Reunion: Complexity-Effective Multicore Redundancy

Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe
Computer Architecture Laboratory (CALCM)
Carnegie Mellon University, Pittsburgh, PA 15213
http://www.ece.cmu.edu/~truss

Abstract

To protect processor logic from soft errors, multicore redundant architectures execute two copies of a program on separate cores of a chip multiprocessor (CMP). Maintaining identical instruction streams is challenging because redundant cores operate independently, yet must still receive the same inputs (e.g., load values and shared-memory invalidations). Past proposals strictly replicate load values across two cores, requiring significant changes to the highly-optimized core.

We make the key observation that, in the common case, both cores load identical values without special hardware. When the cores do receive different load values (e.g., due to a data race), the same mechanisms employed for soft error detection and recovery can correct the difference. This observation permits designs that relax input replication, while still providing correct redundant execution. In this paper, we present Reunion*, an execution model that provides relaxed input replication and preserves the existing memory interface, coherence protocols, and consistency models. We evaluate a CMP-based implementation of the Reunion execution model with full-system, cycle-accurate simulation. We show that the performance overhead of relaxed input replication is only 5% and 6% for commercial and scientific workloads, respectively.*

1. Introduction

Chip multiprocessors (CMPs) have emerged as a promising approach to give computer architects scalable performance and reasonable power consumption within a single chip [3,11,16]. However, increasing levels of integration, diminishing node capacitance, and reduced noise margins have led researchers to forecast an exponential increase in the soft-error rate for unprotected logic and flip-flop circuits [10,19]. Recent work [9,14,22] advocates leveraging the inherent replication of processor cores in a CMP for soft-error tolerant redundant execution by pairing cores and checking their execution results.

Because CMP designs maintain the familiar shared-memory programming model, multicore redundant architectures must provide correct and efficient execution of multithreaded programs and operating systems. Furthermore, redundant execution must not introduce significant complexity over a non-redundant design. Ideally, a single design can provide a dual-use capability by supporting both redundant and non-redundant execution.

Redundant designs must solve two key problems: maintaining identical instruction streams and detecting divergent execution. Mainframes, which have provided fault tolerance for decades, solve these problems by tightly lockstepping two executions [4,20]. Lockstep ensures both processors observe identical load values, cache invalidations, and external interrupts. While conceptually simple, lockstep becomes an increasing burden as device scaling continues [5,12].

Researchers have proposed several alternatives to lockstep within the context of CMPs. Both Mukherjee et al. [14] and Gomaa et al. [9] use a custom load-value queue (LVQ) to guarantee that redundant executions always see an identical view of memory. A leading core directly issues loads to the memory system, while a trailing core consumes a record of load values from the LVQ. Although the LVQ produces an identical view of memory for both executions, integrating this *strict input replication* into an out-of-order core requires significant changes to existing highly-optimized microarchitectures [14].

Strict input replication forbids using existing cache hierarchies for the redundant execution and requires changes to critical components of the processor core and cache hierarchy. In contrast, *relaxed input replication* permits redundant executions to independently issue memory operations to existing cache hierarchies. We observe that, even for shared-memory parallel programs, relaxed input replication produces

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

the correct result in virtually all cases. In the case when load values differ between the redundant cores, called *input incoherence*, mechanisms for soft error detection and recovery can correct the difference [17].

In this paper, we propose the *Reunion* execution model, which exploits relaxed input replication for soft-error tolerant redundant execution across cores. While Reunion allows redundant cores to issue memory operations independently, we prove that Reunion designs can maintain correct execution with existing coherence protocols and memory consistency models. Reunion provides detection and recovery from input incoherence using a combination of light-weight error detection [21] and existing precise exception rollback—the same mechanisms needed for soft-error tolerance.

We make the following contributions:

- **Input incoherence detection.** We observe that light-weight detection mechanisms for soft errors can also detect input incoherence. This observation enables a single recovery strategy for both soft errors and input incoherence.
- **Reunion execution model.** We present formal requirements for correct redundant execution using relaxed input replication in a multiprocessor. These requirements do not change the existing coherence protocol or memory consistency model.
- **Serializing check overhead.** We observe that checking execution at instruction retirement incurs stalls on serializing events, such as traps, memory barriers, and non-idempotent instructions. Architectures that encounter frequent serializing events will suffer a substantial performance loss with any checking microarchitecture.

We evaluate Reunion in a cycle-accurate full-system CMP simulator. We show that the Reunion execution model has an average 9% and 8% performance impact on commercial and scientific workloads, respectively, with a 5-6% performance overhead from relaxed input replication.

Paper Outline. In Section 2 we present background on soft error detection and redundant execution. Section 3 presents the Reunion execution model. We discuss a CMP implementation in Section 4 and its evaluate performance in Section 5. We conclude in Section 6.

