the overall fraction of committed instructions that read a source operand from the GROB (or GARF, if the value has been since exited the GROB), as well as which synchronization condition caused the read. Comparing this to the previous data in Figure 4-6, we see a correlation between the number of GROB reads and writes in some, but not all applications. For example, almost every read results in a corresponding write in *bzip2*, while few of the relatively large number of miss-dependent outputs are ever required outside of the producing core in *swim*. Despite this variance, almost all benchmarks exhibit far fewer GROB reads than writes. In all but one case, fewer than 0.5% of all instructions require miss-dependent sources, suggesting that cores' execution can become significantly skewed between register synchronizations.

Figure 4-7 also provides a breakdown of the conditions that caused the read. In most applications, synchronization occurs when a branch direction or target depends on an L3 miss. Although the branch can be predicted, it cannot commit until its invalid operand(s) are retrieved and its outcome verified. In general, the integer benchmarks exhibit less regular control flow and contain a larger number of miss-dependent branches, while the floating point programs often contain a small number of predictable, miss-independent, static branches that iterate over large data sets. The next most common case occurs when a store address depends on a miss and must be resolved to ensure that each core can

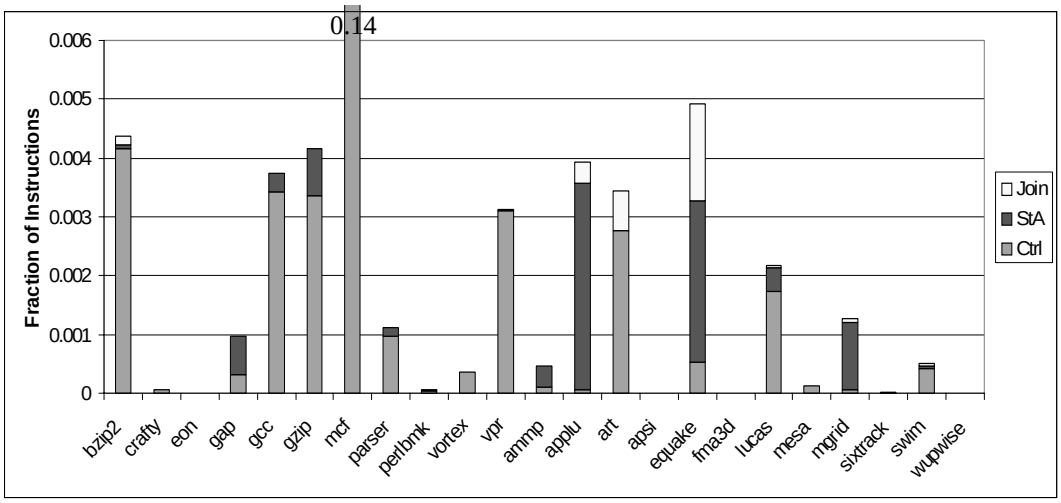

FIGURE 4-7. Synchronization Frequencies and Reasons. Each bar represents the fraction of total instructions required to read the GROB or GARF due to an invalid branch operand, invalid store address, or miss slice join. *mcf* has an exceptionally high fraction of instructions that read invalid source operands (0.14 per instruction); therefore the vertical height of the graph does not consider it

properly disambiguate memory references. Finally dataflow joins, where an instruction simultaneously depends on two misses handled by separate cores, occurs much less frequently, and are only statistically significant in three benchmarks (*bzip2, art, equake*). The remaining conditions (such as write to system registers by instructions with invalid sources, and any combination of multiple reasons) are extremely rare and do not appear at all in Figure 4-7.

In the interest of clarity, the scale of the vertical axis of Figure 4-7 does not consider *mcf*, which requires GROB reads for 14% of its instructions. Examining *mcf*'s critical execution loop provides insight into this behavior:

```
while( arcin )
{
   tail = arcin->tail;

   if( tail->time + arcin->org_cost > latest )
   {
       arcin = (arc_t *)tail->mark;
       continue;
   }

   red_cost = compute_red_cost( arc_cost, tail, head_potential );
```

```
    if( red_cost < 0 )
    {
        if( new_arcs < MAX_NEW_ARCS )
        {
                    insert_new_arc( arcnew, new_arcs, tail, head,
                                    arc_cost, red_cost );
                    new_arcs++;
        }
        else if( (cost_t)arcnew[0].flow > red_cost )
                    replace_weaker_arc( arcnew, tail, head,
                                        arc_cost, red_cost );
    }

    arcin = (arc_t *)tail->mark;
}
```

This code implements pointer chasing where references to almost all nodes in the linked list miss the L3 cache. The body of the function *compute_red_cost* (not shown) is only a single mathematical expression, and returns a positive value in almost every iteration. Consequently, the loop body reduces to only a handful of machine instructions, at least one of which misses L3. Because the loop termination condition tests the result of the miss, virtually every instance of the backward branch requires a GROB read. That is, one core discards the miss caused by dereferencing `tail->mark`, however the subsequent instruction is a dependent branch and results in a blocking GROB read. This large number of serialized misses results in greater than one L3 miss per 10 committed instructions, and an average throughput of only 0.05 instructions per cycle.

4.4.4 Fraction of Registers Marked Invalid

It may seem that once a processor core skips a miss and marks the load destination invalid, the invalid bit will eventually propagate to all remaining architected registers and the core will be unable to perform useful work. Normally, this situation does not occur. Chapter 4.4.1 showed that the dependence chain of L3 misses is usually only a handful of instructions long. It is eventually terminated by a store instruction, and younger loads to

the same address will start a new chain. Furthermore, the fraction of invalid architected registers is also reduced by the fact that registers transition from invalid to global when they are communicated between cores. For example, an invalid branch source register will be communicated through the GROB or GARF, and cores will initialize it to global state after the transfer completes.

We introduce a final optimization to deal with pathological cases where a load miss result remains register-allocated over long periods of time and invalidates significantly more registers. Periodically, we break all dependence chains by triggering a global replay from the oldest in-flight instruction among all cores (as if the oldest instruction raised an exception). Architected state is reconstructed and copied to all cores, and all registers revert to global state.

Figure 4-8 graphs the cumulative distribution of the number of invalid entries in the architected register file (ARF) as a function of cycle count for a single benchmark -- *applu*. *applu* was chosen because of its large fraction of invalid registers and its sensitivity to synchronization frequency; most other applications contain a small number of invalid registers at any time and are therefore not significantly affected by this optimization. It plots three data distributions: the number of invalid registers when synchronization is triggered every 100,000 cycles, every 1 million cycles, and not at all. This data was collected by examining the ARF in a single core of a 2-core skewed redundant system; results for the opposite core are similar.

When synchronization is not applied, the ARF contains no invalid registers for only 3% of the overall execution time. It contains less than two invalid registers for 74% of the time, and less than 26 registers for 90% of the time. The fraction of invalid registers

drops dramatically when local ARFs are synchronized every 1 million cycles. In this case the ARF contains less than 12 invalid registers for 90% of execution time (compared to 26). Although detailed performance results are not presented until Chapter 6, we note here that this simple optimization results in a 22% performance gain in *applu* over the case that does not employ periodic synchronization. The benefit of this optimization is that individual cores stall less often to obtain missing values. The cost is the additional latency due to pipeline squashes and invalid register transfers; however, because they occur so infrequently, we can afford them to be relatively heavy-weight. For example, even if we conservatively assume that synchronizing all invalid registers across cores takes an additional 500 processor cycles, enforcing it every 1 million cycles would result in a performance degradation of only 0.05%. Figure 4-8 also displays the cumulative distribution of invalid registers when we aggressively synchronize all cores every 100,000 cycles. It shows that there is little incremental benefit over the case of 1 million cycles. Because it represents a tenfold increase in overhead, we do not consider it further. All performance data as well as the characterization data presented in Figure 4-7 assume a synchronization period of 1 million cycles.

4.5 Observing Misses Consistently

Our scheme for assigning space in the GROB for instructions in the forward slice of a cache miss is entirely distributed: each core performs this allocation independently, avoiding the need for a centralized allocator. It is also efficient, since only slice instructions are buffered, which is a small subset of the entire virtual instruction window. However, to ensure that all cores agree on these assignments, they must have a consistent view

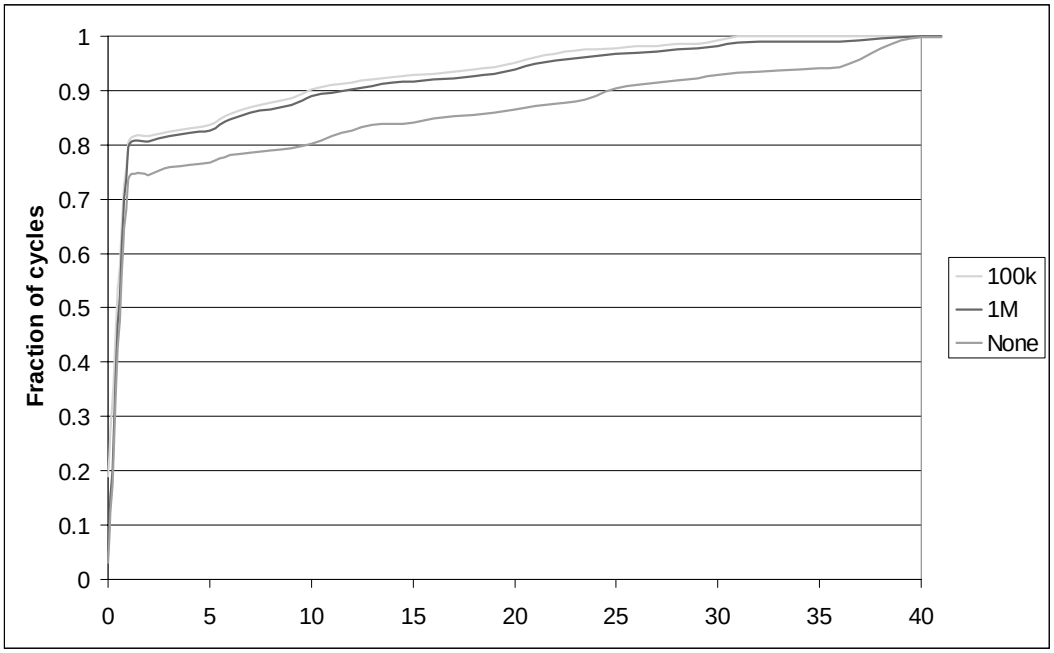

FIGURE 4-8. Distribution of Invalid Architected Registers for *applu*. Cumulative distribution of the number of architected registers marked invalid per cycle. The three data series correspond to synchronizing the entire ARF every 100,000 cycles, 1,000,000 cycles, and not at all

of which loads miss the cache and which loads hit the cache. Since execution can be skewed by thousands of instructions across cores, this is not a trivial task. A leading processor often prefetches a miss that appears as a hit to trailing processors, and conversely, a block that hits for a leading processor may be evicted before a trailing processor references it. Either of these cases can cause local GROB tail pointers to fall out of sync with each other and prevent correct register communication.

Fundamentally, this problem arises because a cache line can change state in between the time it is referenced by a memory operation on one core to the time it is referenced by the identical dynamic instruction on another core. To ensure that all cores observe cache events consistently, we implement a transient state to allow all instructions within the current virtual window at the time of the transition to observe the old state, and force all younger instructions to observe the new state. We now describe how this scheme

can be used to deal with three potentially problematic events that separate multiple instances of the same dynamic memory reference: arrival of miss data, cache evictions, and invalidate from remote stores in multiprocessor systems. Finally, we show how this scheme can enforce sequentially consistent execution in multiprocessor systems composed of skewed redundant processors.

4.5.1 Arrival of Miss Data

Skewed redundancy improves performance by allowing some cores to advance far ahead and prefetch younger cache misses for trailing cores. However if the miss is completely prefetched, the load will appear as a miss to the leading core, but appear as a hit to the trailing core. The leading core will allocate GROB space for the trailing core to write its miss slice instructions, while the trailing core will not allocate any space. This will cause the cores' mapping of local instructions to GROB entries to become inconsistent with each other, and result in incorrect inter-core register communication.

When cache block data arrives at the *miss status holding register* (MSHR), it is initialized to a transient state and tagged with the ID of the youngest instruction currently in-flight in the machine. The cache block will remain in the transient state until that instruction has retired from all cores in the machine; once this happens it is promoted into its conventional stable coherence state. Any references satisfied from the transient block before it is promoted will be marked as misses if they are older than the tag; if they are newer, they are marked as hits. With this simple tagging scheme, all cores will view each reference to the block consistently as either a hit or miss, and will allocate GROB space accordingly.

Figure 4-9 illustrates this concept. Core P1 initiates a load to 0xB but does not wait for it to complete. When the miss data returns from main memory, the youngest instruc-

tion in any window is an add to r6. The cache block is tagged with the ID of this instruction, and subsequent references by P0 that hit in the cache will compare their ID to the ID stored in the cache block. References older than the add will be treated as misses by allocating GROB entries for slice instructions. Younger references will be observed as hits, and no GROB space will be allocated.

Note that speculative execution does not pose a problem with this scheme. The youngest in-flight instruction may eventually be squashed due to a branch misprediction. However we only need to ensure that no instructions with younger IDs have issued and observed the reference as a cache miss. Defining a core's youngest instruction as its committed instruction count plus the number of in-flight instructions (i.e. ROB occupancy) conservatively guarantees this.

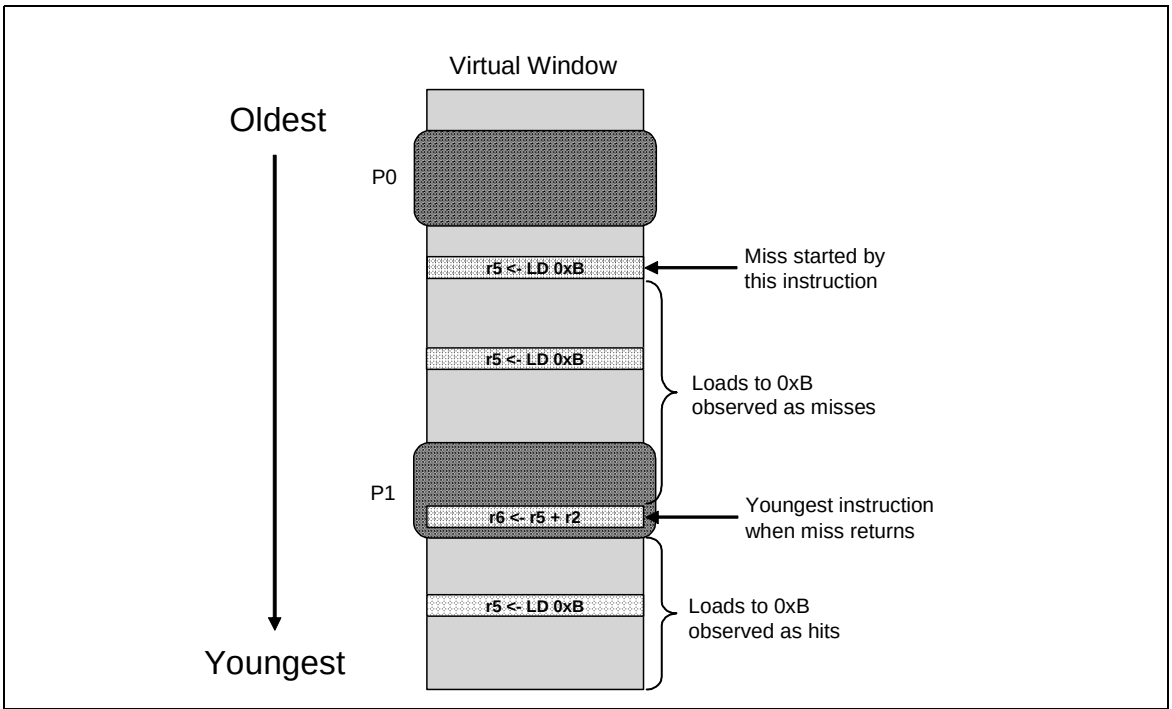

FIGURE 4-9. Observing Misses Consistently. All loads to 0xB with IDs older than the youngest instruction when the miss data returned are treated as misses; all instructions with younger IDs are treated as hits

4.5.2 Evictions

A similar case occurs when an L3 cache line is evicted between identical references by multiple cores. When a leading core hits on a block that is later replaced, it will not allocate GROB space. A trailing core may later observe that load as a miss and incorrectly write slice instruction results to the GROB. Again, this can cause the cores to disagree on which GROB entries correspond to locally executed instructions. Conceptually, the cache block needs to transition to a transient state from when it is evicted until all pending references have observed the transient state.

MSHRs already provide a mechanism to track pending cache line status by address, and can be re-provisioned for the purpose of tracking transient cache lines. In this case, an evicted line allocates an MSHR and marks it with the ID of the youngest in-flight instruction at the time of the eviction. Marking the MSHR in this manner indicates that it does not correspond to a pending L3 miss (the conventional use of MSHRs), but rather to a recently evicted line. If an L3 load miss matches a specially-marked MSHR with a younger ID than itself, another core may have previously treated a reference to this block as a hit and did not allocate GROB entries for the load miss and its dependent forward slice of instructions. We deal with this situation by squashing the entire machine (including L1 caches) and restarting execution at the oldest instruction among all cores. The trailing core first retrieves any invalid register values from the GARF. At this point it has complete architected state, which it then copies to the remaining cores. After all cores have committed instructions with younger IDs than the MSHR the MSHR is added to the free list, ensuring that subsequent accesses by any core to that address will be observed as misses.

Table 4-3 shows how often this occurs. It indicates the number of L3 evictions per

1 million instructions that occur in between the same dynamic load executed on different cores. Benchmarks with a high L3 miss rate generally have a greater number of such evictions for two reasons. First, higher miss rates simply result in more evictions. Second, applications with many L3 misses provide the most opportunity for individual cores to become skewed with respect to each other, thereby increasing the window of opportunity for an eviction to occur in between identical instances of the same load. However, even in such cases this event is exceptionally rare, and any associated squashing penalty is insignificant.

Table 4-4: Frequency of Evictions Between Identical Loads.

Application	Evictions / 1M instructions
bzip2	0.08
crafty	0
eon	0
gap	0.2
gcc	0.03
gzip	0.01
mcf	0.005
parser	0.005
perlbmk	0.01
vortex	0.03
vpr	21
ammp	0
applu	5
art	0
apsi	0.4
equake	0.4
fma3d	0.005
lucas	0
mesa	0.005
mgrid	0.4
sixtrack	0
swim	0
wupwise	0.6

4.5.3 Input Incoherence

The previous section described how L3 evictions caused by capacity misses can prevent cores from observing misses consistently, leading to problems in maintaining a consistent GROB namespace. However in a uniprocessor context, this is simply a book-

keeping problem of maintaining consistent GROB tail pointers -- each core will receive identical data from the memory hierarchy regardless of whether the load hits or misses.

This is not necessarily true in shared-memory multiprocessor systems or even uni-processors that support coherent I/O. Two instances of the same dynamic load can be intersected by an invalidate applied by a remote store on another processor. In this case, the first instance of the load will receive the old value, and the second instance will receive the new updated value. If the two values differ, this will cause a divergence in each core's execution.

The problem of diverging redundant threads caused by remote stores of shared memory locations was first observed by Smolens et al. [80]. They showed that such *input incoherence* occurs very rarely (roughly once per one million instructions), therefore squashing both windows and allowing each to observe the updated value does not impact performance.

Our scheme to deal with capacity-related evictions proposed in the previous section naturally handles multiprocessor invalidates. Similar to evicted lines caused by misses, invalidated lines caused by remote stores allocate an MSHR and tag it with the ID of the youngest in-flight instruction. Loads with older IDs that hit the MSHR trigger a global squash and restart.

