Skewed Redundancy

Gordon B. Bell
IBM Corporation
Research Triangle Park, NC
gbbell@us.ibm.com

Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
mikko@engr.wisc.edu

Abstract

Technology scaling in integrated circuits has consistently provided dramatic performance improvements in modern microprocessors. However, increasing device counts and decreasing on-chip voltage levels have made transient errors a first-order design constraint that can no longer be ignored. Several proposals have provided fault detection and tolerance through redundantly executing a program on an additional hardware thread or core. While such techniques can provide high fault coverage, they at best provide equivalent performance to the original execution and at worst incur a slowdown due to error checking, contention for shared resources, and synchronization overheads. This work achieves a similar goal of detecting transient errors by redundantly executing a program on an additional processor core, however it speeds up (rather than slows down) program execution compared to the unprotected baseline case. It makes the observation that a small number of instructions are detrimental to overall performance, and selectively skipping them enables one core to advance far ahead of the other to obtain prefetching and large instruction window benefits. We highlight the modest incremental hardware required to support skewed redundancy and demonstrate a speedup of 6%/54% for a collection of integer/floating point benchmarks while still providing 100% error detection coverage within our sphere of replication. Additionally, we show that a third core can further improve performance while adding error recovery capabilities.

Categories and Subject Descriptors

B.8.1 [Reliability, Testing, and Fault-Tolerance]

General Terms

Performance, Design, Reliability

Keywords

Error tolerance, memory-level parallelism, distributed processing.

1 Introduction

Out-of-order execution hides memory latency at the expense of additional microarchitectural complexity. It has been an effective tool in this regard, and has successfully enabled high performance across several processor generations. However, such complexity is no longer free, and larger cores that consume more power must be carefully balanced against the desire to include more of them on one chip. Increasing relative memory latency and the inherent poor scalability of structures required to mask it have resulted in declining single-thread performance returns, and have consequently favored designs that exploit thread-level parallelism with many simpler cores.

While technology scaling will allow increasingly more cores to fit in a fixed die area, it is not yet clear how software will be able to effectively use them. Chip-level redundant threading (CRT) [20] provides one option by redundantly executing a single software thread on multiple CMP cores to detect soft errors. Unfortunately, this introduces additional overhead due to error checking, shared resource contention, and synchronization. Initial proposals reported significant slowdowns of up to 30%; since then a large body of research has focused on minimizing this overhead, but even in the best case performance is bounded to that of the unreplicated application.

At the same time, other work has focused on mitigating the effect of cache misses through novel microarchitectures intended to increase the number of concurrently executing instructions. Because conventional structures that enable out-of-order execution are notoriously difficult to scale, simply increasing their capacity is often prohibitive due to power and cycle time constraints. Many of these techniques therefore partition the execution of a single thread into disjoint processing elements, using one element to speculatively execute instructions far into the future, and a second element to maintain architected state. Often these processing elements are separate cores in a CMP. However despite the fact that many of these proposals redundantly execute a large fraction of instructions, they do not exploit this mechanism to detect soft errors. In fact, one could easily argue that such techniques *increase* the likelihood of soft errors by spreading out computation over additional die area.

This work relies on the fact that redundant execution used for error detection and used for performance share similar mechanisms. It builds on the CRT philosophy of detecting faults through redundant program execution on unused CMP processor cores, but without increasing execution time. It proposes a novel architecture, called *skewed redundancy*, to force only a single core in a CMP to stall and wait for L2 cache misses to complete, and allows the second processor to advance ahead by dropping the miss and its forward slice of dependent instructions. Eventually a second L2 cache miss is uncovered and again a core is assigned to wait for it. This partitions cache misses between the cores and has the dual benefit of reducing the total number of stalls experienced by each, as well as allowing one to prefetch data for the other. A similar mechanism of identifying the forward slice of cache misses has been used extensively in a variety of other microarchitectural optimizations [9][21][30][10][33][2].

Because the cache-miss slice instructions executed by a single core have no spatial redundancy, their correctness cannot be checked with an output comparator queue as in CRT. To provide coverage for this small subset of instructions, we leverage previously proposed techniques that exploit temporal redundancy by replicating and checking these instructions within the same core [22]. Specifically, instructions within the forward cache *miss slice* are identified and replicated during instruction issue and commit only after their outputs are have been compared and verified.

This paper describes an implementation of skewed redundancy that does not require any cycle-time-critical inter-core shared structures, relying instead on localized and scalable algorithms for managing, coordinating, and synchronizing the cores. A detailed, full-system, and functionally-correct simulation model records average speedups of 6% and 54% over CRT for the SPEC2000 integer and floating point benchmarks, respectively, while providing 100% soft error detection within our sphere of replication with two cores. It also demonstrates that adding a third core can enable additional speedup, as well as provide the capability to dynamically recover from soft errors, instead of merely detecting their presence.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PACT'08, October 25-29, 2008, Toronto, Ontario, Canada.
Copyright 2008 ACM 978-1-60558-282-5/08/10...$5.00.

The remainder of this paper is organized as follows: Section 2 discusses previous proposals to use redundant execution for error detection and for increased performance. Section 3 provides a detailed description of the microarchitectural changes required by skewed redundancy. Section 4 extends this description to detect and recover from transient errors. Section 5 presents results from a detailed full-system timing model that implements skewed redundancy, and Section 6 concludes the paper.