2. Background

2.1. Fault Model

Our fault model targets soft errors that cause silent data corruption, such as transient bit flips from cosmic rays or alpha particles. We assume that the processor's datapath is vulnerable to soft errors from fetch to retirement, but that the less-vulnerable control logic [19] is protected by circuit-level techniques. Designers already protect cache arrays and critical communication buses with information redundancy (e.g., ECC) [20]. However, the complex layout and timing-critical nature of high-performance processor datapaths precludes these codes within the pipeline. Unretired speculative state, such as speculative register files and the issue queue, can remain unprotected. We assume retired architectural state arrays can absorb the latency and area overhead of ECC protection (e.g., the architectural register file and non-speculative store buffer).

In this work, we investigate microarchitectures that detect and recover from virtually all soft errors, but in very infrequent cases, can leave them undetected or uncorrected. Architects design microprocessors to meet soft error budgets [13] and our design can be engineered to meet the desired budget.

2.2. Redundant Execution

The "sphere of replication" defines three key design requirements for redundant execution [17]. First, all computation within the sphere must be replicated in space or time. Second, all inputs entering the sphere must be replicated for each execution. Finally, all outputs leaving the sphere must be checked to prevent errors from propagating outside the sphere.

We now discuss the two dominant forms of redundant execution in microprocessors in the industry and research communities: lockstep and multithreading.

Lockstep. Classical lockstep redundant execution—where identical processing elements are tightly-coupled on a cycle-by-cycle basis—has long existed in mainframes such as HP NonStop [4] and IBM zSeries [20]. However, lockstep in general-purpose execution encounters significant roadblocks in future process technologies. First, individual cores are likely to operate in separate clock domains for dynamic frequency control, while execution must still match precisely in time despite asynchronous inputs and physical distances between the cores [5,12]. Second, increasing within-die device- and circuit-level variability [6] leads to deviations from precise lockstep because, even in the absence of errors, cores will no longer have identical timing properties or execution resources. Third, lockstep requires precise determinism and identical initialization across all processor components, including in units that do not affect architecturally-correct execution (e.g., branch predictors [15]). As a result, redun-

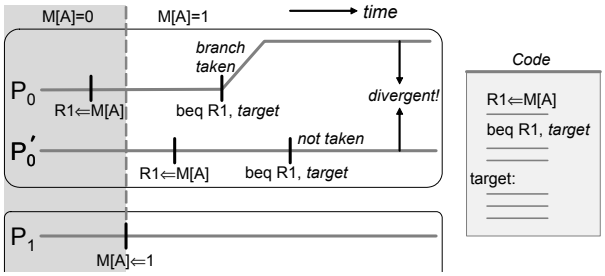

Figure 1. Input incoherence: redundant cores P_0 and P_0' observe different values for memory location M[A] from an intervening store.

dant execution models that avoid lockstep are highly desirable.

Multithreading. Recent proposals investigate using independent redundant threads within a simultaneous multithreaded (SMT) core [17,23] or across cores in a CMP [9,14,22]. Unlike lockstep, the threads execute independently and are bound by architectural requirements rather than microarchitectural timing constraints. Threads synchronize as outputs from the core (e.g., store values or register updates) are compared but remain coupled within a short distance to limit the storage needed for input replication and output comparison.

2.3. Input Incoherence

Multithreading introduces a problem for redundant execution because the threads independently execute and issue redundant memory requests. When executing shared-memory parallel programs, the threads can observe different values for the same dynamic load—which we term *input incoherence*—due to data races. Figure 1 illustrates this situation: these races arise between one execution's read of a cache block and the redundant partner's corresponding read. Writes from competing cores will cause input incoherence. This occurs in ordinary code such as spin-lock routines.

To avoid input incoherence, several prior proposals [9,14,17,23] enforce *strict input replication* across the redundant threads, where a leading execution defines the load values observed by both executions. Strict input replication can be achieved by either locking cache blocks or recording load values.

The active load address buffer (ALAB) [17] tracks cache blocks loaded by the leading thread and prevents their replacement until the trailing thread retires its corresponding load. The ALAB adds multiported storage arrays to track accessed cache blocks, logic to defer invalidations and replacements, and deadlock detection and retry mechanisms. The ALAB must be accessed on each load and external coherence request and requires significant changes to the out-of-order core's memory interface and pipeline control logic.

The LVQ is a FIFO structure, originally proposed as a simpler alternative to the ALAB, that records load values in program order from a leading execution and replays them for the trailing execution [17]. The LVQ requires modifications to the existing, heavily-optimized processor/cache interface. The trailing thread must bypass the cache and store buffer interface in favor of the LVQ, which adds bypass paths on the load critical path. Furthermore, the trailing thread only reads values in program order, which is a major policy change in front-end and out-of-order scheduling logic. The alternative—an out-of-order issue LVQ—eliminates the scheduling restriction, but has a similar complexity and area overhead as a multiported store buffer [14]. Finally, the LVQ also reduces error coverage of memory operations: there is no way to verify that load bypassing and forwarding completed correctly because the trailing execution relies upon leading thread load values.

Alternatively, in *relaxed input replication*, redundant threads independently send load requests to caches and store buffers, as in a non-redundant design. This avoids the added complexity of strict input replication and also provides detection for soft errors in load forwarding and bypass logic. However, this means that redundant executions are susceptible to input incoherence.