4.5.4 Enforcing Sequential Consistency

4.5.4.1 Load Ordering

Input incoherence arises when a remote store causes two different instances of the *same* dynamic load to receive different values. However our technique also enforces coherence across different loads to the same address. Systems with inclusive memory

hierarchies that implement sequential consistency typically track in-flight loads in a *load queue* that is snooped when external invalidates are observed. Matching loads are squashed and re-executed. The problem with applying this approach to skewed redundancy is that a leading core may have already committed a younger load when a remote invalidate is observed, and it is therefore no longer visible in its local load queue. This case could be detected by adding a global load queue to buffer all loads within the virtual instruction window; however, the associative nature of load queues makes scaling them difficult, and searching them on every external store is likely to be prohibitive in terms of latency and power.

The same mechanism that detects input incoherence can also be applied to multiprocessor load coherence. External invalidates snoop existing private load queues in skewed redundant cores and trigger local squashes, just as they would in a conventional design. In-flight loads that do not appear in a local window have already committed in a leading core; the committed load value is either still cached by that core, or it has since been evicted. If it is still cache-resident, then the block will be invalidated by the remote store, and an MSHR will be allocated and marked according to the mechanism in Chapter 4.5.3. If it has since been evicted, the MSHR was allocated and marked at the time of the eviction according to the mechanism described in Chapter 4.5.2. In either case, older loads that match the marked MSHR will trigger a global squash and restart when they are issued by the trailing core. This combination of a local (i.e. small) load queue plus our MSHR tagging scheme will squash and replay any load that has speculatively accessed an invalidated cache line.

Prior work has shown that remote invalidates that match in-flight loads are excep-

tionally rare, even in large multiprocessor systems [14], therefore the impact of coherence squashes on performance is low. Finally, we point out that this does not represent significant additional penalty beyond what a conventional large-window design would suffer, since matching invalidates would likewise trigger squashes of at least part of the instruction window in such designs.

4.5.4.2 Store Ordering

Stores are globally ordered when they become the oldest among all cores and exit the GSQ. Private L1 caches, however, are updated when stores locally commit. We therefore must detect the case where an external invalidate arrives after a store to the same cache block has updated a L1, but before it has reached the GSQ head. Otherwise, the two stores may be applied to the L1s in the incorrect order.

To identify store ordering violations, the head of the GSQ simply needs to ensure that its corresponding cache block is not in transient state before it irrevocably updates lower levels of the memory hierarchy. It uses the identical mechanism that loads use to check for the presence of an invalidate -- the head store checks for a specially-marked L3 MSHR with a younger ID than itself. A match indicates that this store may have updated the L1s before the external invalidate was applied. All cores other than the trailing core are squashed, their L1s are flushed, and execution restarts at the oldest global instructions (in the same manner as if that instruction had raised an exception). To prevent the trailing core from committing stores before they are globally ordered with respect to other processors in the system, we impose a restriction that it must wait for the GSQ to release its head store to the L2 before it can locally commit. In practice, this does not significantly impact performance since the leading core can commit that store and younger instructions, as well

as the fact that store misses are usually ordered well before the miss completes.

4.6 Complexity

Now that we have described the major structures proposed for skewed redundancy, we consider each of them in terms of implementation size and complexity.

4.6.1 GROB

The Global Reorder Buffer (GROB) is organized similar to a conventional reorder buffer. It is implemented as a RAM where entries are inserted at the head and removed from the tail in FIFO order. However, unlike a ROB that contains all in-flight instructions, the GROB only needs to hold instructions that depend on L3 misses. This drastically reduces its number of entries and the frequency of its accesses. Instruction results are inserted whenever a core commits a slice instruction that depends on a miss that it executed. Entries are freed when all cores have committed the instruction corresponding to its oldest entry. The simplest method of re-claiming entries is for every core to announce to the GROB every time it commits a slice instruction and increments its GROB tail pointer. When all local GROB tails advance past a GROB entry, that entry is removed from the GROB head and its output value is retired to the GARF. Most instructions do not depend on L3 misses and therefore the bandwidth required for this communication is low. It can be further reduced by allowing cores to lazily announce the instruction they most recently committed (for example, by announcing the commit of only some slice instructions). GROB entries that are lazily re-claimed simply consume additional space -- they are not ever referenced by the cores nor do they affect correctness. The impact of limited GROB capacity on overall performance is examined in Chapter 6.2.1.

If the last definition of a logical register prior to the GROB head depends on a miss, its result is stored in the GARF. If the last definition does not depend on a miss, it exists (or existed) in all cores' local architected register files (ARF). As discussed in Chapter 4.6.2, the GARF could be implemented with one entry per logical register name, but could also be made considerably smaller because most instructions (and therefore most instruction results) are miss-independent. The number of registers marked invalid at any time is examined in Chapter 4.4.4.

The GARF is implemented as an additional register file, and therefore its structure and organization is similar to conventional register files. There are two features of conventional register files that can make them difficult to build. First, architectures with a single pool of renamed physical registers (e.g. MIPSR10k-style microarchitectures [99]) need to allocate a destination register for every in-flight instruction that produces a result. Therefore register files need to be large to support many concurrent instructions, and they need to be fast so that they can be accessed within a single pipeline stage. These two design constraints are at odds with each other -- it is difficult to scale such hardware structures to be simultaneously large and fast. The GARF, however, needs to be neither large nor fast. It only needs to be large enough to contain the number of logical registers that are marked invalid (or at most, the total number of logical registers). It can be slow because GARF reads only occur due to one of the synchronization conditions described in Chapter 4.3, which we showed are rare. Furthermore, the GARF is only read after the GROB is accessed and the instruction ID comparison indicates that the requested result is no longer in the GROB. This further filters accesses to it, and also enables its read latency to be

4.6.3 GSQ

The additional complexity introduced by the GSQ comes from two sources: inserting locally committed stores and comparing load addresses to those stores.

Stores: Inserting a committed store into the GSQ involves a simple RAM access indexed by the processor's GSQ tail pointer. The address field is updated with the store data (if valid). Because this GSQ access is non-associative and stores are typically removed from the critical path of execution through the addition of a store buffer, we do not expect the latency of store insertion to have performance implications. This store insertion mechanism is identical to that proposed by SRT [69].

Loads: Loads, on the other hand, are not off the critical execution path, and overall performance can be sensitive to their latency. However, unlike a conventional uniprocessor store queue used for comparison to all younger loads, only loads that miss in the private store queue and L1 cache must access the GSQ. When an L1/store queue miss occurs the GSQ is indexed by the load address and indicates whether or not it contains an older store to the same address. If it does, the load is treated as a miss and discarded; otherwise the load can access the L2 normally.

Unlike the case of store insertion, loads need to search the GSQ by address; however, its access can be overlapped with the L2 latency. In the common case the load will not depend on an earlier in-flight private store and there will be no additional penalty as long as the GSQ access latency does not exceed the L2 access latency. Techniques such as Bloom filters [10] have been shown to drastically reduce store queue searches [76], and it would be straightforward to apply a similar mechanism here. This could provide an addi-

4.6.4 Area Analysis

Any optimization that adds new hardware structures must compete for die area that could otherwise be allocated to additional processor cores or cache capacity. Here we provide a short summary of the additional area required for skewed redundancy for an architecture with 64-bit addresses, a 64-bit datapath, 32 integer registers, and 32 floating-point registers. Chapter 6.2 investigates the relationship between the size of these structures to performance. For the purposes of this analysis, we assume that the GROB contains 128 entries and the GSQ contains 128.

- **GROB**: Each entry contains three fields: an output register specifier (6 bits), a register value (64 bits), and a logical timestamp (13 bits). 128 entries x 83 bits/entry = 1328B
- **GSQ**: Each entry contains two fields: an address (64 bits) and a data value (64 bits). 128 entries x 128bits/entry = 2KB
- **GARF**: Each entry contains a 64-bit register value. 64 entries x 64bits/entry = 512B

This translates to less than 4KB of total additional storage per chip (which is insignificant given today's per-die transistor budgets), plus some additional area overhead allocated to control logic and support for associative search of the GSQ.

4.7 Power Implications

While skewed redundancy can achieve performance speedups and soft error tolerance, it also consumes additional energy by executing two instances of a program thread.

Increased transitory density, higher switching frequencies, and increased subthreshold leakage current have raised serious power and heat dissipation concerns in microprocessor design. Therefore, any technique that adds functionality or performance must be carefully balanced against its cost in terms of power and energy. The amount of additional energy consumed by skewed redundancy largely depends on the degree of clock and/or power gating applied by the processor core. Here we qualitatively discuss the two extremes of perfect and non-existent clock/power gating.

4.7.1 Fixed Energy per Instruction

Increased transistor density, faster switching speeds, and the desire to place more cores in a fixed die area have lead to aggressive power management techniques at both the circuit and architectural levels. *Clock gating* is a common approach that inhibits distribution of the clock signal to idle transistors; another, heavier-weight approach can also reduce static leakage power by additionally cutting the supply voltage as well *(power gating)*. The mismatch between processor and memory performance often leads to significant stall times while cores wait for cache misses to complete; therefore, these techniques can be effective at reducing power and energy consumption.

If we assume ideal clock and power gating, then the processor consumes no energy while stalled and the total energy to execute the program (i.e. the *energy per instruction*, or EPI) remains constant regardless of execution time[1]. In this case, skewed redundancy will consume roughly twice the core energy of unreplicated execution, plus some additional energy from the global structures introduced in this chapter (i.e. the GROB, the GARF,

1. Multiple instruction issue performed by superscalar processors may in fact cause the EPI to vary during non-idle periods. However we assume a fixed EPI as a first-order approximation.

and the GSQ). Since these structures are small and accessed infrequently, their overall contribution to system power is likely to be minimal. Furthermore, since unreplicated portions of the system consume no additional energy (e.g. main memory, buses, etc.), the total increase is likely to be substantially lower than 2x.

4.7.2 Fixed Energy per Unit Time

At the other extreme, designs that do not utilize power management techniques (e.g. clock/power gating, frequency throttling, etc.) dissipate constant power regardless of whether the processor is idle or not. Therefore, longer program execution necessarily consumes additional energy. Under this pessimistic assumption, skewed redundancy (using two cores) requires less than double the energy of unreplicated execution. If the resulting speedup is large enough, this can potentially reduce the energy required to execute a program. For example, Chapter 6.1 will show that skewed redundancy enables some floating-point benchmarks to execute in half the time of unreplicated execution (i.e. 100% speedup). If each of the two cores expends constant power, total energy will be reduced.

4.7.3 Realistic Designs

Reality exists between these two extremes. In practice, it is difficult to apply perfect clock gating within a pipeline, and the high startup overhead suffered by power gating makes it ill-suited for precise power-management control. Furthermore, linear decreases in channel length lead to exponential increases in static power dissipation caused by subthreshold transistor leakage. In the absence of power gating, longer execution times result in additional static energy consumption; therefore, faster performance can reduce static energy. As subthreshold static leakage becomes a higher fraction of total power dissipa-

tion (static power dissipation is predicted to exceed dynamic power dissipation within several processor generations [13]), the speedups enabled by skewed redundancy can reduce its energy footprint. Finally, we point out that, regardless of power saving techniques or the relative contribution of static leakage, prior redundant multithreading schemes minimally consume twice the core energy, since their execution time exceeds (or at best, equals) that of non-redundant execution.

4.8 Putting It All Together

Figure 4-10 presents an updated system-level diagram of a skewed redundant system. It shows two cores and their L1 caches combined into a single entity. Comparing it to Figure 4-1, we point out the additional shaded structures that this chapter has described: the GROB, GARF, and GSQ. Each is relatively small, scalable, and outside of the processor cores. The remaining CMP cores are not shown, and would appear on the right side of the figure where they would be conventionally connected to the on-chip bus between the L1 and L2 caches.

4.9 Beyond Two Cores

So far, we have described skewed redundancy primarily in the context of two processor cores. However, the mechanisms we have proposed scale to an arbitrary number of cores that each redundantly execute a program. This section shows how additional cores can further improve performance.

When only two cores are available, the first to uncover a miss starts the memory access, but does not stall while waiting for it to complete. If the second core significantly

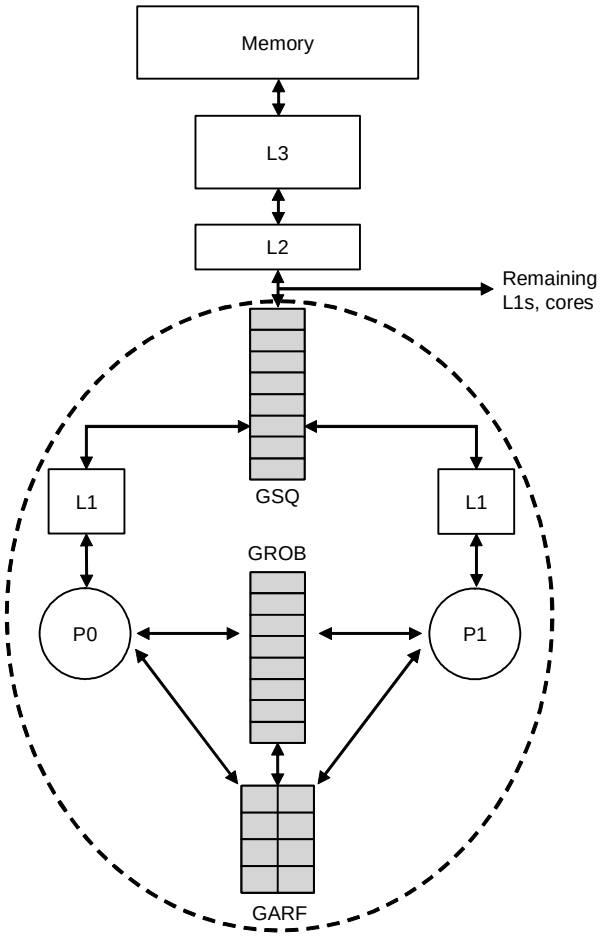

FIGURE 4-10. New CMP System Diagram. Two cores and their private L1s combine to form a single skewed redundant processor (remaining CMP cores not shown). Proposed hardware additions are shaded

trails the first (indeed, the goal of skewed redundancy), many cycles may elapse before it executes instructions skipped by the first. This situation is particularly pronounced when L3 misses depend on prior L3 misses, which can be common in pointer chasing code through linked list data structures. The initial miss is prefetched by the first core, but subsequent dependent misses will not be executed until they enter the trailing core's instruction window.

A third core can help by issuing miss slice instructions early. As before, the first core to reach a miss discards it. Preventing stalls in the leading core maximizes the size of

the virtual window, and we never force it to stall unless absolutely necessary. The second core has the option of waiting for the miss to complete, or to similarly discard it and allow the third core to handle it. Always allowing the second core that reaches a miss to discard it will clearly cause it to behave identically to the leading core, and offers no performance advantage. Conversely, always forcing the second core to handle the miss can result in the two trailing cores continually “leap-frogging” each other and proceeding at similar rates. If the misses are temporally clustered together (and would therefore likely fit into a single instruction window), then partitioning them among two cores in this way will not improve performance. A better alternative would allow one to advance ahead and only stall on misses that are likely to be outside the window of the other trailing core.

We therefore propose a simple heuristic that forces the second core to stall and wait for a miss only when it is ahead of the trailing core by some minimum number of instructions. The policy we adopt sets this minimum distance at twice the ROB capacity. This allows the second core to discard the miss in cases when the third core is likely to reach it shortly; once it has advanced sufficiently far ahead it is eligible to wait for subsequent misses.

Figure 4-11 illustrates this heuristic. P0 is stalled waiting for a cache miss to the address in register r4 to complete. P1 and P2 did not stall on the miss and continued executing and committing instructions until they reach a second miss, this time to the address in register r5. P2 is the leader and does not wait for it to complete and continues. This second miss is separated from the first by at least twice the ROB size (e.g. at least 128 instructions in a 64-entry ROB), therefore P1 waits for the miss to complete, then soon executes the following miss to the newly loaded address in r5. Allowing P1 to wait can

have two performance advantages. First, it can start dependent misses early (such as the second load from the address in r5). If such misses did not issue until P0 reached them, they could slow P0's progress to the point that the remaining cores cannot "slip" any further ahead (for example, because GROB or GSQ resources have been exhausted). Second, if there is a forward dependence from the miss to instructions executed by the leader, the leader will stall until they complete. This case is also shown in the figure. P2 executes a branch that depends on an L3 miss; it therefore cannot commit the branch until the source register (r5) is computed and copied into the GROB. Allowing P1 to advance ahead of P0 and then execute an independent miss slice accelerates the backward branch slice computation. Without the third core, P3 would suffer additional stall cycles while waiting for the sole trailing core to reach the miss. Chapter 5.4 extends this discussion to show that a third core can be used to add error recovery capabilities in addition to increased performance. A discussion of the performance implications of adding a third core will be presented in Chapter 6.8.

4.10 Summary

This chapter presented on overview of skewed redundancy and outlined its basic principles. It described the mechanism that cores use to occasionally transfer miss slice results amongst themselves, and the conditions under which such communication occurs. One of the most challenging aspects of out-of-order execution is correctly enforcing memory dependences; to that end this chapter also detailed how both uniprocessor and multiprocessor memory ordering is maintained. A spectrum of characterization data was presented to indicate the fraction of instructions that participate in inter-core synchroniza-

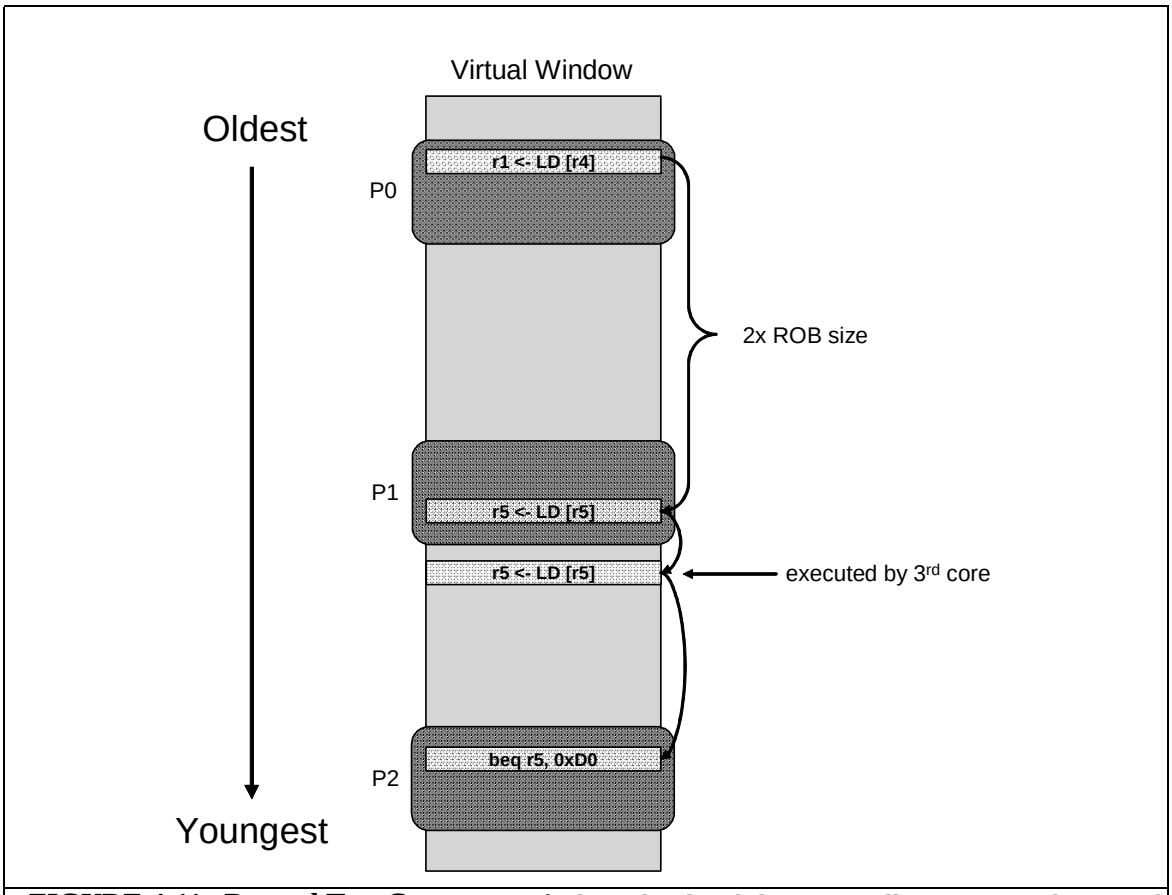

FIGURE 4-11. Beyond Two Cores. P2 is farthest ahead and does not stall on misses. The second load to [r5] begins early if P1 is permitted to wait for the prior miss to complete.

tion as well as the reasons that synchronization occurs. Finally, we briefly discussed the minimal die area costs required for the additional global data structures introduced by shared by skewed redundancy.