2 Background

2.1 Redundant Threading

Simultaneous and redundantly threaded (AR-SMT or SRT) processors execute an identical copy of a program on an unused hardware thread or processor core [25][24]. Errors can be detected by buffering retired stores in a store comparator (SC) queue where they are compared to identical stores executed on a second thread. If a mismatch is detected in either a store's data or address, an error is signaled to the processor so that it can respond appropriately. Combining different instances of the same dynamic store presents the appearance to the memory system that a single thread is running and maintains uniprocessor program semantics. Chip-level redundant threading (CRT) [20] extends this concept to run multiple instances of the same thread on different cores in a chip multiprocessor (CMP) rather than on separate thread contexts in a simultaneous multi-threaded processor (SMT). Depending on the degree of hardware replication in an SMT, CRT is able to provide better performance than SRT by removing contention for resources shared among threads (such as the physical register file, issue window, ROB, load and store queues, fetch unit, execution units, etc.). Subsequent follow-on work [11][28][27] has further improved CRT performance, however even in the best case its performance is limited to that of the original unreplicated thread.

2.2 Enabling Large Virtual Windows

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][3][8][19][30]. 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.

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 [31][16][12] to extensive support for resolving both register and memory dependences in the Multiscalar proposal [29]. 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.

Skewed redundancy leverages a technique that poisons a load cache miss and its forward slice of dependent instructions first proposed by runahead execution [9][21] as a prefetching mechanism. Dual-core execution (DCE) [33] extends runahead execution to two-core chip multiprocessors and achieves improved performance by addressing runahead's shortcoming of continually needing to squash the runahead thread and restart execution at the cache miss. It utilizes a front core that speculatively executes instructions independent of cache misses and passes these results through an instruction queue to a back core that verifies them. Discrepancies indicate a miss-dependent branch or store address and result in a full squash of the front core.

2.3 Achieving Error Detection and Performance

Very few proposals attempt to simultaneously achieve the combined goals of both increased performance and soft-error detection. The authors of DCE recently extended their original design [18] to include error checking capabilities, 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 (minus 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 apply a CRT optimization described by [27] and 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. Section 5 shows that our *global reorder buffer* with 128 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 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 [32] use a complementary method to create "slack" between two cores on a CMP and improve performance. Instead of discarding L2 cache misses and their dependent instructions, they dynamically remove instructions predicted to have no effect on program state (dubbed "ineffectual"). This gen-

erally results in less dramatic speedups than skewed redundancy because ineffectual instructions may not be costly to execute compared to L2 misses. Slipstream processors were also extended to detect soft errors in [23], however this proposal provides less than 100% coverage.

Recent work by Aggarwal et al. has shown that shared resources in CMPs can become single points of failure that can reduce the effectiveness of techniques like redundant threading [1]. The implementation of skewed redundancy described in this paper includes a shared cache and requires three shared, global structures (the *global reorder buffer* or GROB, the *global architectural register file* or GARF, and the *store comparator queue* or SC, all described in Section 3). Since these are all storage structures, their error rates can be lowered substantially with ECC, and their impact on the chip's overall vulnerability is not significant. We leave exploration of redundancy for these structures--several of which are integrally tied in to the chip's error-detection mechanisms--to future work.

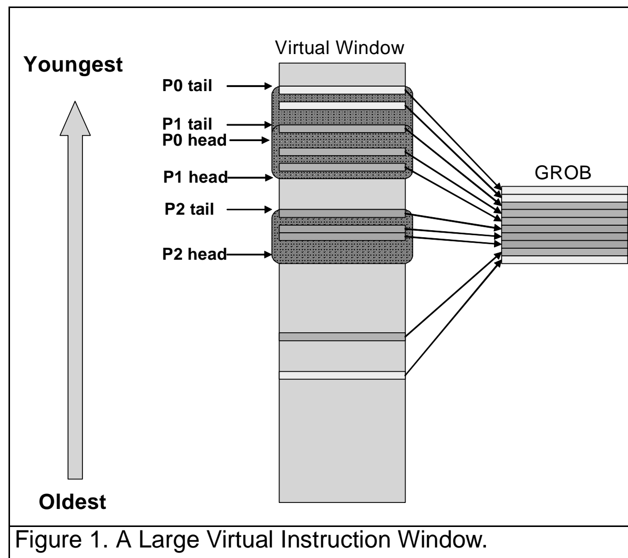

Figure 1. A Large Virtual Instruction Window.

3 Design Details

3.1 Overview

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. Furthermore, the higher level must also provide a policy and mechanism for partitioning work across 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. This section describes skewed redundancy in terms of an arbitrary number of processor cores. Section 4 provides more detail regarding the error detection and recovery implications of using two vs. three cores, and Section 5 will present performance results for both of these configurations.

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 the set of unique instructions executing on any core at one time. Figure 1 depicts this concept. Each processor conventionally executes instructions in the global instruction stream and allocates a local ROB slot and other associated execution resources. Head and tail pointers indicate their oldest and youngest in-flight instructions. Because multiple processors may be simultaneously executing the same set of instructions, their local windows may overlap, as is the case with processors P0 and P1.