There are two general methods for tolerating input incoherence in relaxed input replication: robust forward recovery and rollback recovery. Prior work entrusts a robust checker to resolve input incoherence. For example, DIVA checkers with dedicated caches [8] and slipstreamed re-execution [25] both allow the leading thread's load values to differ from the trailing threads'. However, these proposals do not address the possibility of data races in shared-memory multiprocessors and require complex additions to support robust redundant execution. Alternatively, the naive rollback solution—simply retrying upon error detection—uses existing hardware support, but offers no forward progress guarantee. Because incoherent cache state or races may persist in the memory system, the same incoherent situation can occur again during re-execution. The Reunion execution model addresses this problem.

2.4. Output Comparison

The sphere of replication's boundary determines where outputs are compared. Two main choices have been studied in CMPs: (1) comparing outputs before

the architectural register file (ARF), and (2) comparing before the L1 cache [9,14]. In both cases, stores and uncached load addresses require comparison. For detection before the ARF, each instruction result must also be compared. We limit our study to systems with comparison before the ARF because existing precise exception support can then be used to recover before outputs become visible to other processors.

Output comparison impacts retirement. Serializing instructions, such as traps, memory barriers, and non-idempotent instructions, stall further execution until the serializing instruction has been compared. Both executions must complete and compare the serializing instruction before continuing.

Comparison bandwidth is another design factor in superscalar processors because multiple instructions may need comparison each cycle. Prior work proposes techniques to reduce bandwidth requirements. Gomaa et al. compare only instructions that end dependence chains in a lossless detection scheme [9]. They report bandwidth savings of roughly twenty percent over directly comparing each instruction result.

Smolens et al. [21] propose compressing architectural state updates into a signature called a fingerprint. Fingerprints lower comparison bandwidth by orders of magnitude with a negligible loss in error coverage. We extend this work to include two-stage compression that can match the retirement bandwidth of a wide superscalar (where the amount of data retired per cycle is larger than feasible hash circuits can consume).

3. Reunion Execution Model

This section presents a formal set of requirements for the Reunion execution model. The requirements provide redundant execution and relaxed input replication and allows reasoning about correctness independent of implementation. Figure 2 illustrates the concepts in this section.

3.1. System Definition

Definition 1 (Logical processor pair). A logical processor pair consists of two processor cores that execute the same instruction stream. To provide a single output from the sphere of replication, the logical processor pair presents itself as a single entity to the system.

We differentiate the two cores as follows:

Definition 2 (Vocal and mute cores). Each logical processor pair consists of one vocal and one mute core. The vocal core exposes updated values to the system and strictly abides by the coherence and memory consistency requirements specified by the baseline system. The mute core never exposes updates to the system.

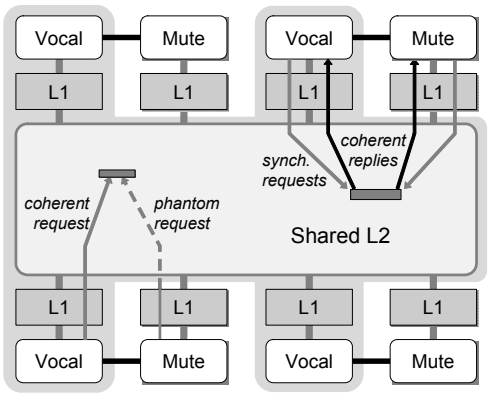

Figure 2. The Reunion architecture.

Vocal and mute cores use their existing private cache hierarchies and on-chip coherence protocol as in a non-redundant design. Definition 2 permits the mute core to write values into its private cache hierarchy, provided these values are not communicated to other caches or main memory.

Reunion uses redundant execution to detect and recover from soft errors that occur in program execution. We formally define safe execution as follows:

Definition 3 (Safe execution). Program execution is "safe" if and only if (1) all updates to architecturally-defined state are free of soft error effects, (2) all memory accesses are coherent with the global memory image, and (3) the execution abides by the baseline memory consistency model. Execution that is not safe is deemed "unsafe".

The state that results from safe execution is:

Definition 4 (Safe state). The architectural state defined by the vocal core at a specific point in time is considered "safe state" if and only if it is free of soft errors; otherwise, the architectural state is deemed "unsafe state".

3.2. Execution Model

Definition 2 requires that only the vocal abide by coherence and consistency requirements. Ideally, the mute core always loads coherent data values. However, precisely tracking coherent state for both vocal and mute would be prohibitively complex (e.g., the coherence protocol would have to track two owners for exclusive/modified blocks).

Instead, Reunion maintains coherence for cache blocks in vocal caches, while allowing incoherence in mute caches. The mechanism for reading cache blocks into the mute cache hierarchy is the *phantom request*, a non-coherent memory request:

0-7695-2732-9/06 $20.00

Definition 5 (Phantom request). A phantom request returns a value for the requested block without changing coherence state in the memory system.

The phantom request does not guarantee that the mute core will be coherent with the vocal, potentially leading to input incoherence within a logical processor pair:

Definition 6 (Input incoherence). Input incoherence results when the same dynamic load on vocal and mute cores returns different values.