Chapter 5

Detecting and Tolerating Soft Errors

5.1 Soft Errors

A *soft error* (or *transient error*) occurs when the state of a digital signal unintentionally changes from 0 to 1 or 1 to 0. They can be caused be a number of external sources, including electrical noise, electromagnetic interference, and most-commonly, cosmic radiation. Unlike hard errors, which refer to permanent physical damage in the underlying circuit, soft errors are transient in nature, and do not imply the presence of a lasting physical defect. We use the term “error” to indicate that the single-event-upset has propagated to an observable program output, as opposed to a fault, which can be masked at the logic or gate level.

Although the existence and causes of soft errors in digital logic have been well-known since the advent of dynamic RAM in the 1970s, they are becoming increasingly problematic in modern system design for several reasons. Shrinking supply voltages, feature sizes, capacitances, and noise margins lower the energy threshold required to cause a bit flip. Decreasing feature sizes additionally leads to higher transistor density, further increasing the likelihood of an error within a fixed die area. Finally, as we grow more reliant on computer systems in everyday living, we become less tolerant of downtime and have higher expectations of system availability.

A common approach to detect soft errors is to introduce redundancy into a design. Redundancy can be implemented in many forms: information redundancy stores duplicate (but often lossily compressed) information in the form of *error-correcting codes* (ECC) or

parity. It is typically very effective at protecting storage structures from soft errors, and has traditionally been the mainstay of error protection techniques. However it is less amenable to protecting combinational logic, which has recently become more vulnerable to single-event-upsets. Designers have therefore turned to spatial redundancy (which re-executes an operation on physically separate hardware components and compares the results) or temporal locality (which performs re-execution on the same component at different times).

This chapter shows how spatial and temporal redundancy can add soft error detection capabilities to our proposal for skewed redundancy. Because we already exploit redundancy to improve performance, fusing it with error-detection requires only minor modifications to our design. We also show how adding a third core (proposed for increased performance in Chapter 4.9) can add error correction as well.

5.2 Detecting Errors

Because skewed redundancy non-speculatively executes identical copies of a program on multiple processor cores[1], it provides a natural mechanism to detect soft errors. Single-bit errors can be identified by comparing the output of identical instances of the same dynamic instruction executed by different cores. Although minimizing inter-core register communication is crucial in achieving high performance, every store updates the shared Global Store Queue (GSQ) as it commits. Because all stores filter through this centralized structure before updating remaining levels of the memory hierarchy, the GSQ is

1. In the common case cores commit instructions non-speculatively. In rare cases, a core will be squashed and restarted. Examples of these events were presented in the prior chapter and include precise exception recovery and multiprocessor memory ordering violations.

5.2.1 Checking Stores

True lock-stepped execution has appeared in a number of commercial systems and can both detect and correct soft errors [53][77][79]. It redundantly executes a thread on multiple processors and compares every instruction result. It suffers considerable overhead because no instruction can commit until its identical pair has also completed and been verification. This inefficiency is particularly pronounced in existing non-CMP implementations of lock-stepping. Processors in such systems occupy physically separate chips, both increasing the verification latency and preventing it from being pipelined (because instructions are only checked when they become the oldest in each processor's window).

Prior work has shown that 100% soft error coverage can be achieved by only checking the results of stores, alleviating systems from expensive instruction-by-instruction lock-stepped execution [69]. Because stores are the only observable output of a program, every instruction either eventually propagates to a store or is dynamically dead (in which case it will not produce a visible error if it executes incorrectly) [12]. This reduces the number of output comparisons with the redundant thread; however, it also complicates error recovery. An erroneous instruction may have already committed and updated the architectural state of the processor long before it is manifested in a store data or address mismatch. Even though the error has been contained within the processor core and has not affected system memory, the core cannot be restored to its pre-error state other than by restarting the program thread. Only checking stores has the added benefit that it will not detect errors that are later masked by subsequent instructions in the dependence chain.

Finally, we point out that only checking the results of committed instructions (stores, in this case) does not expose non-architectural events that occur within the pipeline -- for example, the process of passing values and comparing instruction results executed on different cores does not depend on identical branch prediction behavior or the order that memory accesses are resolved (as might be required for pin-level output comparisons).

Existing redundant multithreading (RMT) techniques detect errors in this manner by only checking store outputs. While they achieve higher performance than per-instruction lock-stepped execution, they still suffer a slowdown compared to the original unreplicated case because no store can commit and update the L1 data cache until it has been verified. Skewed redundancy, on the other hand, permits stores to commit and update the private L1 and GSQ immediately. Stores are released from the GSQ and update the shared L2 after all cores have committed younger instructions.

There is no fundamental reason why RMT cannot similarly buffer committed stores and move error checking off of the core's critical execution path. However this would likely provide little benefit. It will decouple threads' execution from each other by avoiding synchronization on all stores, but without any mechanism to allow them to slip with respect to each other, both will advance at approximately the same rate. They will stall on the same cache misses and overall instruction throughput will approach that of unreplicated execution. Skewed redundancy both decouples cores from each other (because it allows stores to commit locally before they are verified) as well as provides a means for them to advance at different rates (because last-level cache misses stall only a single core).

Skewed redundancy executes redundant threads on separate CMP cores. The previous chapter presented the GSQ as a performance-enhancing technique; here we show how it can be also used to check store results. Conceptually, it is similar to the non-coalescing store comparator queue proposed in the original RMT work [69].

The purpose of the GSQ is to order identical stores committed by each core and release them to the shared L2 in their original program order. Each core minimally indicates that it has reached a particular store, and at least one core updates the entry with the store address and data. This scheme can be extended to include error detection by simply forcing each store to compare its address and data to that already written in that entry (if any). The first core to commit a dynamic store writes its address and data (if valid) into the GSQ. Subsequent instances of that store compare those fields as they locally commit; any discrepancy indicates the presence of an error. Although this provides only error detection (not correction), the error is contained within the cores, their private L1s, and the GSQ.

5.2.3 Sphere of Replication

The *Sphere of Replication* (SoR) represents the logical boundary of redundant execution within a system [69]. Everything inside of the SoR is replicated (either in time or space) and can detect single-bit soft errors. Recent work by Aggarwal et al. has shown that many resources in CMPs are not within the SoR, and can become single points of failure that reduce the effectively of techniques such as redundant threading [2]. The SoR of the implementation of skewed redundancy described in this work encompasses the processor cores and their private L1 caches. Therefore, errors occurring in other on-chip structures must rely on orthogonal techniques to provide additional coverage. Furthermore, this

implementation introduces three shared, global structures (the GROB, the GARF, and the GSQ) that create additional exposure. However, since these are all relatively small storage structures, their error rates are low to begin with and can be lowered substantially with ECC, hence their impact on the chip's overall vulnerability is not significant.

5.3 Exploiting Temporal Redundancy to Increase Coverage

Most instructions globally execute on all processor cores; the previous section described how this spatial redundancy can be exploited to detect soft errors. However, instructions that depend on a last-level cache miss are only executed by a single core, and therefore cannot be checked for correctness by this method. Skipping past long-latency instructions in this way increases performance, but also reduces error coverage. If an error corrupts a forward miss slice instruction, any stores in its data dependence chain cannot be compared to stores previously inserted into the GSQ.

Applications with many last-level cache misses can have a significant fraction of private instructions. Figure 4-6 shows that such instructions can exceed 20% in some cases, leading to unacceptable reductions in error coverage. To increase coverage to 100% within the sphere of replication, we leverage a technique that exploits *temporal redundancy* to detect errors.

This scheme, proposed by Ray et al. [67], modifies the issue logic in a superscalar datapath to insert an additional copy of each instruction into the back-end of the pipeline. Both the original and duplicate instruction perform the same calculation and compare outputs; a mismatch indicates an error, at which point the pipeline can be flushed and restarted. As originally proposed, this technique suffers a significant performance penalty

compared to the baseline unreplicated case because twice as many instructions execute, in addition to instruction checking and synchronization overhead.

Figure 5-1 indicates the relative performance loss when error checking is implemented in our baseline out-of-order model by dual-issuing all instructions and confirms the results presented in [67]. Nine of the SPECINT2000 and nine of the SPECFP2000 benchmarks suffer at least a 20% performance degradation. Although the superscalar out-of-order microarchitecture is able to hide some of the execution latency of the additional instructions (reflected by the relative performance exceeding 0.5), the performance penalty is significant nonetheless.

Two properties of skewed redundancy drastically reduce this cost and make dual instruction issue an appealing technique: first, most instructions execute on multiple cores and therefore do not need to be dual-issued because they exhibit spatial redundancy. Second, even if duplicately-issued instructions slightly slow down one core, the other core can continue executing and uncovering cache misses.

We integrated dual-issuing of instructions in our skewed redundancy model as follow: after miss slice instructions are fetched and decoded, two copies are inserted into consecutive issue queue slots. The output register of the first copy is converted to a special register name added for the purpose of error detection. The second copy is modified to include this register as an additional source operand. Both copies execute the operation; the second copy also compares both outputs to identify soft errors. Figure 5-2 illustrates this technique. The left side represents the original unreplicated instruction. The right side shows the resulting transformation after instruction 3 is duplicated. Register r7 is replaced with special register rx in the first copy (3a). This additional register is supplied as a third

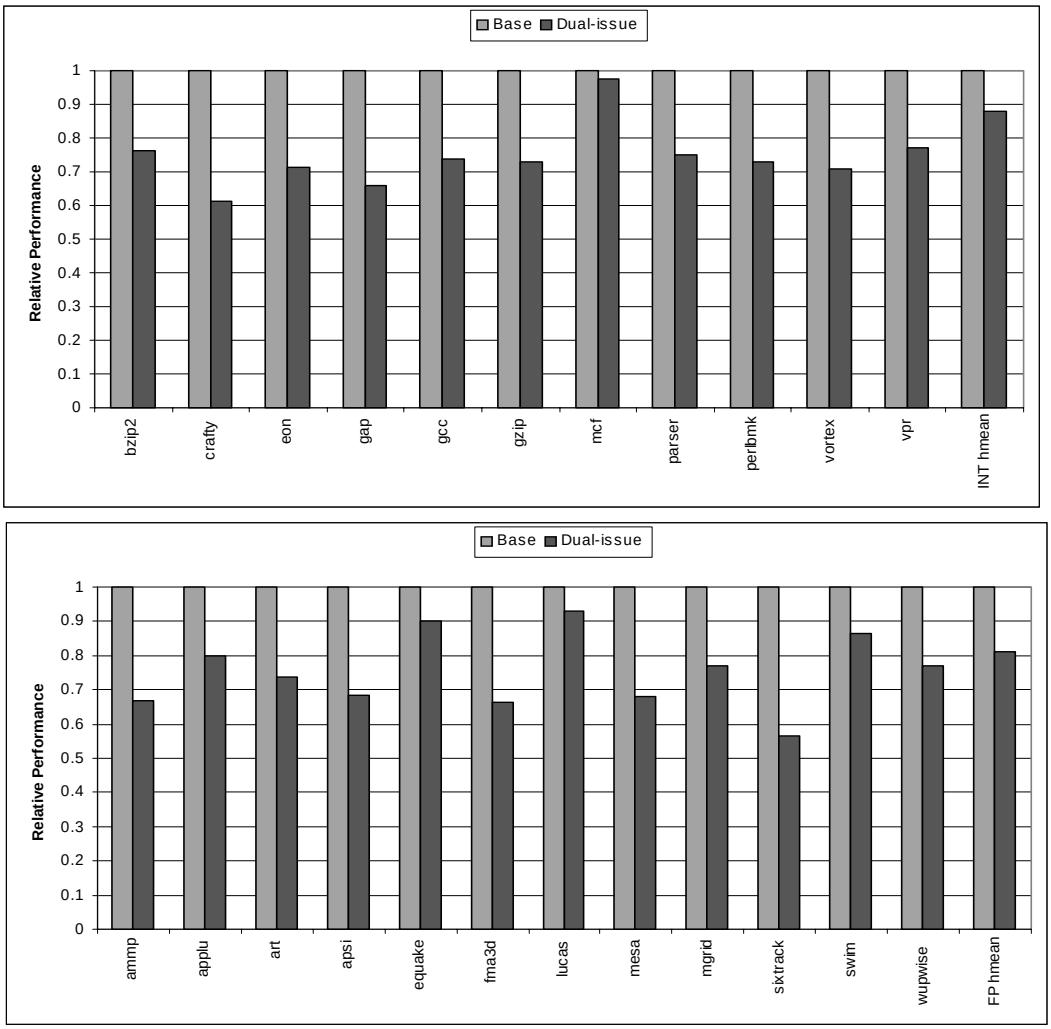

FIGURE 5-1. Slowdown Due to Dual-issuing All Instructions.

input to the second copy (3b). Although this requires a separate register name to communicate the data dependence between the two copies, it does not necessarily require additional capacity in the architectural register file (ARF). When the first copy completes, the second is guaranteed to already reside in the issue queue. Therefore rx can be broadcast directly to the consuming copy without writing to the ARF.

When a miss is detected, one or more instructions from the dependent forward slice may already be in the window. These instructions need to be identified and tagged for duplicate issue. Instructions that are already in flight can be identified with the same

mechanism used for recovery from speculative scheduling [43], while newer instructions inherit this property through the renaming logic.

We point out that our implementation of dual-instruction issue is slightly different than that proposed by Ray et al. Their scheme does not require additional physical register namespace because it infers data dependences from the location of instructions within the ROB -- even indexes hold the original instructions and odd indexes hold the duplicates. Because we only dual-issue slice instructions, even/odd indexes will become unaligned by single-issue instructions. A simple alternative to enable the use of this technique as originally described could pad the ROB with no-ops in order to consistently align slice instructions.

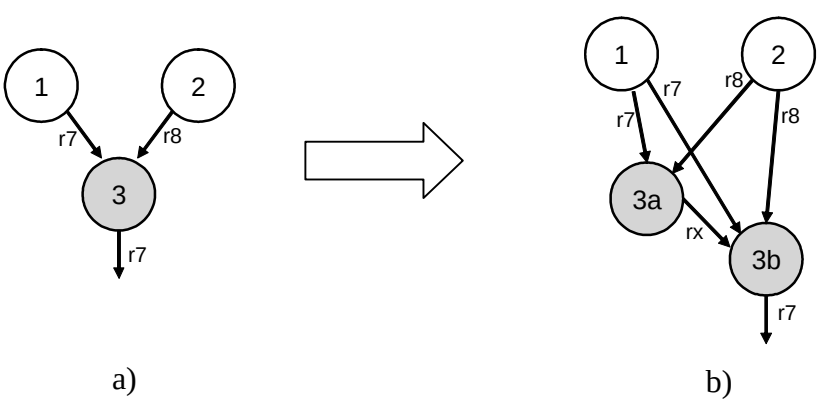

FIGURE 5-2. Dual-issue instruction dependences. a) Original execution. b) Instruction 3 is split into 3a and 3b. Register rx creates a data dependence between the two copies.

5.3.1 Detecting Errors in Global Inputs to Slice Instructions

As the previous section described, miss-dependent instructions do not exhibit spatial redundancy and are therefore dual-issued to provide temporal redundancy within a core. However, a miss-dependent store may also depend on globally-executed instruc-

tions. If these global instructions do not terminate in a globally-executed store, they will not be checked for errors. This section describes extensions to our mechanism to handle such instructions.

For example, instruction 1 in Figure 5-3 is a load cache miss. Any instruction in its *forward miss slice* will be dual-issued and therefore checked for transient errors (i.e. instructions 1, 3, and 5). However, instructions 2 and 4 are globally-executed and fall within the *backwards slice* of the store that terminates the forward miss slice. An error in instruction 2 will not be detected because it neither terminates in a redundantly-executed store nor is dual-issued. We therefore need to provide redundancy in some other way to achieve 100% soft error coverage.

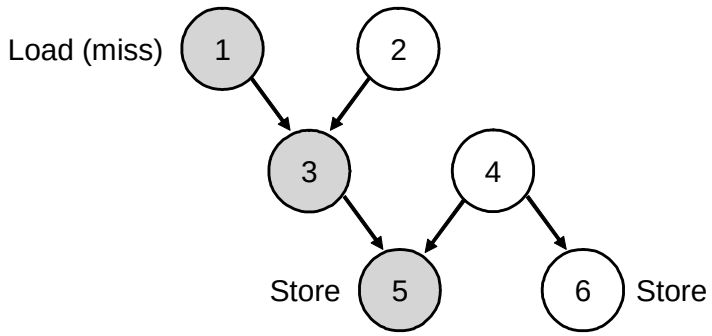

FIGURE 5-3. Protecting Global Slice Instructions. Shaded instructions depend on cache misses; clear instructions are globally executed. The first core to reach instruction 3 writes the source operand produced by instruction 2 to the GROB. Trailing cores read this value and compare it their own local source operand.

The GROB already implements a communication mechanism, and we propose modifying it to accommodate globally-executed operands that fall within the backwards slice of private stores. Instead of containing a single instruction result (as proposed by Chapter 4.3.1), we extend the width of each entry to hold an additional register value. The first core to commit a miss slice instruction with a global source operand writes the global

value into this additional field in the entry pointed to by its local GROB tail. The trailing core(s) will retrieve this value from the GROB and compare it to their own; a mismatch indicates that an error occurred, inducing a trap that causes the operating system to terminate the program thread.

Closing this gap in error coverage adds to the width of each GROB entry. Fortunately, as the following chapter indicates, limiting the GROB to as few as 32 entries does not severely impact performance. This technique can also increase GROB traffic, since in our previously description, cores with invalid operands incremented their GROB tail pointer, but did not need to access the GROB itself. Now they also need to write (or check) global operands as well. However because this occurs off the critical path (i.e. writes and checks can occur after instructions commit), it is unlikely to affect performance. Furthermore, since the written values are globally executed and therefore inserted by the leading core, trailing cores will never need to wait for the global operand to be produced.

Referring again to the example in Figure 5-3, the core responsible for handling the cache miss caused by instruction 1 will detect one private and one global operand when it executes instruction 3. It will write the global operand (produced by instruction 2) to the GROB and continue. Other processors will discard the miss, and will not execute instructions 1, 3, and 5. Before discarding instruction 3, they will identify the fact that a global register feeds into a discarded miss slice, and will retrieve the left operand produced by instruction 2 from the GROB. Comparing this register value to their own provides spatial redundancy and will detect errors.

Finally, we point out that even though instruction 4 eventually propagates to a globally-executed store (instruction 6), and therefore can rely on the GSQ to detect a soft

error, this may not be known when instruction 3 executes. We therefore conservatively write the source operand produced by instruction 2 to the GROB as just described.

5.3.2 Detecting Errors in Front-end Slice Instructions

Re-issuing miss slice instructions exploits temporal redundancy within a core by re-executing them in the functional units. However it does not add redundancy to the front-end pipeline stages, leaving the processor vulnerable to soft errors that occur during instruction fetch and decode. To compensate for this lost coverage, we fall back on the GROB to exploit spatial redundancy. Even though processor cores do not execute instructions with invalid source operands, they still fetch and decode all instructions, which they can compare against corresponding instructions fetched by other cores.