A monolithic instruction window would need to buffer all unique in-flight instructions in the virtual window, which Section 5 will show can span thousands of instructions. However because most buffering and forwarding can be satisfied through the smaller individual ROBs, we only need to provide a mechanism that handles 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 L2 misses, which previous work has shown to be quite small [30]. We propose containing slice instructions in a *Global Reorder Buffer* (GROB) to facilitate inter-core communication and precise exception recovery. Figure 1 illustrates the relationship between the GROB, virtual instruction window, and local instruction windows. While the GROB only needs to contain enough entries to contain the forward slice of the miss, the remainder of this paper shows that actual number of entries that participate in synchronization between processing elements is far fewer.

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: miss slice 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.

3.2 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 logical timestamp used for relative age comparisons. When a processor core commits a slice instruction that it executed it copies the result into the GROB at the index specified by its *GROB tail*. When a processor commits a slice instruction that it did not execute, 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.

After all 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. However because few registers will depend on miss slice instructions at any point, it can potentially be made significantly smaller by mapping architected registers onto a smaller pool of physical storage locations. The entry timestamps are used to determine if the source register value resides in the GROB or the GARF. A newer producer timestamp indicates that GROB entry has been recycled and reallocated to a younger instruction and that the correct value has been retired to the GARF. Their operation is illustrated by example in the following sections.

3.3 Slice Joins

Localizing computation dependent upon cache misses removes the primary culprit of full-window stalls from other instruction windows. The processor responsible for handling the miss waits for data to return from memory and is the only core that 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. 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. We 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.

In addition to an INV bit, each register is associated with a P ("Private") bit to indicate dependence on a miss executed by that processor. 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 1. For a given combination of source states, each row indicates the resulting destination register's state, whether the instruction is locally executed, and whether a GROB read or write is necessary. If a processor has both operands locally available in the G or P state, it executes the instruction normally. If either operand is marked INV the processor discards the instruction and continues executing, with the exception of when the left operand is marked P and the right operand is marked INV. In this case no processor has both inputs and the core with the left operand needs to retrieve the right operand value from the GROB as follows: when the instruction reaches the 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 this point it has both source operands and can execute the instruction.

Table 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

Figure 2 provides an example. It shows two processors as well as their interaction with the additional hardware structures described above. P0's first instruction loads address [r8] and upon detecting that is misses in the L2 cache, determines that it will be responsible to wait for the data to return from memory. 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. The fourth instruction misses the L2 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 incremented 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 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 L2 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 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 in 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 timestamp is newer than the load join instruction, indicating that the entry has been recycled, it instead retrieves r5 from the GARF. Logical timestamps 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 [13] to deal with counter overflows.

3.4 Control Flow

While individual processors 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 an L2 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 processor's ROB head, it retrieves the operand value from the GROB/GARF in the same way it would for a miss

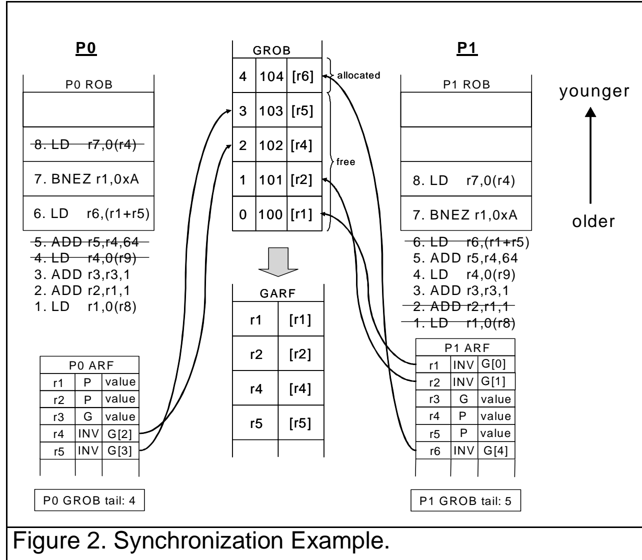

Figure 2. Synchronization Example.

slice join operand. In the example presented in Figure 2, 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 timestamp 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 that depend on cache misses are more likely to be mispredicted, and those mispredictions are likely to occur shortly after the branch [14]. Therefore even if branches marked invalid were speculatively committed, little additional useful work would be exposed. Our own experiments (not detailed here) confirm these findings.

3.5 Maintaining Precise State

Committing an instruction updates the architected state of the processor and deallocates any resources it consumed during execution. The danger in committing instructions 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 2. 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.

3.6 Partitioning Misses

A spectrum of partitioning policies can be envisioned for assigning cores to wait for misses. We adopt a simple and effective approach that allows the first core to reach a miss to discard it; the second core will then be responsible for waiting for the miss to complete if it is still pending. Whether or not another core has reached a miss can be implied by the existence of an MSHR target with a matching instruction identifier. Section 4.3 extends miss partitioning from two to three cores.

3.7 Enforcing Memory Dependences

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. We propose to leverage the store comparator queue (SC) queue already present in CRT for this purpose. Similar to a conventional store queue, the SC can be used to prevent WAW and RAW violations.

3.7.1 WAW Violations

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 SC when they locally commit and are released to the L2 in program order, as proposed by CRT.