Reunion requires vocal and mute to compare execution results as follows:

Definition 7 (Output comparison). Vocal and mute cores must compare all prior execution results before a value becomes visible to other logical processor pairs.

Lemma 1. In the absence of soft errors, input incoherence cannot result in unsafe execution.

Proof: If no soft error occurred during program execution, condition (1) of safe execution (Definition 3) is satisfied. If input incoherence occurred, the register updates and memory writes on the vocal still satisfy conditions (2) and (3). Therefore, safe execution results.

Only undetected soft errors can result in unsafe state. Both input incoherence and soft errors can lead to divergent execution that must be detected and corrected. However, Lemma 1 proves that input incoherence alone cannot result in unsafe state.

3.3. Recovery

Definition 8 (Rollback recovery). When output comparison matches, the vocal's architectural state defines a new safe state that reflects updates from the compared instruction results; otherwise, rollback recovery restores architectural state to prior safe state.

Because only the vocal core's architectural state defines new safe state, Reunion requires a mechanism to initialize the mute core's architectural registers to match the vocal core.

Definition 9 (Mute register initialization). The vocal and mute cores provide a mechanism to initialize the mute core's architectural register file with values identical to the vocal's.

In the presence of input incoherence, naïve retry cannot guarantee forward progress because the condition causing input incoherence can persist. Incoherent cache blocks in the mute's hierarchy can cause input incoherence until replaced by coherent values. Reunion addresses this problem with the *synchronizing request*:

Definition 10 (Synchronizing request). The synchronizing request returns a single coherent value to both cores in the logical processor pair.

We combine mute register initialization and the synchronizing request to create the re-execution protocol and then prove that the protocol guarantees forward progress following rollback recovery.

Definition 11 (Re-execution protocol). After rollback recovery, the mute architectural register file is initialized to the values from the vocal. The logical processor pair then executes subsequent instructions non-speculatively (single-step), up to and including the first load or atomic memory operation. This operation is issued by both cores using the synchronizing request. After successful output comparison following this instruction, the logical pair resumes normal execution.

Lemma 2. (Forward Progress). The Reunion re-execution protocol always results in forward progress.

Proof: Rollback recovery is triggered either by a soft error, which does not persist, or input incoherence, which may persist. In the first case, re-execution eliminates the error and results in successful output comparison. In the second case, the mute register initialization and synchronizing request guarantee safe execution and safe state to the first load.

An implementation must provide the required behaviors of the execution model, but the system designer has latitude to optimize. In Section 4, a fast re-execution protocol implementation handles common case re-execution, while a slower version implements the rarely needed register file copy.

4. Reunion Microarchitecture

In this section, we first describe our baseline CMP and processor microarchitecture. We then discuss the changes required to implement the Reunion execution model in a shared cache controller and processor core.

4.1. Baseline CMP

Cache Hierarchy. We assume a baseline CMP with caches similar to Piranha [3]. A shared cache backs multiple write-back L1 caches private to each processor core. The shared cache controller accepts memory requests from all cores, coordinates on-chip coherence for blocks in private caches, and initiates off-chip transactions. The Reunion execution model can also be implemented at a snoopy cache interface for microarchitectures with private caches, such as Montecito [11].

Processor Microarchitecture. We assume the simplified out-of-order processor pipeline illustrated in Figure 3(a). Instructions are fetched and decoded in-order, then issued, executed, and written back out-of-order. In-order retirement stages inspect instructions

for branch mis-speculation and exceptions, and write instruction results to the architectural register file, as in Pentium-M [18]. Stores initially occupy a speculative region of the store buffer. At retirement, the stores transition to a non-speculative region of the store buffer and drain to the L1 cache.

This paper assumes single-threaded processor cores. Reunion can benefit from the efficient use of otherwise idle resources in SMT; however, cores must run only vocal or mute threads to prevent vocal contexts from consuming incoherent cache blocks.

4.2. Shared Cache Controller

The shared cache controller is responsible for implementing the vocal and mute semantics, phantom requests, and synchronizing requests. As in non-redundant designs, the shared cache controller maintains coherence state (e.g., ownership and sharers lists) for all vocal cores.

Because coherence is not necessary in mute caches, sharers lists never include mute caches and mute caches can never become exclusive or modified block owners. The coherence protocol behaves as if mute cores were absent from the system. To prevent values generated by mutes from being exposed to the system, the shared cache controller ignores all eviction and writeback requests originating from mute cores.

Phantom requests. All non-synchronizing requests from the mute to the shared cache controller are transformed into phantom requests. The phantom request produces a reply, although the value need not be coherent, or even valid. Phantom replies grant write permission within the mute hierarchy.

The phantom request allows several "strengths", depending on how diligently it searches for coherent data. The weakest phantom request strength, *null*, returns arbitrary data on any request (i.e., any L1 miss). While trivial to implement, *null* has severe performance implications. A *shared* phantom request checks for hits in the shared cache and only returns arbitrary values on misses. Finally, the *global* phantom request achieves the best approximation of coherence. This request not only checks the shared cache, but also private vocal caches and issues read requests to main memory for off-chip misses. In terms of complexity, this is a small departure from existing read requests. Unless otherwise noted, this paper assumes global phantom requests.