To detect front-end errors, we introduce an additional field to each GROB entry to hold decoded miss slice instructions. When the leading core commits a slice instruction, it copies the instruction components (i.e. opcode, register specifiers, immediate values, etc.) into the GROB entry pointed to by that its GROB tail. Because the instruction already exists in decoded form in the local ROB, it can simply be copied to the GROB. This process creates a “skeleton” of slice instructions that does not yet necessarily contain their computed results. Trailing cores read the decoded instruction fields from the GROB as they commit identical instances of those instructions and compare them to their own values. A mismatch indicates that an error occurred in either the fetch and/or decode stage, and can be treated similar to a store mismatch in the GSQ.

Expanded GROB fields are shown in Figure 5-4. Chapter 4.6.4 presented an area estimate of the GROB; we update it here to accommodate the additional fields:

- **AgeID:** a 13-bit counter can support a virtual instruction window of 4096 instruc-

tion (including an additional bit to deal with counter overflows [37]).

- **Instruction:** 32 bits (pre-decoded) in most modern architectures. Assuming a 50% overhead to store in decoded form results in a total of 48 bits.
- **Global Source:** 64 bits
- **Output Value:** 64 bits

Each entry therefore occupies 24 bytes of space. Chapter 6.2.1 examines the relationship of GROB size to overall performance. Conservatively assuming a maximum capacity of 128 entries (which achieves nearly all of the performance of the unlimited case) yields a total GROB size of 3KB of random-access storage.

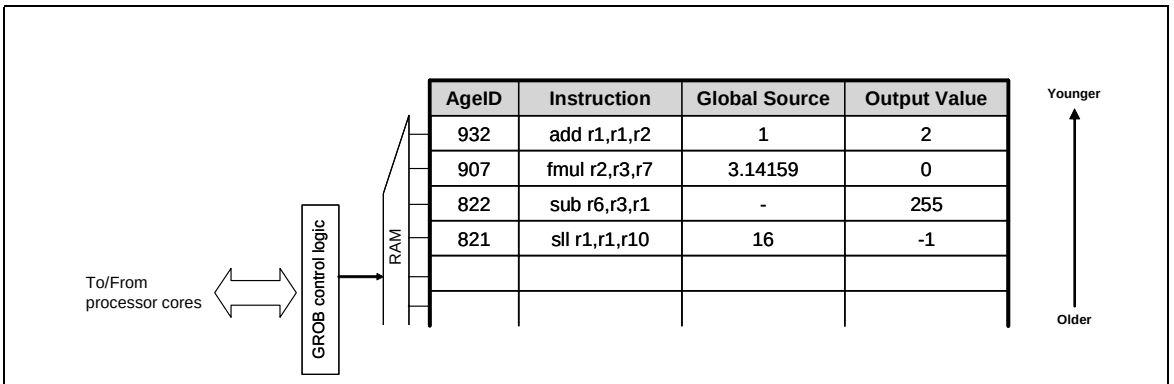

FIGURE 5-4. GROB Fields. *AgeID* is a dynamic instruction count used for relative age comparisons between instructions. *Instruction* is the decoded machine instruction. *Global Source* contains any global source operand values (used for error detection and slice instruction replay). Not all slice instructions contain global sources (e.g. the instruction corresponding to ID 822 sources only miss-dependent values). *Output Value* holds the instruction result (for inter-core communications). The *GROB control logic* tracks the commit point of each core.

5.3.3 Detectable Faults

Although dual-issuing instructions within a pipeline provides soft error coverage for those operations skipped by at least one core, such coverage is arguably weaker than that supplied by exploiting spatial redundancy across cores. For example, hard (or permanent) faults can be detected in instructions duplicated across cores, but may not be

detected in instructions dual-issued within a core (e.g. if both copies of the instruction are scheduled onto the same faulty functional unit). Similarly, transient errors that fan out to both copies of an instruction (either because the error affected a non-replicated portion of the pipeline or because the error spanned multiple clock cycles) can also lead to silent data corruption. Finally, comparing the results of redundant cores can detect any type of error that affects program output, while exploiting temporal redundancy within a single superscalar pipeline may not detect multiple-bit errors that identically corrupt both copies of an instruction. Although the ability to detect some instances of multi-bit errors can be useful, the probability of such errors is generally quite small, and the inability to detect those that affect dual-issued instructions does not significantly impair error coverage.

5.4 Error Recovery

The previous chapter showed that adding a third core can increase performance by allowing cache misses with miss-dependent addresses to execute before they are reached by the trailing core. This section shows how adding a third core can enable skewed redundancy to recover from errors in addition to detecting them.

Comparing the output of stores executed by two cores provides dual-modular redundancy (DMR) and can detect single-bit errors. However it does not indicate which of the two cores is faulty. Because the error may have occurred arbitrarily far in the past, re-executing in-flight instructions does not guarantee recovery. Redundantly executing a program on three cores, on the other hand, provides triple-modular redundancy (TMR), and can be used as a voting mechanism to identify the correct value[1].

The principles that allow three cores to execute the same program thread are essen-

tially identical to what we have described for the case of two cores, with the exception that now two separate store comparisons occur. The first core to execute a store writes its address and data into the GSQ, as before. The second core to reach the same dynamic store instance reads the store address written by the leading core and compares it to its own value. If the store data is globally computed, it is read and compared as well (private store data is checked for errors by the mechanism proposed in Chapter 5.3). A mismatch indicates that one of the two cores is faulty. To determine which core is faulty, both cores stall and wait for the trailing core to execute the store and perform the same comparison; a minority vote indicates an error. If the leading two cores produce the same result and a mismatch instead occurs in the comparison by the third core, then the third core is erroneous (given the single-bit error model assumed throughout this thesis).

At this point the corrupted core has been identified; unfortunately, resetting the store data or address register to the most popular value is not sufficient to correct the error because it may have occurred in an older instruction and since fanned out to an arbitrary number of younger consumers. Reversing the effects of the instructions affected by the error and rolling the core back to a known good state is inherently difficult without a heavy-weight checkpointing mechanism; a much simpler alternative is to simply squash and restart the core that has the minority vote. In this case, one of the two good cores can reconstruct architected state in a similar manner used to recover from a precise exception (Chapter 4.3.5) -- any register values marked invalid can be recovered from the GROB or GARF and the complete state can be transferred to the remaining two cores. However, this

1. Other forms of redundancy could also identify the erroneous core. For example, adding parity protection to each datapath will enable faulty cores to identify themselves and potentially eliminates the need for a third core. We leave exploration of such designs to future work.

technique only works if the error does not affect the trailing core. If the error does affect the trailing core, it can prevent it from making forward progress and from producing the required slice instruction result. We therefore require an alternative mechanism to obtain the missing private register values owned by the faulty core.

To deal with the case of an error in the trailing core, we propose replaying skipped miss slice instructions. Recall from Chapter 5.3.2 that the GROB already contains all decoded slice instructions within the virtual window. One of the two leading cores is assigned to reconstruct architected state (called the *reconstruction core*), and the remaining two cores are squashed.

The reconstruction core needs to compute any values marked invalid in its local ARF and merge those values with its existing state. It first saves its current register state to memory and marks all of its general-purpose registers invalid. It then recomputes all miss slice instructions in the virtual window older than its last-committed instruction by starting execution at the instruction corresponding to the oldest GROB entry. Slice re-execution is possible because global (i.e. miss-independent) slice inputs are stored in the GROB as described in Chapter 5.3.1, and private (e.g. miss-dependent) inputs reside in the GARF. As slice instructions execute they populate the local ARF with their results, which can be subsequently used by younger slice instructions. This represents all information required to reconstruct values produced by skipped slice instructions that have been marked invalid in the ARF.

Figure 5-5 illustrates the error recovery algorithm. The machine instruction field is first read from the GROB head. If a global input operand is associated with this entry, then its value is restored to the appropriate source register. If no global operand exists, then the

source is private, and either resides in the ARF or GARF. The core first looks up the source register in its local ARF; if it is not still invalid, then it has been updated by a slice instruction since the recovery process started and that value is used. If the register is marked invalid, then the correct value was written before slice execution began, and is retrieved from the GARF.

Successive GROB entries are similarly executed until the current GROB index equals the local GROB tail of the reconstruction core. At that point, the reconstruction core retrieves registers that it had originally marked invalid at the time the error was detected. Each missing register name is first looked up in the ARF -- a valid entry indicates a write by a slice instruction re-executed from the GROB -- or the GARF if that register is still marked invalid in the ARF. These values are then merged with the reconstruction core's architected register state (which was saved to memory prior to slice re-execution). This represents the complete architected register state of the system, and is restored to all three cores' register files. The error has been corrected and each core starts fetching at the same instruction address.

Figure 5-6 provides a high-level example. Trailing processor core P0 detects a mismatch when it compares the address and data of its oldest store to the GSQ. Because the other two cores have already verified the store with each other and advanced past it, P0 is known to contain an error. P1 is arbitrarily assigned to be the reconstruction core (we could have alternatively picked P2), and P0 and P2 are squashed. P1 saves its register state to memory, and executes all slice instructions in the virtual window that it skipped and that are older than its oldest instruction (called the *recovery point*). Execution starts at the oldest instruction in the GROB (P0's load to the address in r9) and proceeds until it reaches

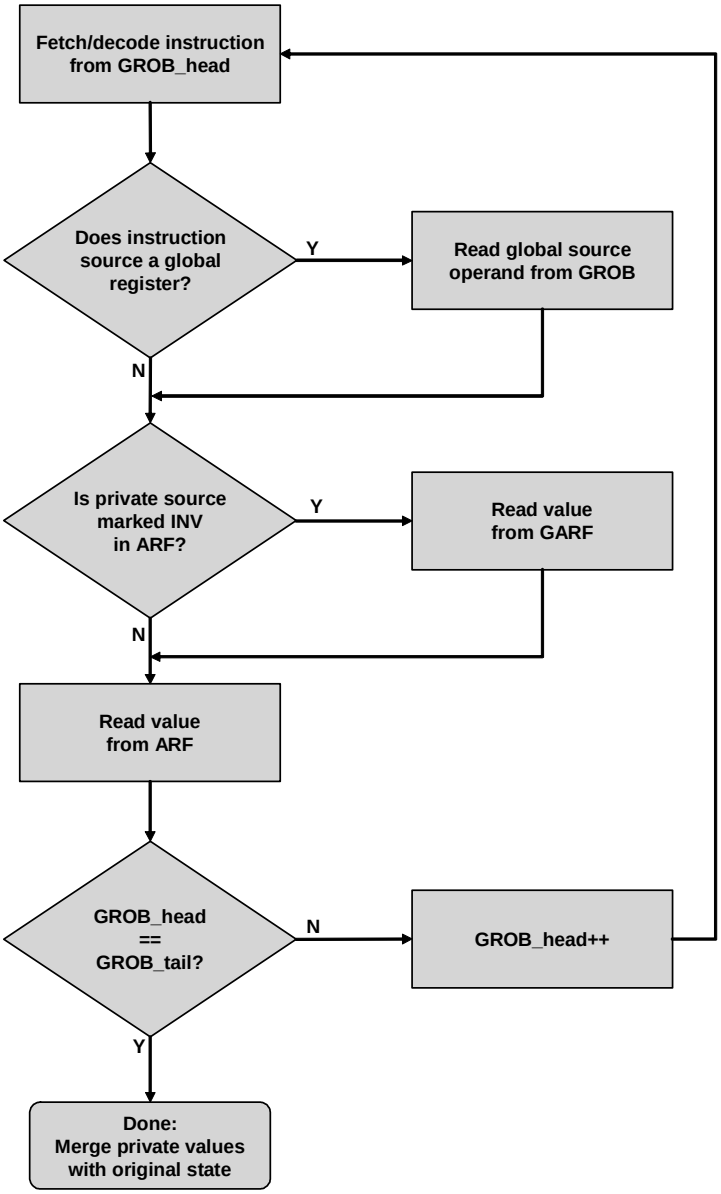

FIGURE 5-5. Error Recovery. Algorithm followed by reconstruction core to compute missing slice values that were never produced by an erroneous trailing core. After the slice values are computed, they are merged with the core's original register state, then copied to the remaining two cores.

the recovery point. Instruction source operands are retrieved from the GARF (if they were produced by instructions older than the oldest GROB entry), the ARF (if they are produced by slice instructions in the virtual window), or the GROB (if there are globally-available slice inputs previously copied to the GROB). Finally, P1 copies its reconstructed

register state to the remaining two cores, and all three cores continue fetching at the instruction address corresponding to the recovery point.

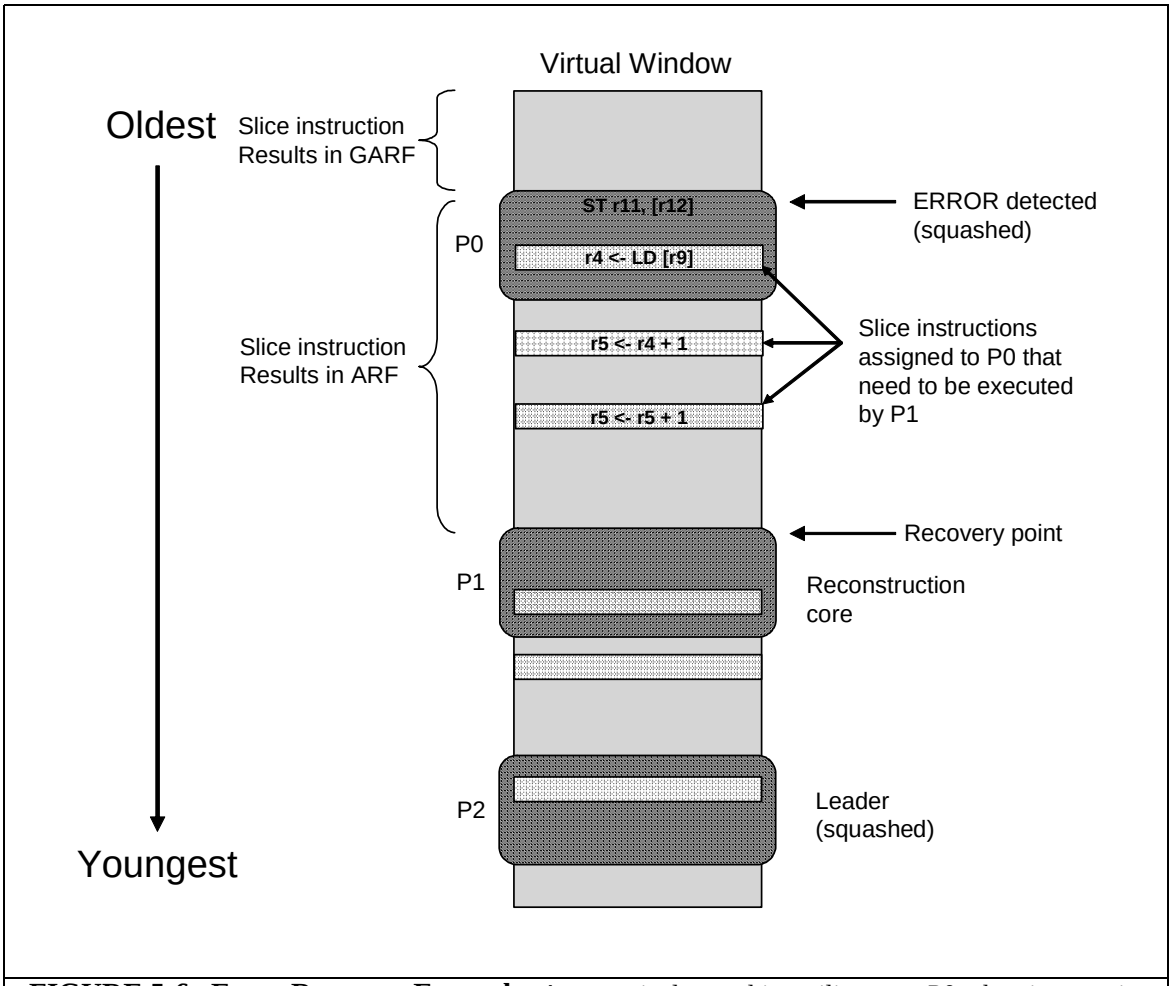

FIGURE 5-6. Error Recovery Example. An error is detected in trailing core P0 when it commits a store. P1 is assigned to be the reconstruction core; it executes slice instructions (highlighted with rectangles) it skipped before merging them with its own register state.

This recovery process involves several steps, but it is only invoked on the rare occurrence of an error in the trailing core. The speed at which it occurs is therefore not critical, and it can be assisted by or completely handled in software (i.e. saving original register state to memory, re-executing the slice instructions, merging the slice results with the saved register state, copying the recovered state to the remaining two cores, etc.). Finally, we point out that the technique of buffering slice instructions and their inputs for delayed execution has been proposed by Srinivasan et al. in their *slice data buffer* (SDB)

5.5 Chapter Summary

This chapter proposed extensions to the global data structures presented in the previous chapter to enable soft error detection and recovery. It started by showing how the GSQ can detect errors in a system with two skewed redundant cores. To compensate for the reduction in error coverage caused by non-redundant execution of miss slice instructions, we leverage an existing technique to exploit temporal redundancy within a single core by dual-issuing slice instructions. Finally, this chapter described a technique to dynamically recover from (rather than simply detect) soft errors.

Chapter 6

Experimental Evaluation of Skewed Redundancy

This chapter presents a detailed performance evaluation of our implementation of skewed redundancy. It begins with a comparison to conventional redundant multithreading and shows that speedups, rather than slowdowns, are possible over unreplicated execution. It then discusses the performance impact of varying the capacity and latency of the shared structures in our design. We provide some comparison data to two other techniques with the similar goal of exploiting memory-level parallelism in programs: conventional (but unimplementable) large instruction windows and runahead execution. We then consider three scenarios: a baseline system composed of simple in-order cores, the impact of discarding L3 cache hits in addition to L3 misses, and additional performance gains from adding a third core. Throughput this chapter we also present characterization data to provide additional insight into our performance results.

Details regarding the underlying machine model used in this evaluation can be found in Chapter 3.2. We selected moderately-aggressive out-of-order cores as the building blocks of our design. Such cores are commonly used in current (and likely future) systems, because they provide a reasonable performance-complexity trade-off. They are large enough to capture *instruction-level parallelism* (ILP) inherent in applications (for which skewed redundancy provides only marginal benefit), but are not nearly large enough to extract significant *memory-level parallelism (MLP)* (which skewed redundancy can effectively capture).

6.1 Speedup Over Chip-level Redundant Multithreading

The original proposal for chip-level redundant threading (CRT) suffered considerable slowdown (around 15%) compared to unreplicated execution [69]. This is primarily due to its inability to commit unverified stores. Every store executed by each of the two redundant threads must wait until the other thread also reaches the same dynamic store and communicates its corresponding address and data. This severely limits the amount of "slip" possible between the two cores and translates a stall in either core into system stall. Until stores are verified at the commit pipeline stage, no other instructions are able to commit and the instruction window can fill up and block dispatch. Later proposals mitigated this overhead by starting the store verification process as soon as the store address and data are ready (rather than when the store becomes the oldest in the window); however, even in the best case, the performance of redundant execution never exceeds that of the baseline unreplicated case [95][26].

Our design allows stores executed by a leading core to locally commit before their correctness is verified. Stores are buffered in the global store queue (GSQ) until they are checked and become the oldest in-flight store in the system, at which point they are released to the shared L2 in program order. This decouples store verification from store commit (thereby avoiding the primary source of overhead in previous schemes), while still protecting the memory system from erroneous execution (because the processor cores can be squashed and their L1s flushed). We experience a slight slowdown that is never worse than 2% compared to unreplicated execution due to the addition of a pipeline stage for store insertion into the GSQ and contention for shared L2 bandwidth, but in most cases this is negligible (data not shown). We use this design as our baseline comparison point

(redundant execution with a GSQ but without discarding cache misses). Comparing our design to the original CRT proposal would yield significantly higher speedups.