Similar to the GROB, all cores maintain a SC tail index that points to the entry corresponding to its oldest uncommitted store. When a processor commits a store it writes its address into the entry pointed to by its SC tail and, if locally available, the store data. If a SC entry has already been updated by a second core, the address and data are read and compared. A mismatch indicates the presence of an error, and the processors are signaled so that they can respond appropriately. After a processor writes a store into the SC it increments its tail modulo the SC size to point to the next entry. As the trailing tail advances past store entries they are released to the shared L2 in program order.

3.7.2 RAW Violations

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, 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 SC. 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 dependent 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 SC, and its address is compared against older stores in the SC. 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 SC.

When a processor commits a store it writes its private L1 cache. However, if the store data is not available (due to an invalid source register), permitting another core to update the SC without concurrently updating the L1 can violate RAW ordering. In this case a younger load of the same cache line could read a stale value from the L1 and fail to identify that the correct data should be supplied by a older miss-dependent store in the SC. Therefore when a processor commits a store with invalid data it must invalidate any matching block from its L1. This forces younger loads to miss the L1 and search the SC for an address match. Figure 3 depicts this algorithm.

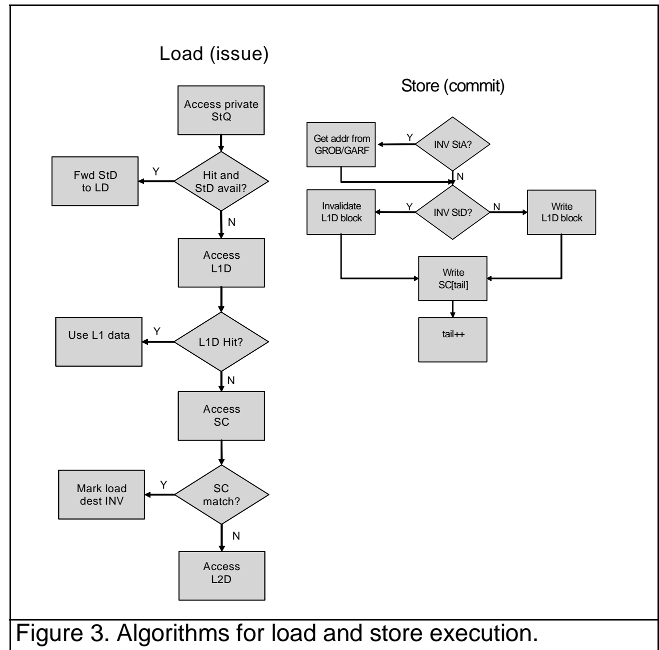

Figure 3. Algorithms for load and store execution.

3.7.3 Store Addresses

Normally, when a source operand is invalid the instruction is not executed by that processor. However, treating stores in this manner can result in a loss of memory dependence information between instructions. If a store is discarded, an older store may incorrectly forward data to a younger load. The load actually needs the value produced by the younger store, which is no longer visible. We deal with this problem by forcing all processors to compute all store addresses. If a store address register is marked invalid it is read from the GROB or GARF as it would for a slice join or invalid branch operand. This ensures that the addresses of all stores older than a given load appear either in that processor's local store queue or the SC. We have observed that 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 performance.

3.7.4 Complexity

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

Stores: Inserting a committed store into the SC involves a simple RAM access indexed by the processor's SC tail pointer. The address field is compared or written with store data (if available). Because this SC 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 CRT.

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 SC. When an L1/store queue miss occurs the SC 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, the SC needs to be content-addressable by load address, however it can be accessed in parallel with the L2. 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 SC access latency does not exceed the L2 access latency. Techniques such as Bloom filters [4] have been shown to drastically reduce store queue searches [26], and it would be straightforward to apply a similar mechanism here. This could provide an additional level of filtering to further reduce SC accesses.

3.8 Observing Misses Consistently

Note that 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. However, to ensure that all cores agree on these assignments, all cores must have a consistent view 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.

First of all, to ensure that any trailing processors treat references to a block that missed in a leading processor as misses, we implement a transient state for pending cache blocks that flags all intervening references to such pending blocks as misses. When the cache block data arrives at the MSHR, it is 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.

Second, to detect the case where a leading processor hits on

a block that is later replaced, control logic at the GROB ensures that whenever any core has reported a miss to a block, all cores must similarly report a miss before global commit can occur past that point. If the global commit pointer reaches an instruction that missed the cache in some processors, but hit in others, a global flush and replay is triggered to ensure that all cores observe the same (new) version of the block.

The second technique enforces coherence across different instances of the same load, but it also enforces load-to-load coherence across different loads to the same address. As a final enhancement, to enforce sequential consistency in a system with multiple cache-coherent skewed redundant processors, we monitor invalidates to blocks that are in transient state. If an invalidate gets applied to such a block, all cores that have speculatively accessed it will get squashed and will replay from the oldest load to reference the block (usually, but not necessarily, the first load to touch the block and cause the miss). Prior work has shown that remote invalidates that match in-flight loads are exceptionally rare, even in large multiprocessor shared-memory systems [5].