Synchronizing requests. The shared cache controller enforces coherence between vocal and mute cores only on synchronizing requests. Synchronizing requests flush the block from private caches (returning the vocal's copy to the shared cache, while discarding the mute's). When both requests have been received at L2, the shared cache controller initiates a coherent write transaction for the cache block on behalf of the pair. This obtains sufficient permission to complete instructions with both load and store semantics. After obtaining the coherent value, the shared cache controller atomically replies to both the vocal and mute cores. The synchronizing request dominates recovery latency and is comparable to a shared cache hit.

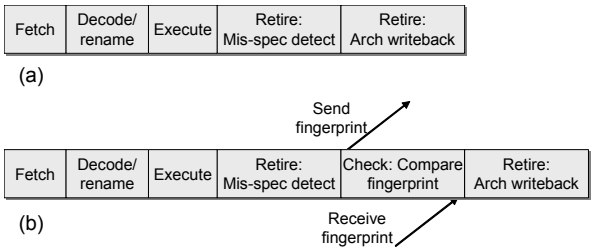

Figure 3. (a) Baseline pipeline and (b) a pipeline with fingerprint checks before retirement.

4.3. Processor Pipeline

We now describe the processor pipeline changes for the Reunion execution model, output comparison and recovery.

Safe state. The vocal processor core maintains safe state in the ARF, non-speculative store buffer and memory. Safe state can always be reached by the vocal by (1) retiring all instructions that have completed output comparison without error to the architectural register file and the non-speculative store buffer and (2) flushing all uncompared instructions from the pipeline (e.g., precise exception rollback).

Output comparison. Instruction outputs must be compared before retiring to architectural state. The key addition is an in-order retirement stage called *check*. *Check* first generates a fingerprint—a hash of instruction results—from the entering instructions [21]. *Check* then compares its fingerprint with the partner core's fingerprint to detect differences. A matching fingerprint comparison retires the instruction and writes the instruction results to safe state in the architectural register file. A mismatch invokes recovery. Instructions cannot enter *check* speculatively; they must be guaranteed to retire if the instruction results match.

Logically, the fingerprint captures all register updates, branch targets, store addresses, and store values. The number of instructions summarized by each fingerprint is a design parameter called the *fingerprint interval*; longer comparison intervals need proportionally less comparison bandwidth. At the end of each fingerprint interval, each core sends its fingerprint to the

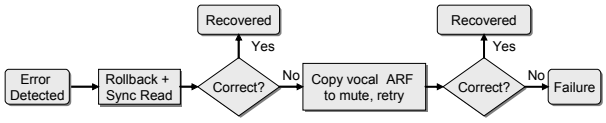

Figure 4. The re-execution protocol.

partner core for comparison. We find empirically that the performance difference between intervals of one and fifty instructions is insignificant in our workloads, despite increased resource occupancy, because useful computation continues to the end of the interval.

We combine the time required to generate, transfer, and compare the fingerprint into a parameter called the *comparison latency*. Because the vocal and mute cores "swap" fingerprints, the comparison latency is the one-way latency between cores. This latency overlaps with useful computation, at the cost of additional resource occupancy. The observed comparison latency, however, may be extended because the two cores are only loosely coupled in time. While the vocal and mute execute the same program, their relative progress may drift slightly in time, due to contention accessing shared resources (e.g., the L2 cache) and different private cache contents.

Fingerprint generation. In wide superscalar processors, fingerprint generation bandwidth is a concern. The total state updates per cycle can exceed 256 bits of state, which exceeds what parallel CRC circuits can consume in a single clock. We solve this problem by leveraging a two-stage compression technique used in circuit testing [7], which places space-compressing parity trees before a time-compressing circuit, such as a parallel CRC [2] or multiple-input shift register (MISR) [7]. Parity trees reduce the raw M bits of state down to N bits of compressed state in a single clock cycle, which then feed the time-compressing circuit in the next cycle. Parity trees necessarily reduce the fingerprint's error detection coverage. Assuming all combinations of bit flips are equally likely, it can be shown that this two-stage technique doubles the aliasing probability. Therefore, the probability of aliasing is at most $2^{-(N-1)}$, where N is the width of a CRC circuit. The analysis in [21] shows that a 16-bit CRC already exceeds industry system error coverage goals by an order of magnitude.

Re-execution. Upon detection of differences between the vocal and mute, the logical processor pair starts the re-execution protocol illustrated in Figure 4. To optimize for the common case, the protocol is divided into two phases. The first handles detected soft errors and detected input incoherence errors. The second phase addresses the extremely rare case where results of undetected input incoherence retire to architectural registers.

Both vocal and mute cores invoke rollback-recovery using precise exception support and, in the common case, restore to identical safe states in their architectural register files. Both cores then non-speculatively single-step execution up to the first memory read. Each core then issues a synchronizing memory request—eliminating input incoherence for the requested cache block—and compares a fingerprint for all instructions in the interval. Following comparison, the re-execution protocol has made forward progress by at least one instruction. The cores then continue normal speculative, out-of-order execution.