Figure 6-1 presents three bars for each of the SPEC CPU2000 integer and floating-point benchmarks. The left-most bar represents the baseline CRT performance as described in the preceding paragraph. We reiterate that this bar does not correspond to the original CRT proposal (which suffers significant slowdown), but rather to our implementation that buffers and commits unverified stores. The performance of this configuration is nearly the same as unreplicated execution, and we refer to them interchangeably in most cases. This bar has been normalized to 1.0; the remaining two data points indicate relative improvement over this case. Absolute baseline IPCs are presented in Table 3-2.

The middle bar in the figures presents speedup of 2-core skewed redundancy over our baseline CRT implementation. It utilizes the same base machine model as CRT, except that it allows cores to become “skewed” with respect to each other by discarding L3 cache misses. The GROB and GSQ are sized according to the parameters presented in Table 3-1. We observe significant speedup in many of the benchmarks, indicating that their performance is severely constrained by the baseline 64-entry instruction window. Many of the benchmarks that exhibit little or no speedup are not sensitive to instruction window size at all, as the limit study in Chapter 6.4 will show. Integer benchmarks speedup by an average of 5%, but we point out that this number is heavily skewed by *mcf*. *mcf* traverses a linked list in a tight loop that misses the L3 on every pointer dereference. Consequently, it commits an average of only 1 instruction per 20 cycles, and benefits little from skewed redundancy (which does not generally improve pointer chasing programming constructs). This combination of exceedingly low IPC and marginal performance improvement (3%

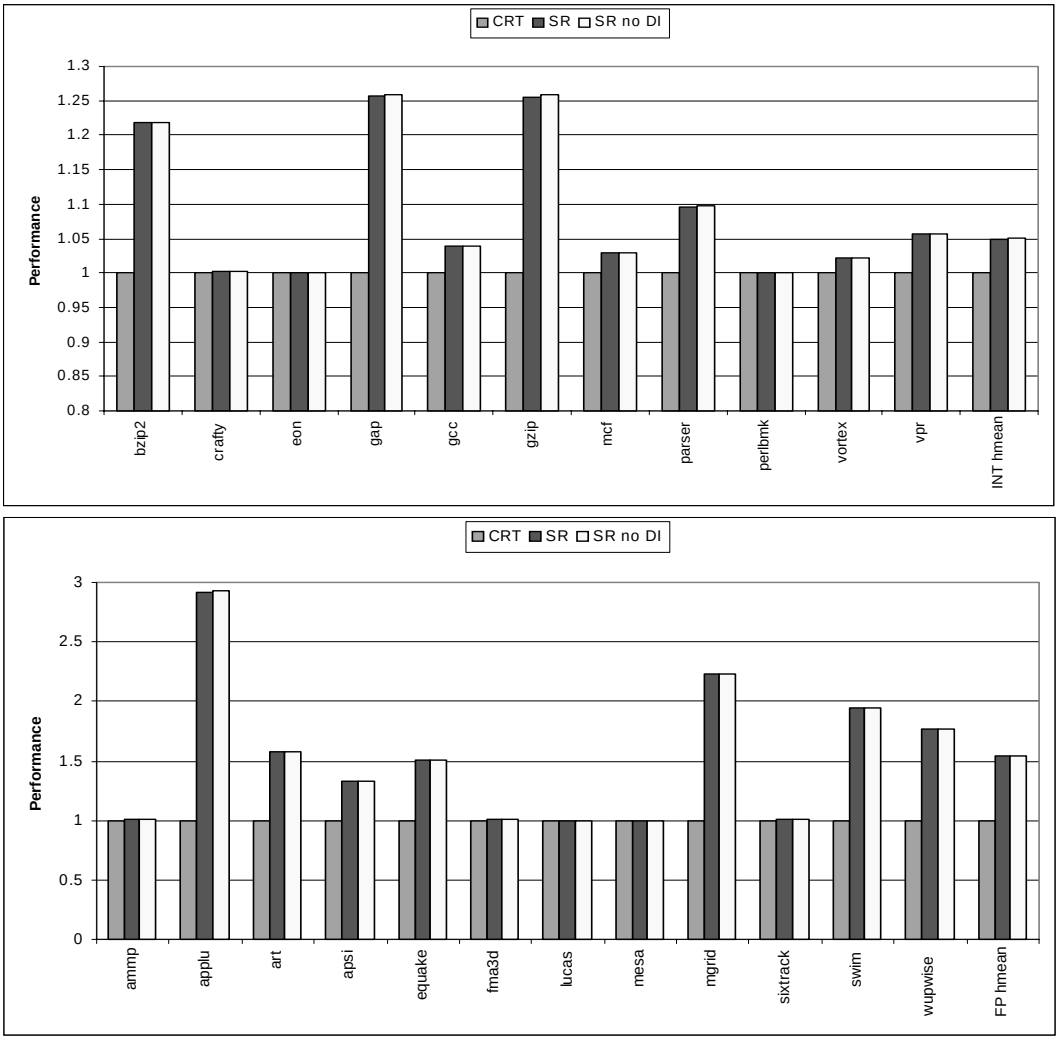

FIGURE 6-1. Skewed Redundancy Performance. From left to right: CRT, skewed redundancy, skewed redundancy without dual-issuing miss slice instructions. Note different y-axis scales.

speedup) leads to its significant contribution to the harmonic mean. If *mcf* is not considered, the average integer speedup doubles to 10%. Specific details regarding *mcf's* inner loop were presented in Chapter 4.4.3.

The floating-point benchmarks exhibit significantly higher speedups. Their large working sets lead to many independent cache misses (data presented in Table 4-3), and any techniques that help overlap concurrent misses can lead to large performance gains. Six of the benchmarks increase their performance by 50%, leading to an average speedup

of 54%. As in the case of several integer benchmarks, not all applications require large instruction windows for high performance; others are not limited as much by cache misses as they are by long-latency serialized floating-point operations (skewed redundancy helps neither of these cases). Finally, we point out that several of the floating-point benchmarks already realize significant speedups through the baseline hardware prefetcher described in Chapter 3.4. For example hardware prefetching doubles *mgrid's* initial IPC (Figure 3-1), before skewed redundancy doubles it again.

The third bar in the figures reports speedup without the overhead of dual-issuing miss-dependent instructions. This configuration can theoretically achieve higher performance by reducing the total number of executed instructions, but it also leaves a gap in error coverage because miss slice instructions do not redundantly execute. However unlike the case presented in Chapter 5.3 proposed by Ray et al. [67], where every instruction issues twice, only a minority of instructions depend on cache misses and need to be dual-issued (data presented in Chapter 5-1). Furthermore, any resulting slowdown only affects one core and allows the other(s) to continue. Consequently, using dual instruction issue to close the gap in error coverage caused by discarding cache misses comes at a very small price -- well under 1% overhead in all cases.

6.2 Sensitivity to Skewed Redundancy Resources

Chapter 4.3 proposed three global structures to facilitate inter-core communication: the GROB, the GARF, and the GSQ. It presented motivating data indicating that accesses to these structures are rare, implying that they can be small and slow. This section adds characterization data and discusses the effect that their size and latency has on overall

6.2.1 Global Reorder Buffer

Figure 6-2 shows the fraction of time that the *global reorder buffer* (GROB) has various numbers of entries. We highlight two important points in this data: First, the GROB is empty for the majority of the time in all benchmarks, suggesting potential power savings through power or clock gating during periods with no last-level cache misses. Second, it rarely contains more than a handful of entries -- less than 10 entries for over 98% of the time in every benchmark we studied. Referring back to Figure 4-6, we see that, not surprisingly, those benchmarks that write to the GROB more frequently (i.e. have a large fraction of miss-dependent instructions) also have higher GROB occupancies.

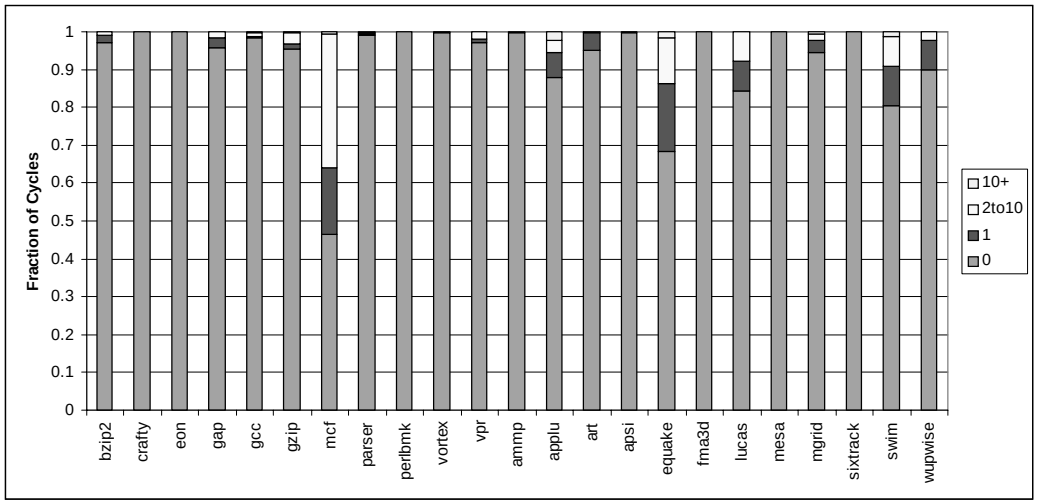

FIGURE 6-2. GROB Occupancy. Fraction of time that the GROB has various occupancies

Of course, since the capacity of the GROB only matters during long-latency memory accesses, the average occupancy does not tell the entire story in terms of overall performance. We therefore performed a series of experiments that varied the maximum GROB capacity from an unlimited number of entries down to a modest 32 entries.

Figure 6-3 presents the results of this experiment. It shows that a GROB with as few as 32 entries can obtain most of the performance of one of unlimited capacity (particularly if one only considers the average speedups reported by the last column). Clearly, applications that exhibit the highest degrees of “skewing” place the most pressure on the GROB, and their performance is consequently the most sensitive to its capacity. For those cases, larger GROBs can achieve higher speedups, however only marginal gains occur beyond 128 entries. Because a 128-entry FIFO RAM can be accessed quickly and consumes little power, we select it as our design point (Table 3-1 and Figure 6-1).

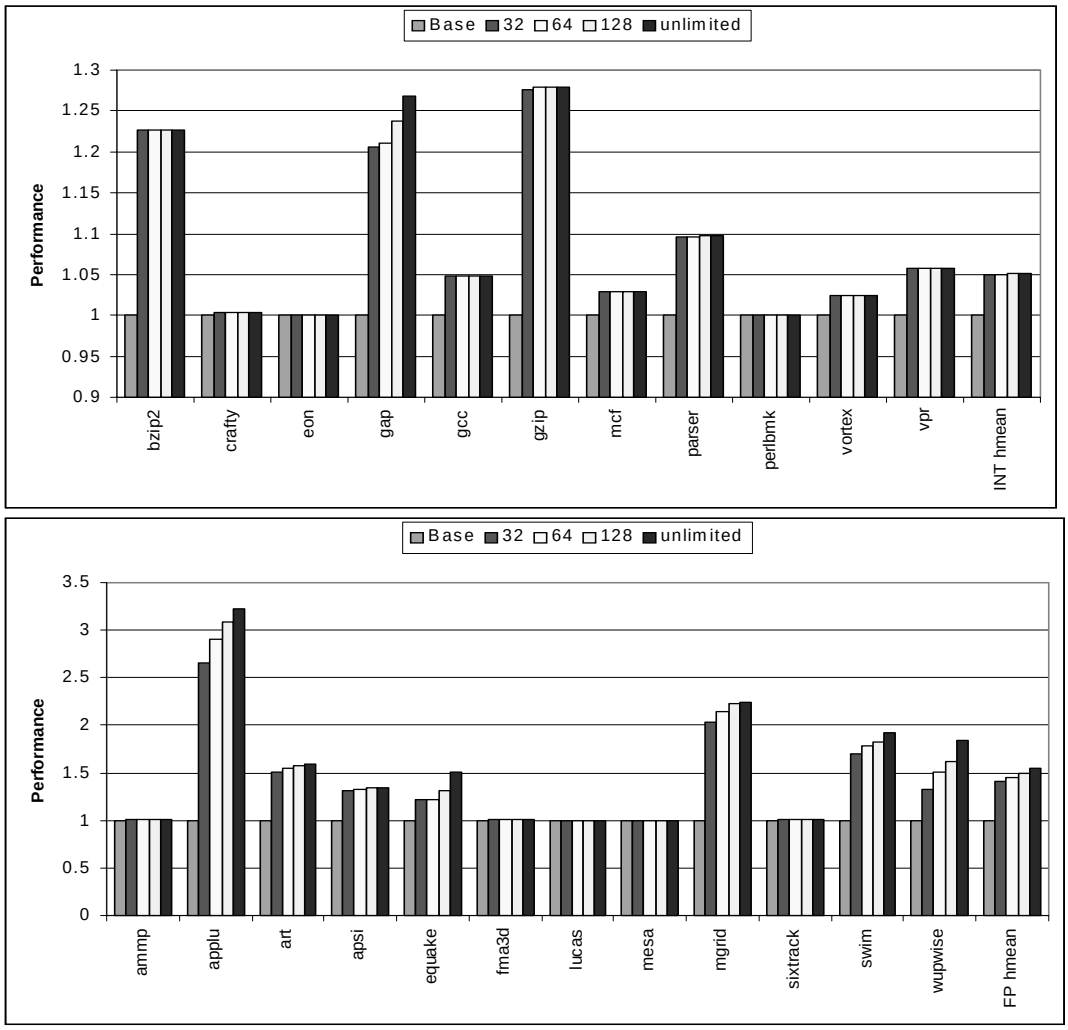

FIGURE 6-3. GROB Size Sensitivity. Performance speedups are normalized to replicated execution that does not discard cache misses (left-most bar). Note different y-axis scales

We also studied the effect of GROB read latency on performance by varying the time required for a core to retrieve a miss-dependent operand, from 0, to 10, to 20 cycles. We found very little variation -- less than 2% performance degradation when increasing the latency from 0 to 20 cycles in all cases (data not shown). This makes sense in light of the data presented in Figure 4-7, which showed that very few instructions require inter-core synchronization (less than 0.5% of all instructions, discounting *mcf*). Furthermore, by definition, the transferred values depend on long-latency cache misses and are likely to already have been stalled for some time. From the receiving core's point of view, the GROB read latency is already high, and forcing it to wait a relatively small amount of additional time does not impact performance.

6.2.2 Global Architected Register File

The *global architectural register file* (GARF) contains slice results older than the oldest in-flight instruction among all cores. Because it only needs to contain the last value written to each register, its size is bounded by the number of general purpose registers defined by the architecture (typically between 64 and 100). If the last write to a register does not depend on a miss, the GARF does not need to retain that value at all because it is globally computed and available in each core.

Figure 6-4 graphs the number of active entries in the GARF over time, and was collected using the same methodology used for Figure 6-2. Both the integer and floating-point benchmarks exhibit low occupancy, containing less than 10 entries over 99% of the time. Within this 99%, however, we observe clearly distinctive behavior between the two suites. While most of the integer benchmarks contain no entries in the GARF the vast majority of the time, the floating-point applications often have occupancies between zero

and ten. We hypothesize that several program attributes could lead to this.

- **Higher miss rates:** The floating-point applications we studied have higher miss rates than the integer applications, leading to more miss slice dependence chains.
- **Higher register file utilization:** Most integer applications do not make extensive use of floating-point registers. Since these registers are rarely used, they are unlikely to depend on misses.
- **Longer variable lifetimes:** Many floating-point applications perform multiple mathematical operations on the same value. If that value depends on a miss, its corresponding invalid bit may propagate to many additional registers. For example, if a vector of length n is multiplied by a n x m matrix, the i^{th} element in the vector participates in m*(n - i) operations. Integer code, on the other hand, is more likely to use variables for short-term calculations, such as computing control flow conditions that feed nearby branches.

Although implementing the GARF as an additional architected register file is not likely to be prohibitive in terms of complexity or power, this data suggests that smaller alternative configurations are possible (for example, through associative indexing or by mapping the logical register namespace onto a small pool of physical locations).

Finally, we reiterate that results are read from the GARF only after they have been moved from the GROB (similar to the relationship between a conventional ROB and ARF); therefore, latencies to access each structure can be overlapped. Consequently, we do not explicitly model any additional penalty above and beyond the GROB access time when a miss-slice value is required to be communicated between cores.

6.2.3 Global Store Queue

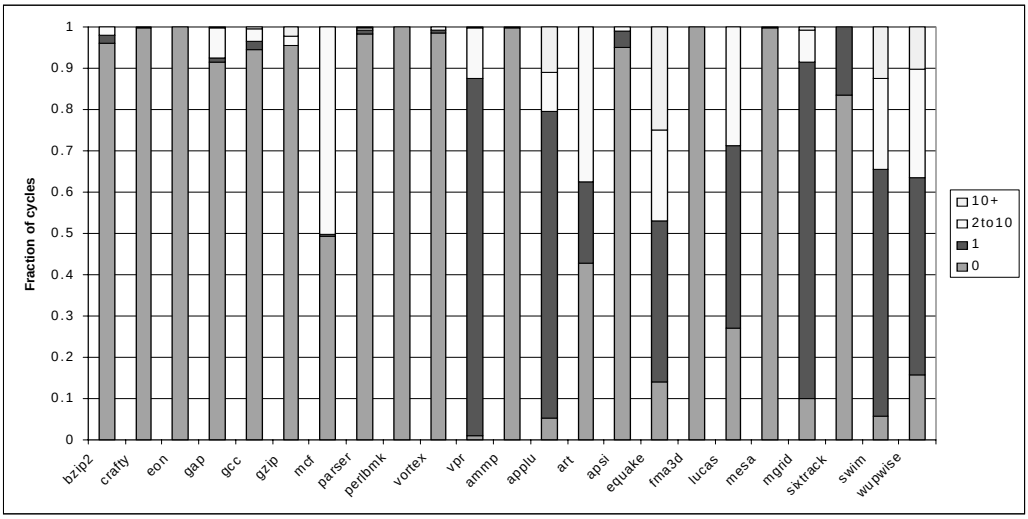

FIGURE 6-4. GARF Occupancy. Fraction of time that the GARF has various occupancies

Unlike the GROB and GARF, which only need to buffer relatively rare miss-dependent instructions, the GSQ needs to contain all stores in the virtual instruction window, which this chapter will show can span thousands of instructions. This buffering is required for several reasons:

- *Error detection:* since all errors eventually propagate to stores, comparing store data and addresses computed across cores is sufficient to detect errors.
- *Store re-ordering:* stores must be released to the shared L2 cache in program order, even though they may be committed by individual cores in a different order.
- *Memory disambiguation:* loads must be able to identify the most recent store to their address.

This last requirement (memory disambiguation) requires that loads can search the GSQ by address. Such content-addressable indexing is notoriously slow, and can potentially delay loads and their dependent operations. However, several properties of the GSQ mitigate this penalty. First, the contents of the L1 data cache are always up-to-date. When a core commits a store with invalid data, it invalidates any matching lines in its private L1.

This prevents the core from using stale data and forces it to access the GSQ or L2. Therefore, GSQ searches are only required on L1 cache misses.

Sethumadhavan et al. showed that a large fraction of associative store queue searches can be eliminated through the addition of a small Bloom filter [76]. Bloom filters [10] can be efficiently implemented with a hashing data structure and conservatively indicate if an address is in the backing store queue. Misses in the Bloom filter guarantee a miss in the store queue as well. With this enhancement, loads only need to search the GSQ when they miss the local L1 and hit in the Bloom filter. As a simple example, if we hypothetically assume a 90% L1 hit rate and that 90% of L1 misses are filtered by the Bloom filter (Sethumadhavan et al. showed up to 98% of associative searches can be filtered, and many applications have a much higher than 90% L1 hit rate), only 1% of all dynamic loads are required to search the GSQ.