4 Dealing with Soft Errors

4.1 Error Detection

CRT executes every instruction on two cores and detects soft errors by comparing their outputs in its store comparator queue (SC). Skewed redundancy uses this spatially redundant approach to error detection for the vast majority of instructions. However store data that depends on an L2 miss will only execute on a single core, and therefore cannot be checked using the SC. To provide coverage for this small subset of instructions, we leverage a previous proposals by Ray et al. [22] that provide error detection through temporal redundancy within a single core.

In this scheme, the issue logic is modified such that instructions with invalid source operands are duplicated before they are issued to functional units. What was originally a single miss-dependent instruction becomes two distinct instructions in the execution pipeline. Neither instruction is permitted to commit until both have completed and verified their outputs with the other. As originally proposed, this technique suffers a significant performance penalty compared to original unreplicated case (30% average reported slowdown) because twice as many instructions execute, in addition to instruction checking and synchronization overhead. However, two properties of skewed redundancy drastically reduce this cost: first, few instructions depend on L2 cache misses and need to be dual-issued. Most instructions are executed on two cores, and error checking is performed by the SC. Second, even if duplicately issued instructions slightly slow down one core, the other core can continue uncovering cache misses unimpeded provided that the SC and GROB can accommodate this additional slack.

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 [15], while newer instructions inherit this property through the renaming logic.

4.2 Increasing Coverage

We now point out additional classes of errors that are not detected by our proposed scheme, and describe a simple detection mechanism that increases coverage to 100% within the sphere of replication. One example is globally-executed instructions that feed into forward miss slices. We may fail to detect errors in such instructions because they are neither dual-issued within a pipeline, nor necessarily terminate in redundant stores checked by the SC. A second case involves errors affecting front-end pipeline stages of miss slice instructions. Since instructions are not replicated until they reach the issue stage, errors that occur prior to this point (e.g. during fetch or decode) may not be detected.

We handle both of these cases with simple extensions to the GROB. To cover global slice inputs, we extend each GROB entry with an additional field to contain global values. When slice instructions commit, they write (or check) this value; a mismatch indicates an error. To detect front-end data corruption of slice instructions, the leading core additionally inserts decoded instructions into the GROB. The trailing core then checks these values against its own local copies to detect fetch or decode errors. Both of these additionally fields are available in from the local ROB entry, and can simply be copied to the GROB as slice instructions commit. This mechanism is identical to the *slice data buffer* proposed by continual flow pipelines [30].

Increasing coverage in this manner nominally increases the size of each GROB entry; however, since there are relatively few entries, these changes are not significant. It also requires that all slice instructions access the GROB when they commit, rather than only those instructions executed by the core producing the miss-dependent value. We note that this only affects the minority of instructions that depend on cache misses, and that this error checking can be performed off of the critical execution path.

4.3 Error Recovery

CRT only detects transient errors that eventually propagate to stores, and therefore is unable to correct and recover from errors. A mismatch in the store comparator could indicate that an erroneous value was written to a register far in the past and has subsequently fanned out to an arbitrary number of dependent instructions. Skewed execution as we have described it inherits this shortcoming of CRT. This section proposes adding a third core to also provide error recovery, and also shows that the cost of the third core is mitigated by even higher performance than is possible with two cores.

With three cores available, most instructions execute on all three cores, and the store comparator can use triple-modular redundancy (TMR) to vote on and correct errors when a mismatch is detected. This requires additional store comparator fields as follows: the first two cores to commit a specific dynamic store write its address and data into the store comparator entry. The third core to reach the store compares the two buffered values to its own, and on a mismatch identifies the core with the minority value. As long as that core is not trailing behind the other two, it can be squashed, and the leading core can read missing state from the GROB/GARF and restart execution. On the other hand, if the minority core is trailing, then the error may prevent it from catching up and producing private results marked invalid in the leader's ARF. To deal with this problem, we propose replaying all slice instructions from the GROB. The GROB extensions proposed in the previous section provide all information required for this: decoded instructions and global operands are written by the leading core, and all non-global slice inputs can be read from the GARF. Note that this is identical to the method used by continual flow pipelines [30] to replay miss slice instructions from its *slice data buffer.*

In addition to providing error recovery capabilities, a third core can also improve performance of skewed redundancy by increasing the number of concurrent L2 cache misses. 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 is significantly behind the first (for example,

Table 2: Machine Configuration

Parameter	Value
Out-of-order execution	4-wide fetch/issue/commit, 10-cycle pipeline, 64 ROB, 32 LSQ, 64 rename registers, 16 IQ, memory dependence predictor
Functional Units	4 integer ALU, 2 FP ALU, 2 integer MULT/DIV, 2 FP MULT/DIV, 2 memory ports
Branch Prediction	Combined bimodal (16k entry) / gshare (16k entry) with selector (16k entry), 16 RAS, 1k entry 4-way BTB
Memory System (latency)	32KB 2-way 32B line IL0 (2), 32KB 2-way DL0 (2), 256KB 8-way unified L1 (15), 1MB 8-way unified L2 (50), 64B lines, main memory (300), Power4-style hardware prefetcher
Skewed Redundancy	128-entry GROB, 10-cycle GROB read latency, 128-entry SC, additional pipeline commit stage to insert store into SC.

because it is waiting on another miss to complete), many cycles may elapse before it executes dependent instructions skipped by the first. This situation can occur when L2 misses depend on prior L2 misses. 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 allowing such miss-dependent misses to issue early. As before, the first core to reach a miss discards it. The second core has the option of waiting for it to complete, or to similarly discard it. Clearly, always discarding this miss will cause it to behave identically to the leading core, and always waiting for the miss will cause it to execute the miss at the same time as the trailing core. We therefore adopt 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. This helps increase the likelihood that it will execute dependent misses early. Simulation results for two and three cores will be presented in the following section, and set this minimum distance at twice the ROB capacity.