If the first phase fails output comparison, the second phase starts. The vocal copies its architectural register file to the mute and the pair proceeds with re-execution, as in the first phase. Because the vocal core always maintains safe state in the absence of soft errors, this will correctly recover from all incoherence errors. If the cause was a soft error missed by fingerprint aliasing, the protocol cannot recover safe state and therefore must trigger a failure (e.g., detected, uncorrectable error interrupt). The re-execution protocol can be implemented in microcode.

External interrupts. External interrupts must be scheduled and handled at the same point in program execution on both cores. Fingerprint comparison provides a mechanism for synchronizing the two cores on a single instruction. Reunion handles external interrupts by replicating the request to both the vocal and mute cores. The vocal core chooses a fingerprint interval at which to service the interrupt. Both processors service the interrupt after comparing and retiring the preceding instructions.

Hardware cost. Fingerprint comparison requires queues to store outstanding fingerprints, a channel to send fingerprints to the partner core, hash circuitry, and a comparator. The fingerprint queues can be sized to balance latency, area, and power. The *check* stage delays writing results into the architectural register file. The results can be stored in a circular buffer during the *check* stage or read again at retirement.

4.4. Serializing Check Overhead

Instructions with serializing semantics—such as traps, memory barriers, atomic memory operations, and non-idempotent memory accesses—impose a performance penalty in all redundant execution microarchitectures. Serializing instructions must stall pipeline retirement for a full comparison latency, because (1) all older instructions must be compared and retired before

Table 1. Simulated baseline CMP parameters.

Processor Cores	4 logical processors, UltraSPARC III ISA 4 GHz 12-stage pipeline; out-of-order 4-wide dispatch / retirement 256-entry RUU; 64-entry store buffer
L1 Cache	64KB split I/D, 2-way, 2-cycle load to use, 2 rd, 1 wr ports, 64-byte lines, 32 MSHRs
Shared L2 Cache	16MB unified, 4 banks, 8-way, 35-cycle hit latency, 64-byte lines, crossbar to L1s, 64 MSHRs
ITLB	128 entry, 2-way; 8K page
DTLB	512 entry 2-way; 8K page
Memory	3 GB, 60 ns access latency, 64 banks

Table 2. Application parameters.

Commercial Workloads	
DB2 OLTP	100 warehouses (10GB), 64 clients, 450MB BP
Oracle OLTP	100 warehouses (10GB), 16 clients, 1.4GB SGA
DB2 DSS	Qry 1 (scan); Qry 2 (join); Qry 17 (balanced) 100 warehouses (10GB), 450MB BP
Apache Web	16K connections, fastCGI, worker thread model
Zeus Web	16K connections, fastCGI
Scientific Workloads	
em3d	768K nodes, degree 2, span 5, 15% remote
moldyn	19,652 molecules, boxsize 17, 2.56M max iters.
ocean	258x258 grid, 9600s relax, 20K res., errtol 1e-7
sparse	4096x4096 matrix

the serializing instruction can execute and (2) no younger instructions can execute until the serializing instruction retires.

Upon encountering a serializing instruction, the fingerprint interval immediately ends to allow older instructions to retire. The fingerprint is updated to include state that must be checked before executing the serializing instruction (e.g., uncacheable load addresses). Once the older instructions retire, the serializing instruction completes its execution and check. This comparison exposes timing differences between the cores (due to loosely-coupled execution) and the entire comparison latency. Normal execution continues once the serializing instruction retires.

5. Evaluation

We evaluate Reunion using FLEXUS, which provides cycle-accurate, full-system simulation of a chip multiprocessor [24]. FLEXUS extends Virtutech Simics with cycle-accurate models of an out-of-order processor and memory system.

We simulate a CMP with four logical processors: four cores for non-redundant models and eight cores for redundant models. We assume on-chip cache bandwidth scales in proportion with the number of cores. Our CMP model uses a cache hierarchy derived from Piranha [3]. For Reunion, the vocal and mute L1 cache tags and data are independently modeled. We list system parameters in Table 1.

Table 2 lists our commercial and scientific application suite. All workloads run on Solaris 8. We include the TPC-C v3.0 OLTP workload on two commercial database management systems, *IBM DB2 v8 Enterprise Server Edition*, and *Oracle 10g Enterprise Database Server*. We tune the database to maximize performance of our non-redundant system model.

We select three representative queries from the TPC-H decision support system (DSS) workload, showing scan-dominated, join-dominated, and mixed behavior. We evaluate web server performance with the SPECweb99 benchmark on *Apache HTTP Server v2.0* and *Zeus Web Server v4.3*. We report the server performance of web servers saturated by separate clients over a high-bandwidth link. We also include four parallel shared-memory scientific applications that exhibit a range of memory access patterns.

We use a sampling approach that draws many brief measurements over 10 to 30 seconds of simulated time for OLTP and web applications, the complete query execution for DSS, and a single iteration for scientific applications. We target 95% confidence intervals of ±5% error on change in performance using matched-pair comparison [24]. We launch measurements from checkpoints with warmed caches and branch predictors, then run for 100,000 cycles to warm pipeline and queue state prior to 50,000 cycles of measurement. We collect the aggregate user instructions committed per cycle as our performance metric, which is proportional to overall system throughput. We compare fingerprints on every instruction. We do not inject soft errors; however, input incoherence events, output comparison, and recovery are modeled in detail.