In the infrequent case where a load address misses the L1 data cache and hits in the Bloom filter, the GSQ access time can be overlapped with the L2 latency. If the address is not found in the GSQ, the L2 access can proceed as usual; a GSQ hit indicates that the core that issued the load must wait until the store updates the shared L2. We assume that the GSQ can be searched within the L2 latency (15 cycles in our model) and therefore does not introduce additional delay.

If the GSQ fills, the leading core cannot commit additional store instructions and will stall. To study the effect of GSQ capacity on overall performance, we performed a serious of simulations that varied its maximum occupancy from 64, to 128, to 256, to an optimistic unlimited case. Figure 6-5 presents the outcome of these experiments. Performance with a 256-entry GSQ is almost identical to that of the unbounded case, indicating

that there are rarely more than 256 stores between the oldest and youngest instructions executing in any core. Limiting the GSQ to only 64 entries obtains most of the speedup of the unbounded case, although noticeably degrades performance for some benchmarks. Given the assumptions regarding GSQ latency and access frequency presented earlier in the section, 128 entries seems to be a reasonable design point and provides almost all of the performance benefit of higher capacities. The data presented in Chapter 6.1 and later in this chapter assume 128 entries.

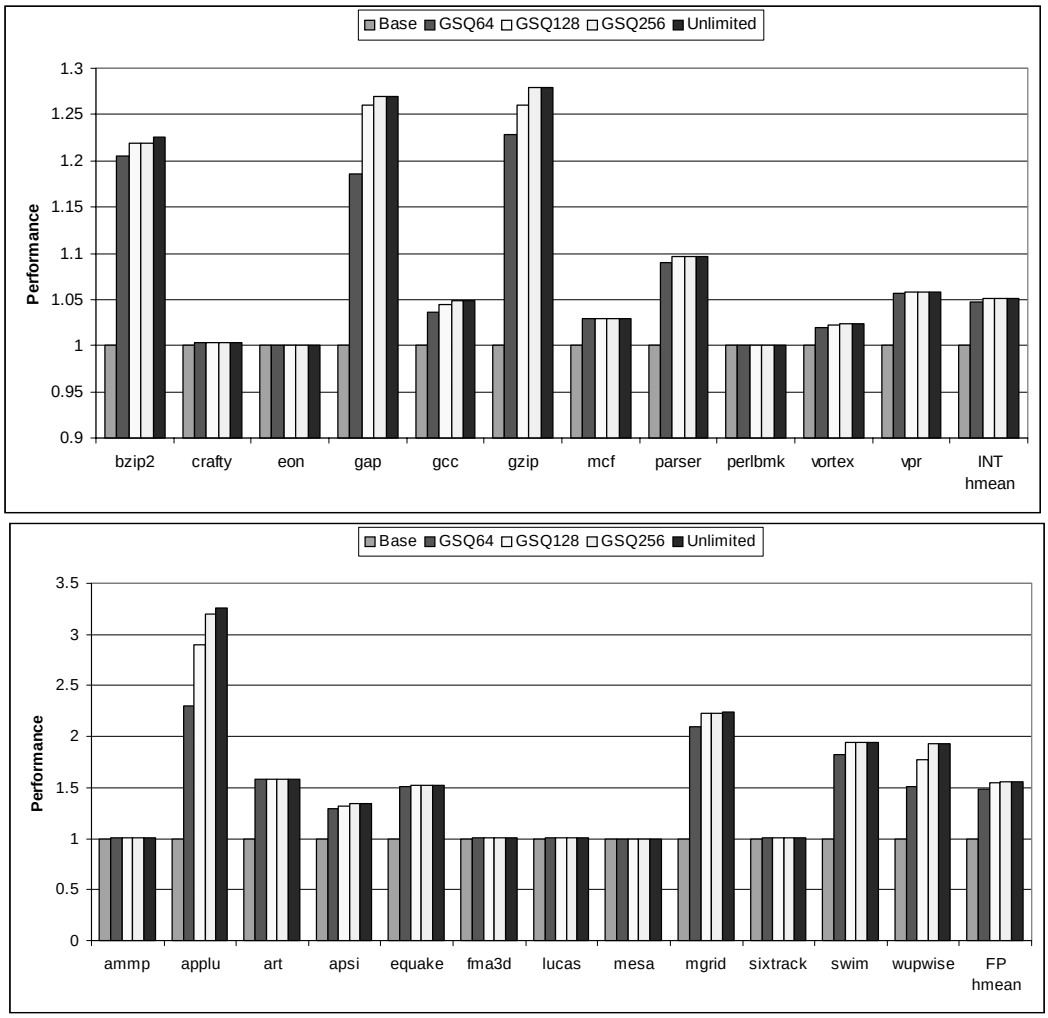

FIGURE 6-5. GSQ Size Sensitivity. Performance speedups are normalized to replicated execution that does not discard cache misses (left-most bar). Note different y-axis scales.

6.3 Virtual Instruction Window Size

This thesis has promoted skewed redundancy as a technique to realize a large "virtual" instruction window that provides the same ability to overlap cache misses as a conventional instruction window, but without the high implementation costs. We refer to an instruction as being within the virtual window if its program order falls between the oldest and youngest in-flight instruction in any core. Independent cache misses within this range will be prefetched by the leading core before they are re-executed by the trailing core.

Figure 6-6 graphs the average size of the virtual window for each benchmark. It indicates how successful discarding cache misses is towards "skewing" each core's execution with respect to the other, and provides a rough estimate of the instruction buffering resources required by a conventional microarchitecture to obtain comparable performance. The degree of skewing depends primarily on the frequency and penalty of L3 cache misses and the frequency of inter-core synchronizations (Chapter 4.4). Clearly, applications with many misses and few synchronizations offer the most potential speedups. In order to clearly present a wide range of window occupancies, the y-axis uses a logarithmic scale.

Even though each of the two cores can contain only 64 in-flight instructions each, we observe that the *average* virtual window size approaches 1,000 instructions for several benchmarks. Not surprisingly, benchmarks that exhibit large windows in this figure synchronize infrequently (Figure 4-7) and have many L3 misses (Table 4-3), yielding significant speedups (Figure 6-1). Note that because the average virtual window size is certainly deflated by periods of execution with few or no misses, the *peak* separation of the leading and trailing core can be significantly higher -- up to thousands of instructions in some

In addition to the total number of instructions spanning the two cores, Figure 6-6 also displays the subset of those instructions which are stores. This number is important because it indicates the average number of entries the GSQ needs to contain to avoid stalls (GSQ capacity was not limited while collecting this data). Those benchmarks with a large number of in-flight stores are consequently more sensitive to the GSQ capacity (e.g. *gap, applu, wupwise). mcf* and *lucas* appear to be missing data points; their combination of small virtual window size and relative infrequency of store instructions lead to very few in-flight stores at any time.

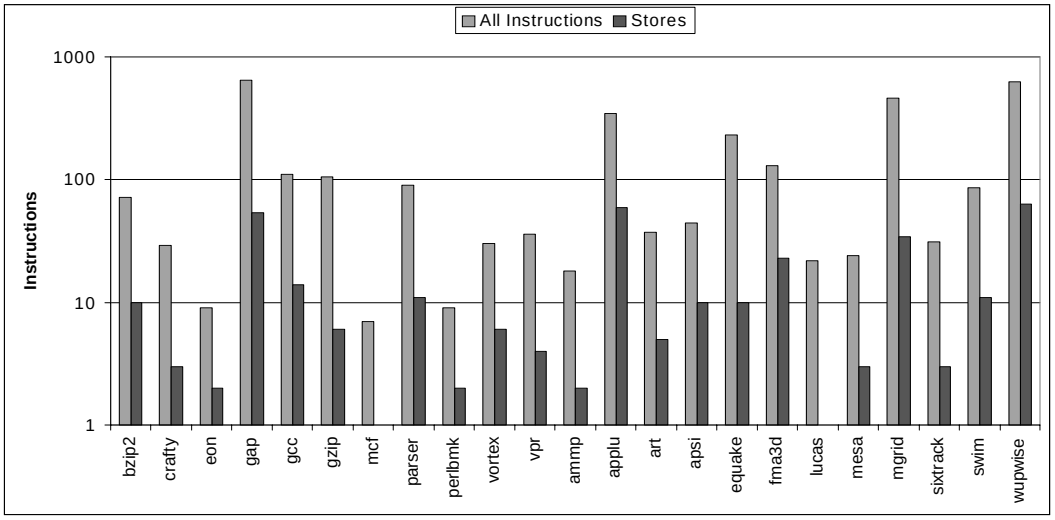

FIGURE 6-6. Virtual Window Size. Number of instructions and number of stores between the oldest and youngest in-flight instructions. Note log scale

6.4 Limit Study: Large Conventional Out-of-Order Windows

Because skewed redundancy attempts to obtain the benefit of a monolithic instruction window, it makes sense to compare it to a conventional design scaled to handle many in-flight instructions. Even though such designs are difficult to implement without impacting the clock period, they provide a pseudo-upper bound on the performance improvement

made possible through increased instruction window capacity (whether physical or virtual).

Figure 6-7 reports the results of a limit study we performed that successively doubles the reorder buffer (ROB) capacity from 64 up to 1024 instructions. All other microarchitectural parameters are sized according to Table 3-1 except for the LSQ (which is set to 1/2 of the ROB size), and issue window (which is set to 1/4 of the ROB size).

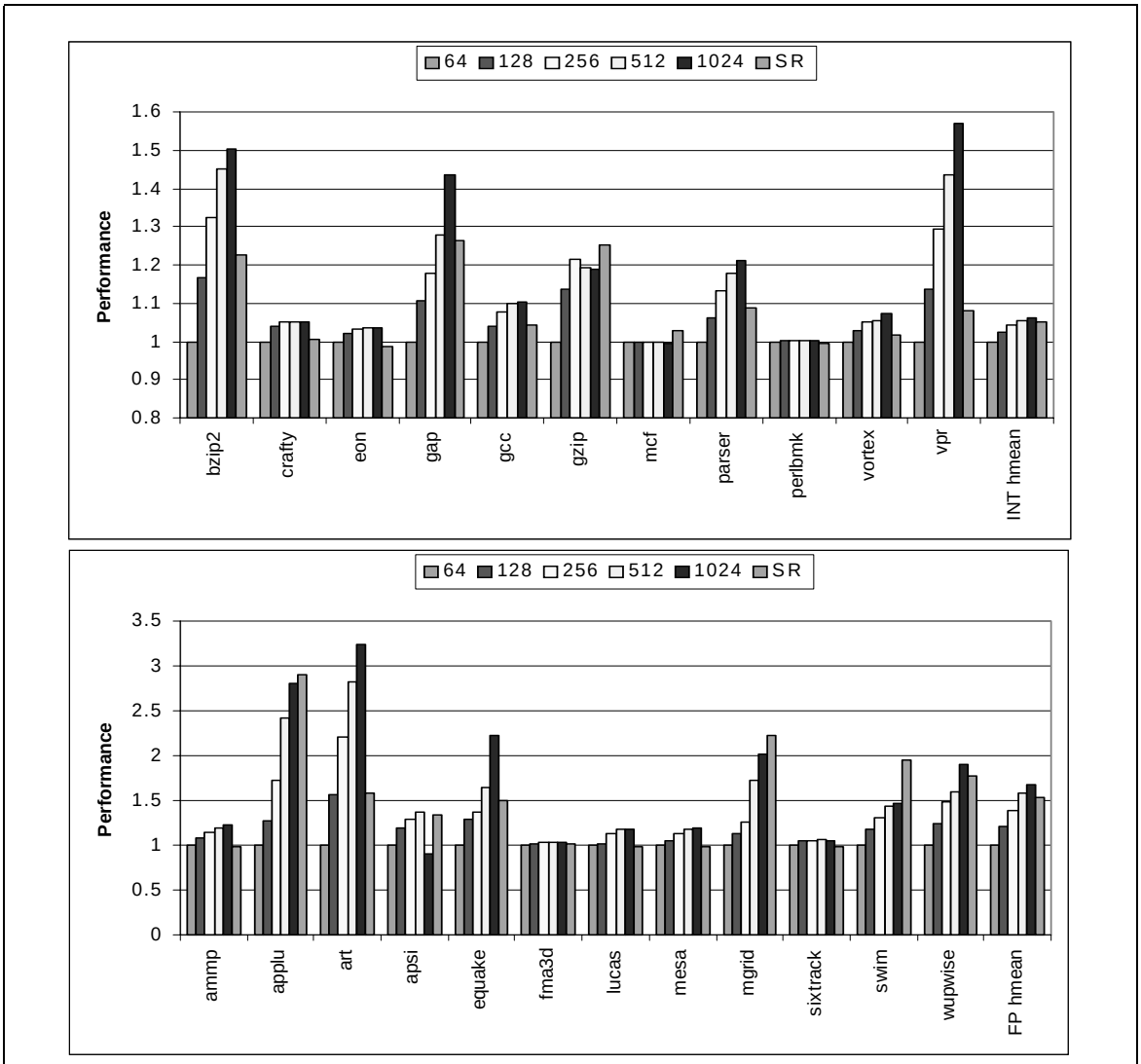

FIGURE 6-7. Large Window Limit Study. Performance normalized to a baseline machine with a 64-entry ROB, 32-entry LSQ, and 16-entry issue window. Bars 2-5 successively doubles each of these resources. The final bar indicates skewed redundancy speedup. Note different y-axis scales

Although the average integer skewed redundancy performance matches that of the

256-entry ROB, it is noticeably worse than a conventional large window design for several of the benchmarks. We attribute this performance difference to several factors:

- *Synchronization:* integer benchmarks are more likely to synchronize register values through the GROB/GARF.
- *Instruction-level parallelism:* a true large-window design can exploit ILP because instructions executed while waiting for a cache miss do not need to re-execute after the miss completes. In skewed redundancy, miss-independent instructions are always executed by all cores. This introduces additional latency to re-execute miss-independent instruction when the miss returns.
- *Redundant execution overhead:* Even though the overhead from redundant execution is small, it can cause a slight slowdown in some cases. For example. *eon* obtains no speedup from skewed redundancy and also has the highest redundant execution overhead (about 2%). This results in a slight slowdown compared to the single-core 64-entry ROB.

Floating-point performance more closely approaches that of the conventional design. A 1024-entry ROB achieves an average 67% speedup over a 64-entry ROB. Two 64-entry skewed redundant cores achieve 80% of this, or a 54% average speedup. The performance of the 256-entry ROB (which is still larger than any out-of-order window built to date) is approximately equal to our design (58% vs 54% speedups, respectively).

We note that skewed redundancy exceeds the performance of a 1024-entry ROB in several cases (*gzip, applu, mgrid, swim*). This indicates that greater speedups are possible through even larger conventional windows. We do not include data for ROBs beyond 1024 entries due to limitations in simulation speed -- the same properties that make large win-

dows difficult to implement in hardware also significantly increase simulation time (e.g. instruction wakeup, resolving memory dependences, etc.). Finally, we point out one anomalous data point exhibited by *apsi*, which suffers a significant performance drop when the ROB size is increased to 1024 instructions. We attribute this to non-optimal instruction scheduling decisions when the scheduler is presented with a very large number of eligible instructions.

6.5 Compared to Runahead Execution

We also compared skewed redundancy to runahead execution [22][60], since runahead's ability to discard expensive cache misses and continue executing is the fundamental technique we use to improve performance. Figure 6-8 presents three bars for each benchmarks. All data is normalized the left-most bar, which represents performance by the baseline, 64-entry ROB configuration. The second bar indicates the relative improvement of runahead execution measured on a simulator we modified to implement runahead execution as described by [60]. Because all state updates in runahead mode must be nullified when the cache miss returns and the runahead thread is squashed, stores do not update the caches or memory. Any younger loads that access that address will therefore access stale data if the store has already committed. The third bar overcomes this limitation by adding a runahead cache that is exclusively used to buffer stores executed during the runahead period. When the initial miss returns, the entire contents of the cache are cleared. Machine model parameters are set according to Table 3-1.

In general, benchmarks that show improvement through one technique also improve by the other, however we observe that skewed redundancy achieves better

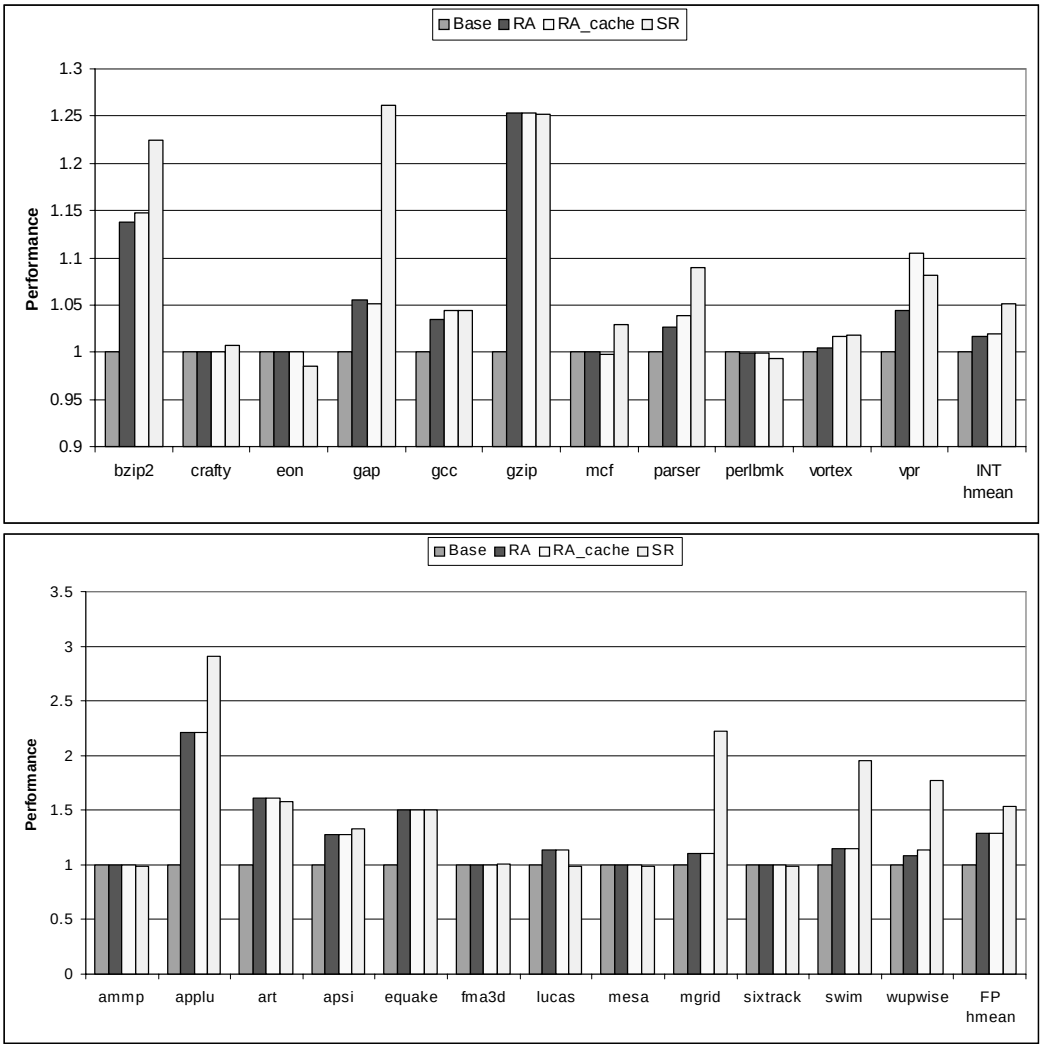

FIGURE 6-8. Runahead Execution. All data is normalized to the performance of a single 64-entry ROB. RA: runahead execution with without a runahead cache (i.e. stores are converted to no-ops); RA_cache: runahead execution with a dedicated runahead cache; SR: 2-core skewed redundancy. Note different y-axis scales.

speedup than runahead execution in almost all cases. This highlights runahead execution's well-known shortcoming of limited reach into only those instructions that fall within instructions within the miss shadow. The runahead thread can only advance until the initial miss returns, and must restart at the same point as the main thread when the next miss is encountered. Skewed redundancy, on the other hand, does not restart when a last-level cache miss returns, and cores maintain any lead garnered by discarding misses. Because stores executed by redundant cores update their L1 data caches, there is no need for a sep-

arate runahead cache, which can achieve moderate speedups over baseline runahead execution in some cases (e.g. *vp*r). A few of the benchmarks achieve nominally better performance with runahead execution than with skewed redundancy. This can occur when a significant fraction of instructions synchronize between skewed redundant cores (Chapter 4.4.3). In this case a leading core may need to wait for a miss slice result to be produced by a trailing core, while runahead execution will continue executing instructions without stalling.