5 Results

5.1 Methodology

This section presents characterization and performance data collected with PHARMsim [6], a full-system execution-driven timing model of the PowerPC instruction set architecture. 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. We implemented skewed redundancy and its related hardware structures on top of this model as they have been described in this paper. Table 2 presents the machine model configuration used to collect the data in this section. It utilizes moderately-aggressive out-of-order superscalar cores similar to those found in modern chip multiprocessors, and modestly-sized additional structures required to support skewed redundant execution. The characterization data presented corresponds to the use of two redundant cores; additional performance results for a third core is presented in the section describing performance improvements.

A collection of SPEC2000 integer and floating point benchmarks were compiled for the PowerPC instruction set with peak optimization by the IBM AIX optimizing compiler, and executed with the reference input sets. Benchmarks were fast forwarded 5 billion instructions before timing simulation was performed on the following 200 million instructions.

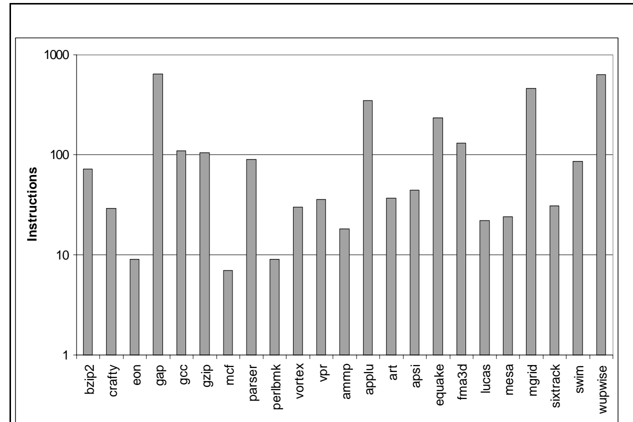

Figure 4. Virtual Window Size. Distance between the oldest and youngest instructions among both cores (log scale).

5.2 Supporting Characterization Data

Skewed redundancy can speedup execution by decoupling cores from each other and provide the benefit of a single large instruction window. Figure 4 presents the average "slip" between two redundant cores. Specifically, it measures the number of dynamic instructions between the oldest and youngest instructions on either core. It shows that the *average* distance can approach 1000 instructions for some benchmarks. This provides an approximation of the average number of in-flight instructions that a single core would need to sustain to achieve comparable performance. Because the average is certainly deflated by periods with few or no cache misses, the peak slip can be significantly higher than the average. Figure 5 emphasizes that few instructions depend on L2 misses (as previously noted by [14][30]). It indicates the fraction of instructions that are globally executed by both cores, by only P0, and by only P1. Generally most instructions are executed by both cores (over 90% in most cases), leading to few synchronizations through the GROB and few instructions that need to be dual-issued to provide complete error detection coverage.

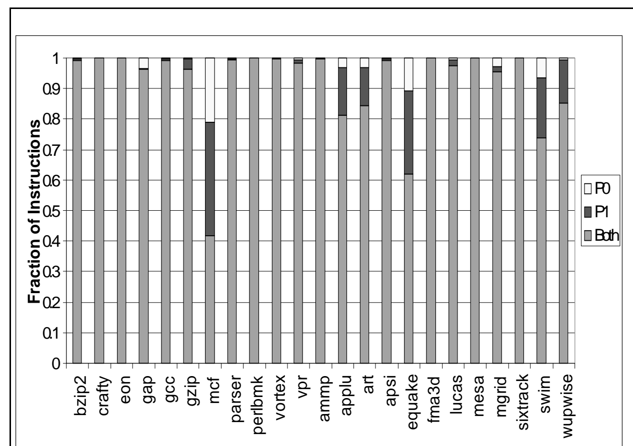

Figure 5. Fraction of instructions executed by P0 and P1.

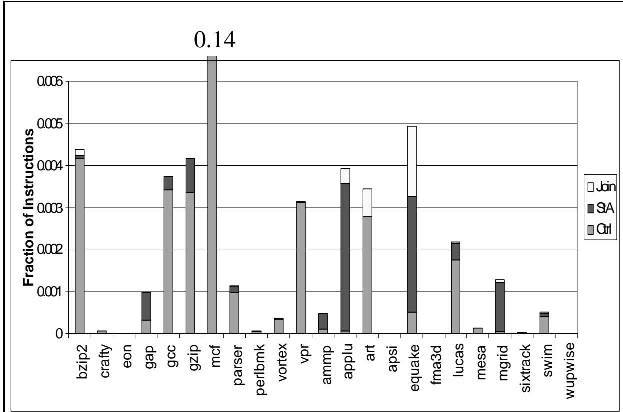

Figure 6. GROB/GARF Reads. Fraction of all instructions that require a value produced by another core and the reason for the communication: invalid branch inputs (Ctrl), invalid store address (StA), or dataflow join (Join).