5.1. Baseline Performance

We evaluate the baseline performance of redundant execution in a CMP for a representative system using strict input replication ("*Strict*") and Reunion.

Strict models a system with strict input replication, fingerprint comparison across cores for error detection, and recovery within the ROB (as described for Reunion in Section 4.3). *Strict* serves as an oracle performance model for all strict input replication designs with recovery. It imposes no performance penalty for input replication (e.g., lockstepped processor cores or an LVQ with no resource hazards). However, we model the penalties from buffering instructions dur-

0-7695-2732-9/06 $20.00

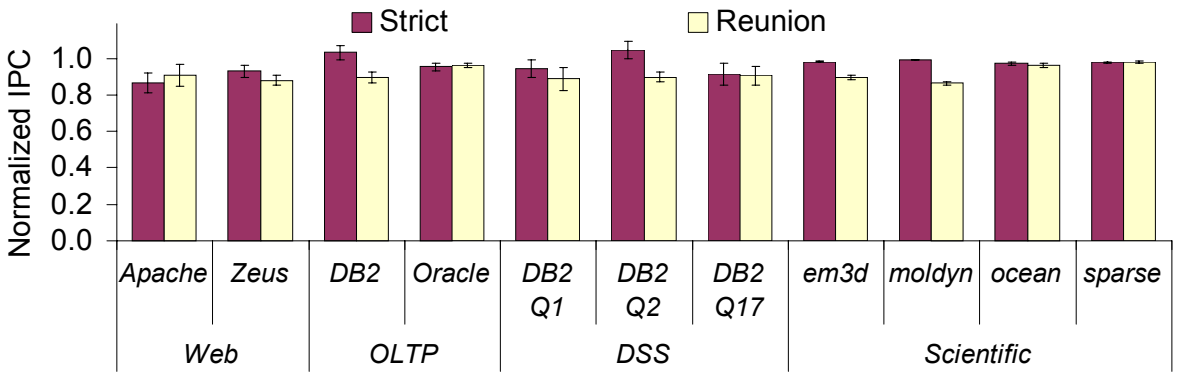

Figure 5. Baseline performance of redundant execution with strict input replication and Reunion normalized to a non-redundant baseline, with a 10-cycle comparison latency.

ing check. The Reunion model demonstrates the performance of relaxed input replication and fingerprint comparison and recovery. To support recovery within the speculative window, both systems check instruction results before irrevocably retiring them to the architectural register file and non-speculative store buffer.

Figure 5 shows the baseline performance of both models normalized to the performance of a non-redundant baseline CMP, with a ten-cycle comparison latency between cores. As compared to the non-redundant baseline, the strict model has a 5% and 2% average performance penalty for commercial and scientific workloads, respectively, while Reunion shows 10% and 8% average respective performance penalties. The low performance overhead of Reunion demonstrates that relaxed input replication is a viable redundant execution model. In the following sections, we explore the performance of these execution models in more detail.

5.2. Checking Overhead

We first examine the performance of *Strict* to understand the performance penalties of checking redundant executions across cores in a CMP. First, serializing instructions cause the entire pipeline to stall for the check because no further instructions can execute until these instructions complete. The check fundamentally extends this stall penalty: as the comparison latency increases, the retirement stalls must also increase. Second, pipeline occupancy increases from instructions in check occupying additional ROB capacity in the speculative window. For workloads that benefit from large instruction windows, this decreases opportunities to exploit memory-level parallelism (MLP) or perform speculative execution.

Figure 6(a) shows the average performance impact from checking in *Strict* for each workload class over a range of on-chip comparison latencies, normalized to a non-redundant baseline. At a zero-cycle comparison latency, the workloads do not show a statistically significant performance difference from non-redundant execution. The performance penalty increases linearly with increasing comparison latency. Both commercial and scientific workloads exhibit similar sensitivity to the comparison latency; however, the mechanisms are different.

In commercial workloads, the dominant performance effect comes from frequent serializing instructions. With increased comparison intervals, the number of these events remains constant, but the stall penalty increases. At forty cycles, the average performance penalty from checking is 17%.

In contrast, the scientific workloads suffer from increased reorder buffer occupancy because they can saturate this resource, which decreases MLP. At a comparison latency of forty cycles, the average performance penalty is 11%. While space constraints limit more detailed analysis, larger speculation windows (e.g., thousands of instructions, as in checkpointing architectures [1]) completely eliminate the resource occupancy bottleneck, but cannot relieve stalls from serializing instructions.