Finally, we point out that, like the high-performance conventional large-window design discussed in the previous section, runahead execution does not provide any form of error detection or tolerance, which is one of the principal goals of this work.

6.6 In-order Cores

The baseline microarchitecture used throughout this thesis assumes moderately-aggressive processor cores that utilize 64-entry ROBs. Such designs provide a good trade-off between complexity and performance -- they are generally capable of extracting ILP from programs, but do not contain nearly enough instruction buffering capacity to exploit significant degrees of MLP.

This section considers an alternate design point that implements a CMP with many simple in-order cores. Designs of this nature typically target workloads with abundant *thread-level parallelism* (TLP), where total system throughput is more important than the performance of any single thread. Constrained chip-level power budgets and the desire for more cores have lead to the recent introduction of such throughput-oriented systems by several microprocessor vendors (e.g. Sun Ultrasparc T1/T2, IBM Power6].

Skewed redundancy is a natural fit for emerging many-core CMPs. The availability of many cores lowers the opportunity cost to allocate some of them redundantly. Furthermore, not all applications are easily parallelizable and single-thread performance is still often important. Skewed redundancy applies a limited form of out-of-order execution to in-order designs to reclaim some of the performance that simple cores cannot achieve on their own.

Figure 6-9 indicates the speedup skewed redundancy achieves over a baseline in-order and out-of-order design. Each bar is normalized to its respective microarchitecture -- "io" therefore represents the speedup two skewed in-order cores achieves over two non-skewed redundant in-order cores, and "oo" represents the speedup two skewed out-of-order cores achieves over two non-skewed redundant out-of-order cores. Because the performance of the non-skewed redundant cores approximately equals that of non-redundant execution (Chapter 6.1), this graph can equivalently be interpreted as speedup over a single-core.

We see that skewed redundancy can provide significant speedups in in-order designs, making it an effective technique in the growing number of such systems. In some cases, its speedup exceeds that of the out-of-order case, since unlike dynamic instruction issue, in-order issue provides very little opportunity to overlap cache misses on its own. However at other times, we observe greater out-of-order speedups. This occurs in benchmarks with significant ILP: when expensive cache misses are eliminated through prefetching, the benefit of issuing instructions out-of-order becomes more pronounced (due to Amdahl's Law). The performance of in-order cores, on the other hand, becomes throttled by in-order issue, even when freed of miss-related stalls. This effect also appears in the

hardware prefetching evaluation in out-of-order and in-order cores that appeared in Chap- 137
ter 3.4.

Finally, by multiplying the speedups in Figure 6-9 with the baseline IPCs in Table 3-2, we point out that two skewed in-order cores can outperform a single baseline out-of-order core in some cases (e.g. *applu*, by 42%, and *swim*, by 51%).

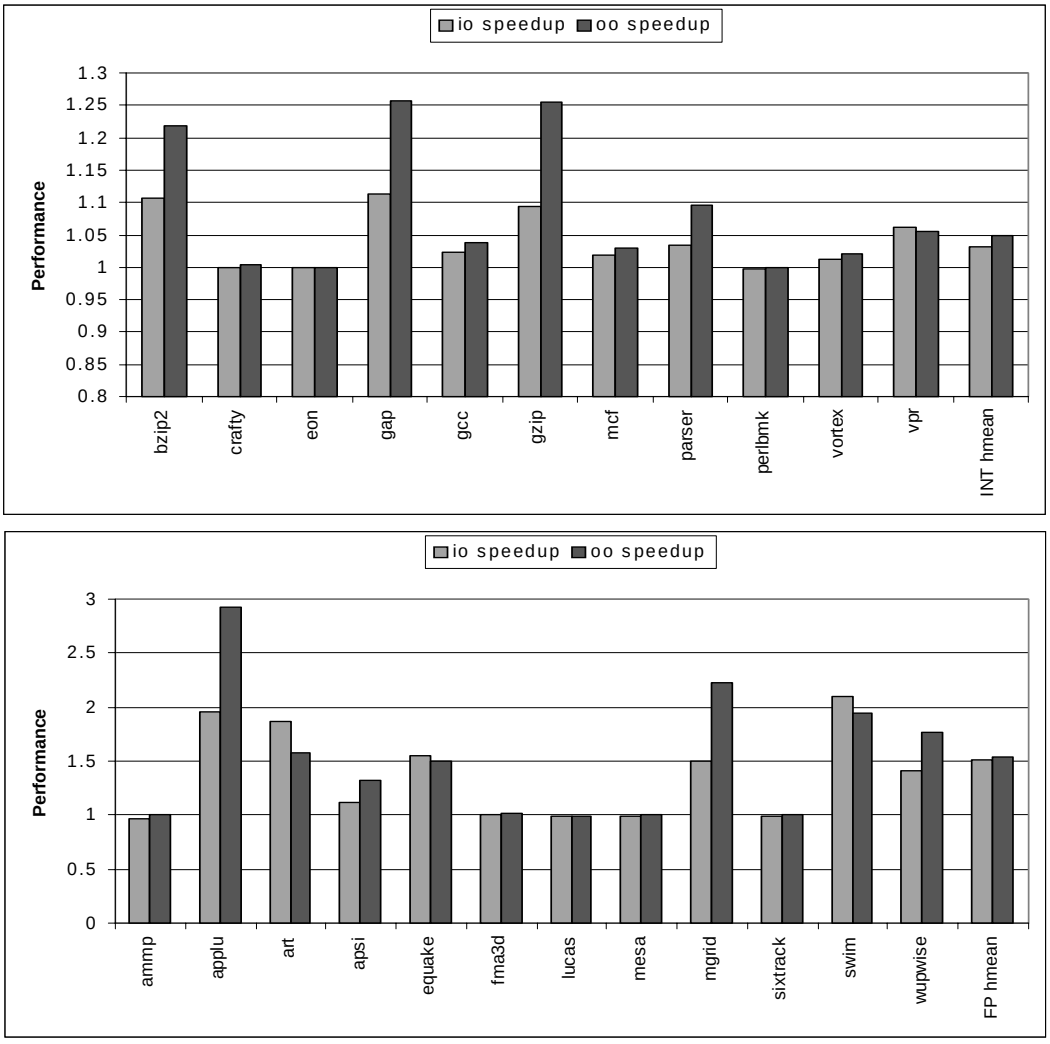

FIGURE 6-9. In-order Core Performance. "io speedup" indicates in-order skewed redundancy over an in-order baseline. "oo speedup" indicates out-of-order skewed redundancy over an out-of-order baseline. Note different y-axis scales

6.7 Skipping L3 Hits

Cache misses that access main memory are among the most expensive operations performed by a program, and are the primary target of skewed redundancy. Since our model uses three levels of cache, processor cores only discard L3 misses. This section considers the performance implications if L3 hits, in addition to misses, are partitioned among cores. Increasing clock frequencies, additional levels of caching, and higher degrees of sharing (and therefore arbitration) all push up the number of cycles required to access last-level caches, making skipping them more attractive. Our model assumes a 50-cycle latency to access the shared L3.

Figure 6-10 presents the results of this experiment for both out-of-order and in-order cores. It indicates the relative performance added by discarding L3 hits and misses over a configuration that only discards misses. We see that discarding L3 hits helps performance in some cases (i.e. performance greater than 1.0) and hurts performance in others (i.e. performance less than 1.0). A 50-cycle penalty on L3 hits is sufficiently costly that alleviating cores from needing waiting for them to complete can achieve better performance. However it also has the negative attribute of poisoning more registers, which leads to additional synchronization between cores. In many cases, any advantage from skipping L3 hits is offset by this additional communication overhead.

We expect skipping L3 hits to help the in-order design more than the out-of-order design, since even a 64-entry out-of-order window is at least partially effective at tolerating a 50-cycle completion latency. The figures show that this is, in fact, the case -- the in-order bars are higher than the out-of-order bars for almost every benchmark. We highlight one benchmark (*art*) that more than doubles the performance obtained by skipping only

L3 misses (which, in turn, doubles the performance of the baseline configuration). This is a result of the fact that *art* has a much higher number of L3 hits than any of the other benchmarks -- about one out of every five instructions is a load that misses the L2 cache and hits in the L3.

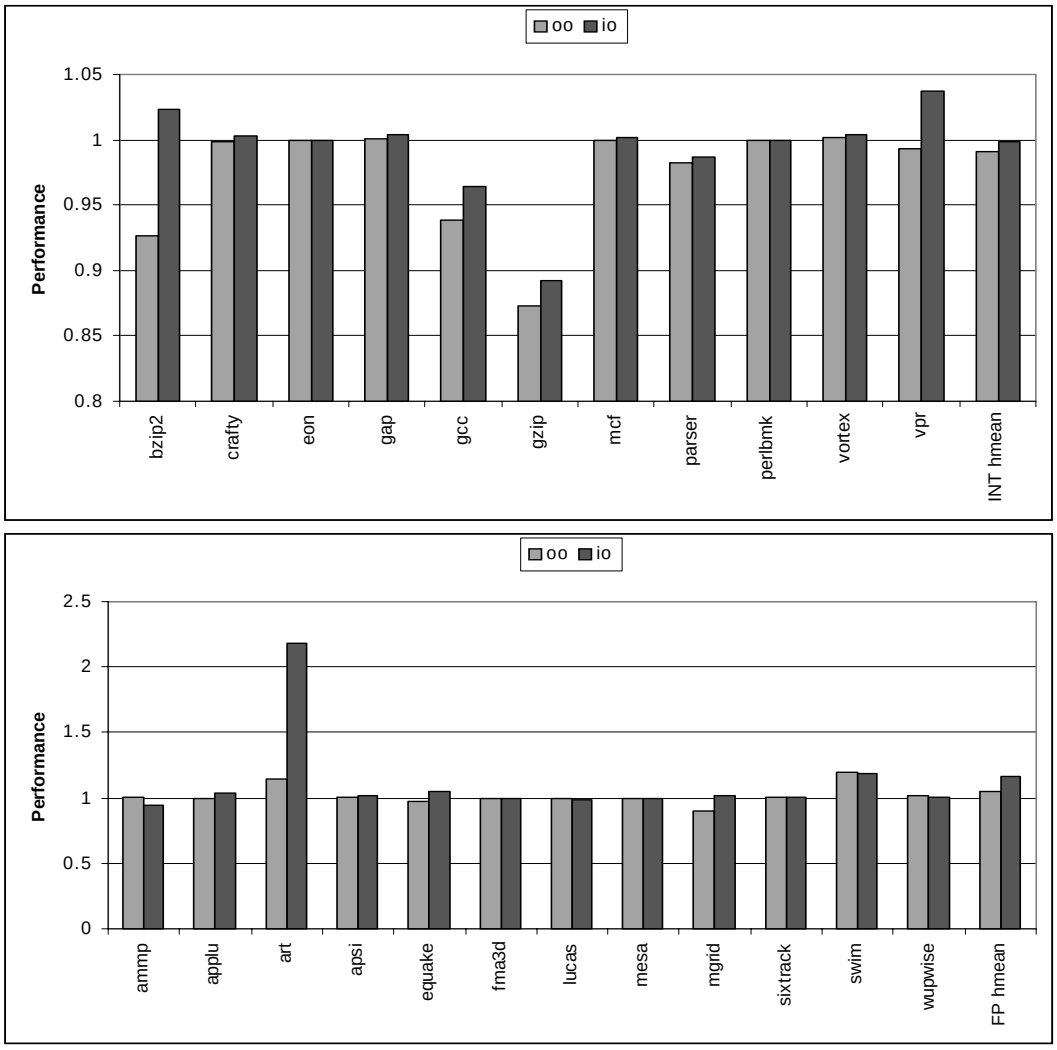

FIGURE 6-10. Impact of Skipping L3 Hits. Each bar indicates the relative performance of discarding L3 hits and misses over discarding only L3 misses for out-of-order and in-order core microarchitectures. Note different y-axis scales.

6.8 Three cores

Chapter 4.9 described how adding a third core can potentially improve perfor-

mance, and Chapter 5.4 showed that it can provide error correction in addition to detection. This section quantifies the performance benefit yielded by a third core.

Figure 6-11 graphs three data points. The first two bars indicate the performance improvement of two skewed redundant cores over the non-skewed configuration; this data has appeared in several other charts in this chapter and is included for reference. The third bar shows the additional speedup obtained by adding a third core. Generally, a third core further improves performance with applications that have many cache misses with addresses that depend on prior misses. With two cores, the later miss must wait until the trailing core reaches it before it can issue. If a third core is assigned to wait for the miss that the address depends on, the second miss can issue early. Additional details regarding this benefit were discussed in Chapter 4.9.

For the most part, incremental speedups over a two-core system are minimal. This makes sense, as cache misses are generally followed by more independent misses than dependent misses. Furthermore, the independent misses can issue as soon as they are uncovered by a leading core, while the dependent misses must at least wait for the initial miss to complete. A third core, therefore, helps a small number of misses issue slightly earlier than they would otherwise. Although allocating a third redundant core for such marginal performance improvements may not be a compelling use of chip resources, we point out that it also provides error recovery in addition to error detection. Finally, the policy of assigning misses to the third core is based on a heuristic presented in Chapter 4.9. Additional tuning of this heuristic could potentially lead to greater speedups.

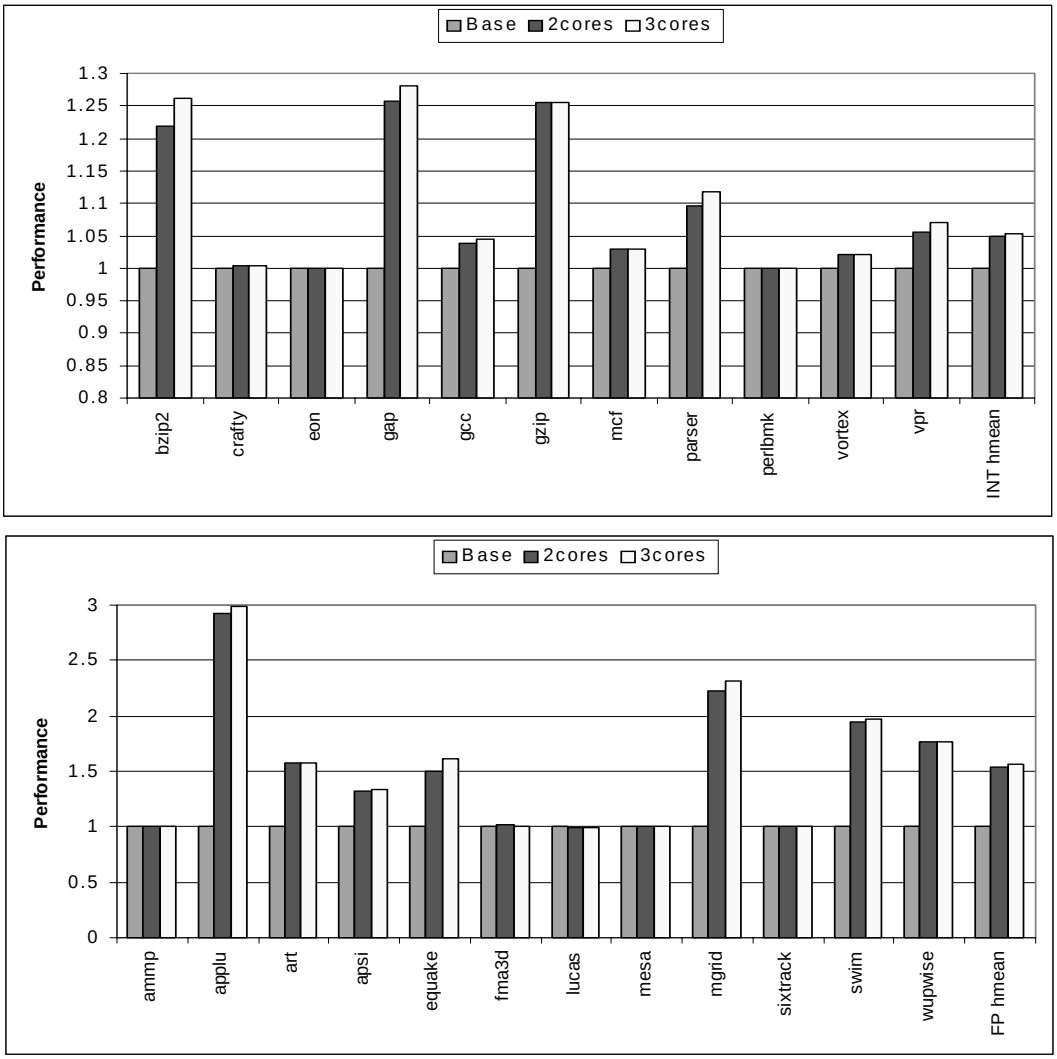

FIGURE 6-11. Performance Improvement from Adding a Third Redundant Core. Note different y-axis scales.

6.9 Chapter Summary

This chapter presented an experimental evaluation of skewed redundancy. It started by presenting average speedups of 5% for the SPEC CPU2000 integer benchmarks (or 10% speedup discounting *mcf*) and 50% for to floating-point benchmarks on a realistic implementation of our design over non-redundant execution. It continues by examining the additional performance gains possible through larger and more aggressive resource

configurations. We then compare our technique to two other techniques that also exploit memory-level parallelism to achieve high performance (but not error tolerance): runahead execution and conventional large instruction windows. Finally, we explore three alternative configurations of skewed redundancy: discarding L3 hits in addition to L3 misses, a chip multiprocessor composed of in-order processor cores, and finally a system that can additionally correct errors by allocating a third cores to redundantly execute a program.

Chapter 7

Conclusion

Despite sustained exponential growth in transistor density characterized by Moore's Law, it is increasingly difficult to improve the performance of single-thread programs. At the same time, shrinking feature sizes and reduced supply voltages have propelled soft error mitigation techniques to the forefront of research and development agendas. Prior work has proposed allocating redundant processor cores on emerging chip multiprocessors to execute a program thread to achieve either higher performance or error detection. However to date, these two goals have been mutually exclusive and often at each other's expense.

This thesis presented *skewed redundancy,* an approach to simultaneously achieve both high single-thread performance and soft error tolerance through redundant execution on CMPs. The first part of this thesis described the correctness requirements to enable two threads to each simultaneously execute a subset of the overall program. Our detailed performance evaluation indicates that this results in average SPEC CPU2000 integer and floating-point speedups of 5% and 50%, respectively, and speedups of over 200% in some cases. The second part described relatively minor modifications to our design to add error detection (two cores) and error recovery (three cores) as well.