Figure 6 indicates the fraction of instructions that need to retrieve a missing source register value from the GROB/GARF, as well as which of the reasons described in Section 3 caused the read. That this figure is low is important, not so much because of the latency to access the shared structures, but rather because a read implies that a core skipped a miss-dependent instruction and the value will not likely be available for some time. In all but one case, less than 0.5% of all instructions need to retrieve an invalid source value. The exception, *mcf*, traverses a linked list in a tight loop. The branch condition depends on a cache miss to a list entry; therefore, almost all of its branches require a GROB read (14% of all instructions). The most common reason for a read is that a branch condition or target address register is marked invalid, and the source operand is required to verify the prediction. The next most common cause of synchronization is to retrieve missing store addresses to enforce proper memory ordering as described in Section 3.7.3. Far less common reasons include dataflow joins (indicating that few instructions simultaneously depend on multiple misses handled by multiple cores), reads of non-renamed system registers, or any combination of multiple reasons. Data points corresponding to these other reasons are not shown due to their insignificant frequency. We point out that benchmarks that exhibit large average virtual window sizes in Figure 4 have few instructions that require synchronization through the GROB. Conversely, benchmarks with frequent synchronizations prevent either core from advancing very far ahead of the other, and consequently limit performance. Note that the fraction of instructions required to read the GROB is significantly lower than the fraction of slice instructions which that write it, indicated by Figure 6 and Table 1: when a slice instruction commits, it is not known if a later exception will require that value in the process of reconstructing architected state.

5.3 Performance

Figure 7 presents speedup normalized to a two-core CRT system. We do not present results for the unreplicated single-thread case, since our baseline implementation of CRT moves store verification off of the critical execution path and into the SC, and closely approaches this level of performance. The second bar shows that skewed redundancy can provide significant speedup over CRT. Integer benchmarks speed up by an average of 6% (10% not counting *mcf*), and floating point benchmarks by an average 54%. In particular, several of the floating point applications have a significant number of L2 misses and their performance is severely constrained by the baseline 64-entry instruction window. Techniques that overlap more concurrent misses can therefore provide dramatic speedups. The third bar corresponds to three-core skewed redundancy configuration. As described in Section 4.3, a third core can further improve performance when a large amount of slack exists between the two cores. Consequently, a third core tends to help benchmarks that already obtain a significant speedup from skewed redundancy, while not affecting those with only marginal initial speedup. Adding a third core increases the average integer speedup from 6% to 7%, and improves average floating point performance from 54% to 57%.

An alternative to skewed redundancy could implement conventional CRT on top of high-MLP cores. The first bar represents such a design, and uses cores that add runahead execution [9][21] to the baseline microarchitecture. We observe that in almost all cases we outperform runahead execution, highlighting its 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. Furthermore, runahead on top of CRT adds another dimension of redundant instruction execution, which burns additional power. Our own experiments indicate that runahead executes an average of 7% / 77% additional instructions *per core* for the integer / floating-point benchmarks (and up to nearly 4x more instructions in some cases), compared to our fixed overhead of roughly twice as many instructions for two cores. Our solution therefore consumes less power and delivers better performance than this alternative.

The final bar represents the performance potential by using an inefficient large-window core that is not amenable to chip multiprocessors and high-frequency designs, and serves as an indicator of the instruction throughput a conventional large and aggressive core is capable of. This data point models a 512-entry ROB, with a 256-entry LSQ and a 128-entry issue window. As Figure 4 shows, skewed redundancy can enable virtual window sizes that far exceed even 512 entries, and therefore provide speedups beyond that of the large-window core (e.g. *gzip, applu, mgrid, swim*). Of course, the 512-entry ROB provides no error tolerance, which is a primary benefit of skewed redundancy.

6 Conclusion

This work extends prior research of executing identical programs on multiple cores to detect transient errors. It makes the observation that not all instructions have the same impact on system performance and selectively discarding them can realize significant performance gains while still providing 100% soft error coverage. Two system design trends accentuate this point: larger L2 (and L3) caches will decrease the fraction of miss-dependent non-redundant instructions. At the same time, increasing relative memory latency will make these misses more expensive in terms of performance, and skipping them more attractive. While several proposals have attacked the problems of soft errors and increasing relative memory latency in isolation (and often at the expense of each other), this is among the first to simultaneously achieve the benefits of both.