7.1 High Performance

Skewed redundancy executes the same program thread on multiple CMP cores. To enable better performance than each core could achieve on its own, expensive last-level

cache misses only cause a single core to stall. The other core(s) continue to execute miss-independent instructions, realizing a large *virtual instruction window* that overlaps subsequent cache misses. A primary contribution of this thesis is that it describes in detail many of the design and implementation issues to allow a processor core to non-speculatively execute a program with incomplete architected state. We highlight the scenarios when execution cannot continue -- namely, miss slice dataflow joins, miss-dependent control flow, miss-dependent store addresses, and precise exceptions. We then propose three global structures to facilitate the passing of miss slice results between cores: the *global reorder buffer* (or GROB), the *global architected register file* (or GARF), and the *global store queue* (or GSQ). Our experimental evaluation shows that these structures can be small, simple, and are infrequently accessed.

7.2 Error Detection and Recovery

Redundant execution is the cornerstone of soft error mitigation techniques. Since soft errors are by definition transient in nature, they are unlikely to affect multiple instances of an instruction replicated in time and/or space. Therefore, comparing the results of redundant computations can detect errors.

To maintain binary compatibility with existing software, a system must maintain the appearance that a single instance of a program thread is running. Since a processor's actions are generally only visible to the rest of a system through the store instructions it executes, most forms of redundant execution require coalescing multiple instances of redundant stores before they are released to shared levels of the memory hierarchy.

Similarly, because all non-dead computation eventually propagates to a store

instruction, only checking the data and addresses of redundant store instructions is sufficient to detect errors. Since we already use a *global store queue* to combine redundant stores before releasing them to the memory hierarchy, extending this structure to incorporate store comparisons is relatively straightforward.

In order to deal with the error coverage gap in last-level cache misses and their dependent instructions, we leveraged an existing technique to *dual-issue* miss slice instruction within a single processor core. This provides temporal redundancy by re-executing slice instructions; however instruction fetch and decode still occurs only once in that core and is therefore still vulnerable to soft errors. We close this gap in coverage by populating our *global reorder buffer* with miss slice instructions fetched by each core, and comparing decoded instructions to detect errors.

Finally, this thesis shows that, in addition to performance improvements, redundant execution on three cores can provide error recovery as well. Instruction mismatches trigger a vote between all three cores to identify where the error occurs; any missing state not executed by the third core is reconstructed from the GROB.

7.3 Future Work

We believe that diminishing returns on single-thread performance and increasing prevalence of soft errors make skewed redundancy an appealing technique for future processors. This thesis examined many fundamental design issues and detailed our proposed implementation; however, a large number of interesting design trade-offs remain than warrant additional exploration. We highlight several areas of future work here.

7.3.1 Multithreading

Chip multiprocessors represent one way to enable concurrent execution of multiple application threads on a single die. An alternate technique to expose thread-level parallelism is to allow each core to support multiple threads. Although similar from a software point of view, the underlying implementation of multithreading is often organizationally different than that of CMPs; consequently, we did not discuss such a design point in this thesis. The fundamental problem with implementing skewed redundancy on multithreading hardware is that it requires that the GSQ is located *before* a data cache that is shared between threads. In the design we described, redundant threads have exclusive access to private L1 caches, and can therefore write to them without regard to stores executing on other cores. Allowing skewed redundant threads to share the L1 data cache can be problematic because stores executing by a leading thread can update the cache before stores to the same address by trailing threads. Locating the GSQ between the cores and the L1 solves this problem, but involves substantial microarchitectural modifications, and additionally forces every load to search the GSQ before accessing the L1 data cache.

"Cross-coupling" redundant threads on a multithreaded chip multiprocessor [58] can solve this problem by ensuring that redundant threads do not share an L1 cache. Such a technique could be an especially good fit for coarse-grain (a.k.a switch-on-event) multithreading [23], since threads can be swapped out while waiting for cache misses to complete, effectively consuming zero microarchitectural resources.

Despite these apparent implementation difficulties, multithreading is an efficient and increasingly popular technique to increase processor utilization, and could be an appealing substrate for skewed redundancy.

7.3.2 Adaptability

This thesis described skewed redundancy as an “always on” technique that the system has little or no control over (e.g. a boot-time parameter). Real-world workloads, however, may emphasize higher performance and reliability on some applications over others. Furthermore, the opportunity cost of allocating multiple cores to a single program thread may dynamically change with a system’s load. We therefore believe that an opportunity exists to adaptively control the assignment of cores to applications according to desired levels of service and available resources. Support for such a feature could be implemented at the operating system level, or perhaps more appropriately, at the hypervisor/virtual machine monitor level (where more detailed performance profiling can be implemented). This may be particularly useful in light of several recent papers that show that increasing parallelization of applications emphasizes the performance of the its serialized portions [3][32]. Finer levels of control could disable redundant execution during parallel application phases, and enable it to speed up the serial portions.

7.3.3 Enabling Deeper Speculation

Despite the clear trend towards manycore systems, single-thread performance is likely to still be important in modern applications [32]. The high latency to access main memory is the predominant inhibitor to high performance, and after spending several years studying this problem, my own belief is that runahead-esque techniques that allow execution to continue in the absense of a load miss result are among the most effective and efficient means to this end. Our approach is mostly non-speculative -- unless an exception is raised or a remote store invalidates a cache line accessed by an in-flight load, instruction commit is final and irrevocable. While comparing non-speculative instruction streams to detect errors is intuitive and simple, it introduces stalls that can limit performance when

speculation cannot continue. This effect can clearly be seen in Figure 6-7, where monolithic conventional windows outperform skewed redundancy in several of the benchmarks.

Future work could enable higher performance by relaxing the non-speculative commit constraint and rely on checkpointing schemes to rollback architected state on mis-speculations. Several prior proposals advocate this approach and report considerable speedups [20][21][52][88]. The problem with integrating them with error detection is that wrong-path instructions can mask errors and reduce covereage. If an error affected program output in the same way as the mis-speculation, silent data corruption can occur. Nevertheless, it remains an appealing approach to further improve performance and we leave exploration of enabling techniques to future work.

7.3.4 Statistical Fault Injection

This thesis pessimistically assumes that all single event upsets will eventually propagate to architecturally-visible errors, which are detectable with our proposed techniques. In practice, however, some fraction of bit-flips will be masked at either the logic level (e.g. a gate that “ands” a logical 0 with a corrupted input) or the architecture level (e.g. errors affecting dynamically dead instructions, speculative state, etc.). Combining these masking effects with die area estimations can yield an *architectural vulnerability factor (AVF)* [59] metric that indicates how susceptible a given structure is to particle strikes.

As future work, a more detailed analysis could consider the AVFs of a given microarchitecture to statistically perturb state within the simulator and observe the application’s response. This technique could highlight portions of a design that are outside of our sphere of replication (e.g. shared buses, cache controllers, I/O logic, etc.), as well as

provide realistic estimations of error recovery overhead for a given error rate (in the case where three cores are utilized). Studies of this nature require an integrated performance/functional simulator such as the model used in this thesis (PHARMsim). Performance simulators that do not execute the semantic routines of each instruction or do not retain program values (e.g. trace-driven models) cannot faithfully simulate masking effects and determine if an upset will result in an error. We therefore believe that infrastructures such as ours can be utilized for a new class of real-world vulnerability and overhead studies and illuminate future paths of error-tolerance research.

References

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, New York, 1992.

[2] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. *SIGARCH Comput. Archit. News*, 35(2):470–481, 2007.

[3] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating amdahl's law through epi throttling. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 298–309, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Semiconductor Industry Association. 1999 sia roadmap. 1999.

[5] Todd Austin. Diva: A reliable substrate for deep-submicron processor design. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-32)*, Los Alamitos, November 30–December 2 1999. IEEE Computer Society.

[6] Todd M. Austin and Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 342–351, 1992.

[7] Michel Banatre, Alain Gefflaut, Philippe Joubert, Christine Morin, and Peter A. Lee. An architecture for tolerating processor failures in shared-memory multiprocessors. *IEEE Trans. Comput.*, 45(10):1101–1115, 1996.

[8] Ronald D. Barnes, Erik M. Nystrom, John W. Sias, Sanjay J. Patel, Nacho Navarro, and Wen mei W. Hwu. Beating in-order stalls with "flea-flicker" two-pass pipelining. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003.

[9] Gordon B. Bell and Mikko H. Lipasti. Deconstructing commit. In *Proceedings of the 4th International Symposium on Performance Analysis of Systems and Software*, Austin, Texas, March 2004.

[10] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[11] D.C. Burger and T.M. Austin. The simplescalar tool set, version 2.0. Technical report, University of Wisconsin Computer Sciences, 1997.

[12] J. Adam Butts and Guri Sohi. Dynamic dead-instruction detection and elimination. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 199–210, New York, NY,

[13] J.A. Butts and G. Sohi. A static power model for architects. In *In Proceedings of MICRO-33*, December 2000.

[14] Harold Cain. *Detecting and Exploiting Causal Relationships in Hardware Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, 2004.

[15] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.

[16] Yuan Chou, Lawrence Spracklen, and Santosh G. Abraham. Store memory-level parallelism optimizations for commercial applications. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 183–196, Washington, DC, USA, 2005. IEEE Computer Society.

[17] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 142–153. IEEE Computer Society, 1998.

[18] Neil Cohen, T.S. Sriram, Norm Leland, David Moyer, Steve Butler, and Robert Flatley. Soft error considerations for deep-submicron cmos circuit applications. In *IEEE International Electron Devices Meeting: Technical Digest*, pages 315–318, Washington, DC, USA, 1999. IEEE Computer Society.

[19] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 14–25, New York, NY, USA, 2001. ACM Press.

[20] A. Cristal, M. Valero, J.-L. Llosa, and A. Gonzalez. Large virtual ROBs by processor checkpointing. Technical Report UPC-DAC-2002-39, Univ. Pol. de Catalunya, July 2002.

[21] Adrian Cristal, Daniel Ortega, Josep Llosa, and Mateo Valero. Out-of-order commit processors. In *Proceedings of HPCA-10*, Madrid, Spain, February 2004.

[22] James David Dundas. *Improving processor performance by dynamically pre-processing the instruction stream*. PhD thesis, University of Michigan, 1998. Chairman-Trevor Mudge.

[23] Richard J. Eickemeyer, Ross E. Johnson, Steven R. Kunkel, Mark S. Squillante, and Shiafun Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *Proceedings of the 23rd Annual International Symposium on Computer*

Architecture, pages 203–212, Philadelphia, Pennsylvania, May 22–24, 1996. ACM SIGARCH and IEEE Computer Society TCCA.

[24] Ilya Ganusov and Martin Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 350–360, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Andrew Glew. Mlp yes! ilp no! In *Proceedings of ASPLOS-VIII, Wild and Crazy Ideas Forum*, San Jose, CA, October 1998.

[26] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 98–109, New York, NY, USA, 2003. ACM Press.

[27] Mohamed A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 172–183, Washington, DC, USA, 2005. IEEE Computer Society.

[28] R. Gustafson and F. Sparacio. Ibm 3081 processor unit: Design considerations and design process. *IBM Journal of Research and Development*, 26, 1982.

[29] Tom R. Halfill. An error in a lookup table created the infamous bug in intel's latest processor. *BYTE*, March 1995.

[30] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 58–69, New York, NY, USA, 1998. ACM Press.

[31] J.L Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Ed.*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 2006.

[32] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. Technical report, University of Wisconsin Computer Sciences, 2007.

[33] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor, 2001.

[34] M.S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar. The optimal useful logic depth per pipeline stage is 6-8 fo4. In *International Symposium on Computer Architecture*, 2002.

[35] Jie S. Hu, G.M. Link, Johnsy K. John, Shuai Wang, and Sotirios G. Ziavras. Resource-driven optimizations for transient-fault detecting superscalar microarchitectures.

[36] D. Hunt and P. Marinos. A general purpose cache-aided rollback error recovery (carer) technique. In *Proc. 17th Int. Symp. on Fault Tolerant Computing*, pages 170–175, 1987.

[37] P. Jordan, B. Konigsburg, H. Le, and S. White. Us patent #5805849: Data processing system and method for using an unique identifier to maintain an age relationship between executing instructions. United States Patent 5,805,849, 1997.

[38] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, New York, NY, USA, 1997. ACM Press.

[39] T. Juhnke and H. Klar. Calculation of the soft error rate of submicron cmos logic circuits. *IEEE Journal of Solid-State Circuits*, 30(7):830–834, 1995.

[40] Tejas Karkhanis and J. E. Smith. A day in the life of a data cache miss, 2002.

[41] Jim Keller. The 21264: A superscalar Alpha microprocessor with out-of-order execution. In *Proceedings of the Microprocessor Forum*, October 1996.

[42] Jeffrey Kellington, Ryan McBeth, Pia Sanda, and Ronald Kalla. Ibm power6 processor soft error tolerance analysis using proton irradiation. In *SELSE 3: Proceedings of the 2007 IEEE Workshop on Silicon Errors in Logic - System Effects*, Washington, DC, USA, 2007. IEEE Computer Society.

[43] Ilhyun Kim and Mikko H. Lipasti. Understanding scheduling replay schemes. In *Proceedings of the 10th International Symposium on High-performance Computer Architecture*, San Diego, California, February 2004.

[44] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.*, 48(9):866–880, 1999.

[45] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[46] Shih-Chang Lai, Shih-Lien Lu, and Jih-Kwon Peir. Ditto processor. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 525–536, Washington, DC, USA, 2002. IEEE Computer Society.

[47] A.R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *International Symposium on Computer Architecture*, pages 59–70, 2002.

[48] Kevin Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *Proceedings of ISCA-2000*, Vancouver, June 2000.

[49] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[50] Yi Ma, Hongliang Gao, Martin Dimitrov, and Huiyang Zhou. Optimizing dual-core execution for power efficiency and transient-fault recovery. *IEEE Transactions on Parallel and Distributed Systems*, 18(8):1080–1093, 2007.

[51] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors – a survey. *IEEE Trans. on Computers*, C-37(2):160–173, Feb. 1982.

[52] Jose Martinez, Jose Renau, Michael Huang, Milos Prvulovic, and Josep Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *International Symposium on Microarchitecture (MICRO)*, November 2002.

[53] Dennis McEvoy. The architecture of tandem's nonstop system. In *ACM 81: Proceedings of the ACM '81 conference*, page 245, New York, NY, USA, 1981. ACM Press.

[54] Albert Meixner, Michael E. Bauer, and Daniel J. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *MICRO 40: Proceedings of the 40th annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2007. IEEE Computer Society.

[55] Avi Mendelson and Neeraj Suri. Designing high-performance & reliable superscalar architectures - the out of order reliable superscalar (o3rs) approach.

[56] Gordon E. Moore. Cramming more components into integrated circuits. *Electronics*, 38(8), 1965.

[57] Andreas Moshovos, Scott E. Breach, T.N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, Denver, CO, June 1997.

[58] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 99–110, Washington, DC, USA, 2002. IEEE Computer Society.

[59] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 29, Washington, DC, USA, 2003. IEEE Computer Society.

[60] O Mutlu, J Stark, C Wilkerson, and YN Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *Proceedings of HPCA-9*,

[61] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 96, Washington, DC, USA, 2004. IEEE Computer Society.

[62] Michael Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *VTS '99: Proceedings of the 1999 17TH IEEE VLSI Test Symposium*, page 86, Washington, DC, USA, 1999. IEEE Computer Society.

[63] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, October 1996.

[64] M. Pericas, A. Cristal, R. Gonzalez, D. Jimenez, and M. Valero. A decoupled kilo instruction processor. In *Proceedings of the 12th International Symposium on High-performance Computer Architecture*, Austin, Texas, February 2006.

[65] Moinuddin K. Qureshi, Onur Mutlu, and Yale N. Patt. Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 434–443, Washington, DC, USA, 2005. IEEE Computer Society.

[66] Ravi Rajwar and Jim Goodman. Simplemp multiprocessor simulator. Personal communication., 2000.

[67] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 214–224, Washington, DC, USA, 2001. IEEE Computer Society.

[68] Vimal K. Reddy, Eric Rotenberg, and Sailashri Parthasarathy. Understanding prediction-based partial redundant threading for low-overhead, high- coverage fault tolerance. *SIGARCH Comput. Archit. News*, 34(5):83–94, 2006.

[69] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 25–36, New York, NY, USA, 2000. ACM Press.

[70] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Ar-*

chitecture, pages 148–159, Washington, DC, USA, 2005. IEEE Computer Society.

[71] Mendel Rosenblum. Simos full system simulator. http://simos.stanford.edu.

[72] Eric Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Fault-Tolerant Computing Symposium*, June 1999.

[73] Eric Rotenberg. Exploiting large ineffectual instruction sequences. Technical report, North Carolina State University, 1999.

[74] Toshinori Sato, Akihiro Chiyonobu, and Kazuki Joe. Improving instruction issue bandwidth for concurrent error-detecting processors. In *IWIA '06: Proceedings of the International Workshop on Innovative Architecture for Future Generation High Performance Processors and Systems*, pages 21–28, Washington, DC, USA, 2006. IEEE Computer Society.

[75] N. Saxena and E. McCluskey. Dependable adaptive computing systems – the roar project, 1998.

[76] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. Scalable hardware memory disambiguation for high-ilp processors. *IEEE Micro*, 24(6):118–127, 2004.

[77] L. Sherman. Stratus continuous processing technology – the smarter approach to uptime. Stratus Technologies Whitepaper, 2003.

[78] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on soft error rate of combinational logic, 2002.

[79] Timothy J. Slegel, Robert M. Averill III, Mark A. Check, Bruce C. Giamei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. Ibm's s/390 g5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[80] Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe. Reunion: Complexity-effective multicore redundancy. In *MICRO 39: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2006. IEEE Computer Society.

[81] Jared C. Smolens, Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Fingerprinting: bounding soft-error detection latency and bandwidth. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 224–234, New York, NY, USA, 2004. ACM Press.

[82] Jared C. Smolens, Jangwoo Kim, James C. Hoe, and Babak Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. *micro*, 00:257–268, 2004.

[83] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, Denver, CO, June 1997.

[84] G. Sohi, M. Franklin, and K. Saluja. A study of time-redundant fault tolerance techniques in high-performance pipelined computers. In *Proceedings of 19th Fault-Tolerant Computing Symp.*, pages 436–443, June 1989.

[85] Gurindar S. Sohi, Scott E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[86] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 123–134, Washington, DC, USA, 2002. IEEE Computer Society.

[87] L. Spainhower and T. A. Gregg. Ibm s/390 parallel enterprise server g5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 43(5/6):863, 1999.

[88] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. Continual flow pipelines. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 107–119, New York, NY, USA, 2004. ACM Press.

[89] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 2, Washington, DC, USA, 1998. IEEE Computer Society.

[90] S. Storino, A. Aipperspach, J. Borkenhagen, R. Eickemeyer, S. Kunkel, S. Levenstein, and G. Uhlmann. A commercial multi-threaded RISC processor. In *International Solid-State Circuits Conference*, 1998.

[91] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 257–268. ACM Press, 2000.

[92] Systems Performance Evaluation Cooperative. SPEC benchmarks. http://www.spec.org.

[93] Joel M. Tendler, S. Dodson, S. Fields, and B. Sinharoy. IBM eserver POWER4 system microarchitecture. IBM Whitepaper, October 2001.

[94] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[95] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. *SIGARCH Comput. Archit. News*, 30(2):87–98, 2002.

[96] John von Neumann. First draft of a report on the edvac. *IEEE Ann. Hist. Comput.*, 15(4):27–75, 1993.

[97] David W. Wall. Limits of instruction-level parallelism. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188, New York, NY, USA, 1991. ACM Press.

[98] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for store-wait-free multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 266–277, New York, NY, USA, 2007. ACM Press.

[99] K. C. Yeager. Mips r10000 superscalar microprocessor. *IEEE Micro*, 1996.

[100] Huiyang Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 231–242, Washington, DC, USA, 2005. IEEE Computer Society.

[101] J. F. Ziegler. Terrestrial cosmic rays. *IBM J. Res. Dev.*, 40(1):19–39, 1996.

[102] Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.