This paper proposed *skewed redundancy*, and presented a detailed description of the design requirements to implement it. Our design requires minimal changes to the underlying microarchitecture, and utilizes additional structures that are small, simple, and outside of the processor cores. Our results show these struc-

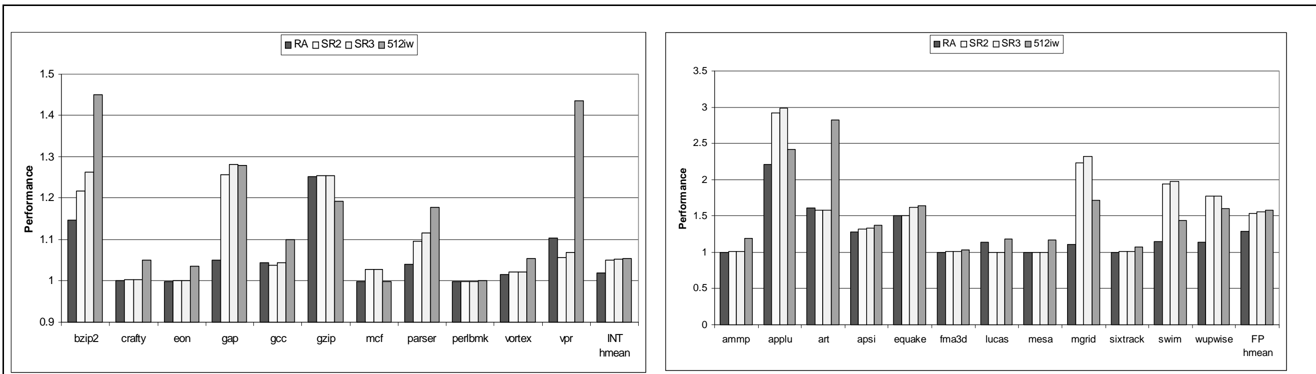

Figure 7. Speedup. From left to right: skewed redundancy with runahead execution, skewed redundancy with two cores, with three cores, with a 512-entry ROB. All results are normalized to CRT with decoupled store checking.

tures are infrequently accessed, and can therefore tolerate long, multi-cycle access latencies.

We report average SPEC2000 integer/floating point speedups of 6%/54% for two out-of-order superscalar cores, and show that additional performance as well as error correction is possible with a third core. Finally, we point out that small in-order cores are becoming a common and compelling design point; we assert that skewed redundancy is an appealing substrate for such simple cores to boost their low single-thread performance, as well as the fact that a design with simple cores is likely to have more of them available for error detection and tolerance techniques. Although we have not reported such results in this paper, additional experiments show that in-order speedups substantially exceed the out-of-order results presented here.

Acknowledgments.This work was supported in part by NSF grants CCR-0133437, CCF-0429854, and CCF-0702272, as well as grants and equipment and donations from Intel and IBM.

References

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

[2] R. Barnes et al. Beating in-order stalls with "flea-flicker" two-pass pipelining. In *MICRO-36*, 2003.

[3] G. Bell and M. Lipasti. Deconstructing commit. In *ISPASS-4*, Austin, Texas, March 2004.

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

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

[6] H. Cain, K. Lepak, B. Schwarz, and M. Lipasti. Precise and accurate processor simulation. In *CAECW*, Feb. 2002.

[7] A. Cristal et al. Large virtual ROBs by processor checkpointing. Tech. Rep. UPC-DAC-2002-39, Univ. UPC, July 2002.

[8] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. *HPCA-10*, Madrid, Spain, Feb. 2004.

[9] J. Dundas. *Improving processor performance by dynamically pre-processing the instruction stream*. PhD, 1998.

[10] I. Ganusov and M. Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *PACT '05*, pages 350–360, Washington, DC, USA, 2005.

[11] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *ISCA '03*, pages 98–109, New York, NY, USA, 2003.

[12] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII*, 1998.

[13] 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, 1997.

[14] T. Karkhanis and J. Smith. A day in the life of a data cache miss, In Workshop on Memory Performance Issues, 2002.

[15] I. Kim and M. Lipasti. Understanding scheduling replay schemes. In *HPCA-10*, San Diego, California, Feb. 2004.

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

[17] A.R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *ISCA*-29, pages 59–70, 2002.

[18] Y. Ma, H. Gao, M. Dimitrov, and H. Zhou. Optimizing dual-core execution for power efficiency and transient-fault recovery. *IEEE TPDS*, 18(8):1080–1093, 2007.

[19] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *MICRO-25*, Nov. 2002.

[20] S. Mukherjee, M Kontz, and S. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *ISCA-29*, 2002.

[21] O Mutlu, J Stark, C Wilkerson, and YN Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *HPCA-9*, Jan. 2003.

[22] J. Ray, J. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *MICRO 34*, 2001.

[23] V. Reddy, E. Rotenberg, and S. Parthasarathy. Understanding prediction-based partial redundant threading for low-overhead, high- coverage fault tolerance. In *ASPLOS-XII,* October 2006.

[24] S. Reinhardt and S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA-27*, NY, 2000.

[25] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *FTCS-29*, June 1999.

[26] S. Sethumadhavan, R. Desikan, D. Burger, C.. Moore, and S. Keckler. Scalable hardware memory disambiguation for high-ilp processors. *IEEE Micro*, 24(6):118–127, 2004.

[27] J. Smolens, B. Gold, B. Falsafi, and J.. Hoe. Reunion: Complexity-effective multicore redundancy. In *MICRO 39*, 2006.

[28] J. Smolens, J. Kim, J. Hoe, and B. Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. *MICRO-37*, 2004.

[29] G. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar processors. In *ISCA-22*, June 1995.

[30] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *ASPLOS-XI*, 2004.

[31] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA-4*, 1998.

[32] K. Sundaramoorthy et al. Slipstream processors: improving both performance and fault tolerance. In *ASPLOS-IX*, 2000.

[33] H. Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *PACT '05*, 2005.