5.3. Reunion Performance

We first evaluate the performance penalty of relaxed input replication under Reunion, then explore Reunion's sensitivity to comparison latencies. Unlike the strict input replication model, vocal and mute execution in Reunion is only loosely coupled across the cores. For non-serializing instructions, these differences can be absorbed by buffering in the check stage. However, serializing instructions expose the loose coupling because neither core can make further progress until the slower core arrives at and compares the instruction. This introduces additional retirement stalls

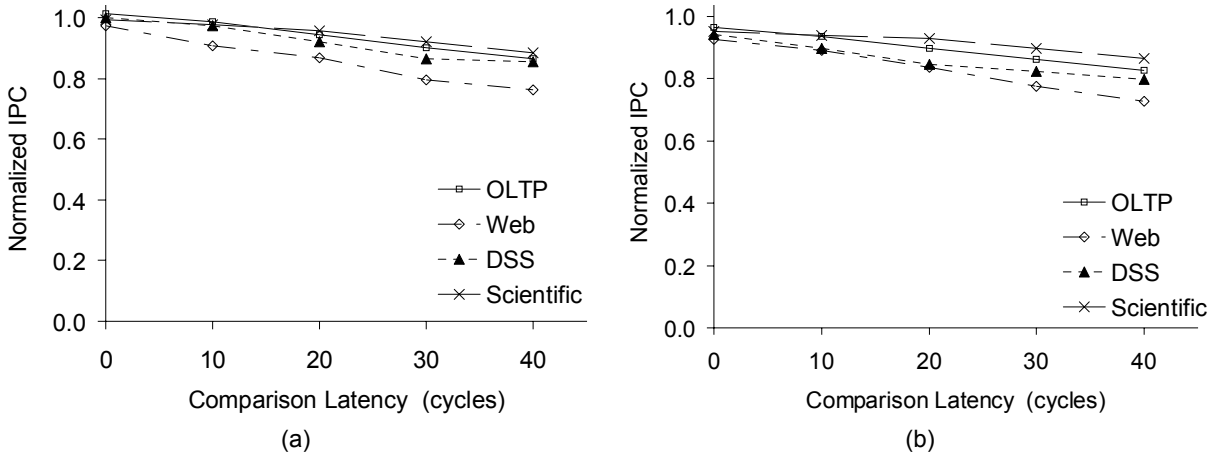

Figure 6. Performance sensitivity to the comparison interval of (a) strict input replication and (b) Reunion.

that affect both cores, on top of the comparison overheads discussed above. Figure 6(b) shows the performance of Reunion for a range of comparison latencies. Reunion's performance is determined by checking overheads, loose coupling, and input incoherence.

The first observation from Figure 6(b) is that, unlike *Strict*, Reunion has a performance penalty from loose coupling and relaxed input replication at a zero-cycle comparison latency. For commercial workloads, the serializing events expose the loose coupling, because one core must wait for its partner to catch up before comparing fingerprints. For scientific workloads, contention at the shared cache increases the effective memory latency, decreasing performance. This result shows that the baseline performance penalty of Reunion's relaxed input replication is small—on average, 5% and 6% for commercial and scientific workloads, respectively.

The second observation in Figure 6(b) is that at non-zero comparison latencies, performance converges towards the limits set by the strict input replication model. As the comparison latency grows, the comparison overhead and resource occupancies dominate the performance, because more time is spent waiting on the comparison than resolving loose coupling delays. At a forty-cycle comparison latency, the average performance penalty is 22% and 13% for commercial and scientific workloads, respectively, which closely follows the *Strict* model's trend. This result shows that the primary performance impact with larger comparison latencies comes from fundamental limits of checking and recovery, instead of relaxed input replication.

5.4. Input Incoherence

We now provide empirical evidence to demonstrate that input incoherence events in Reunion are uncommon. Table 3 shows the frequency of input incoherence events per million retired instructions in Reunion for all three phantom request strengths, with a ten-cycle comparison latency. As a point of comparison, we juxtapose the input incoherence events with another common system event with a comparable performance penalty: the translation lookaside buffer (TLB) miss.

We first consider Reunion with *global* phantom requests. Recall that *global* phantom requests initiate on- and off-chip non-coherent reads on behalf of the mute. As shown, our workloads encounter input incoherence events infrequently with *global* phantom requests, while data and instruction TLB misses generally occur orders of magnitude more frequently. From this data we conclude that input incoherence events are, in fact, uncommon in our workloads. Furthermore, even with relaxed input replication, the penalty of recovery is overshadowed by other system events.

Next, we investigate whether choices for weaker phantom requests from Section 4.2 are effective. The data for *shared* and *null* phantom requests in Table 3 show that input incoherence events are three to four

Table 3. Input incoherence events for each phantom request strength, TLB miss frequency.

	Per 1M instructions			
	Input Incoherence			
Workload	**Global**	**Shared**	**Null**	**TLB Misses**
Apache	0.9	3,818	8,620	1,973
Zeus	0.2	1,818	5,456	1,654
DB2 OLTP	0.7	5,340	16,197	2,492
Oracle OLTP	0.6	4,578	17,140	3,297
DB2 DSS Q1	21.1	1,909	4,004	206
DB2 DSS Q2	0.7	4,852	7,991	1,040
DB2 DSS Q17	1.5	4,863	10,466	1,089
Avg. Scientific	0.4	17,406	22,607	239

0-7695-2732-9/06 $20.00

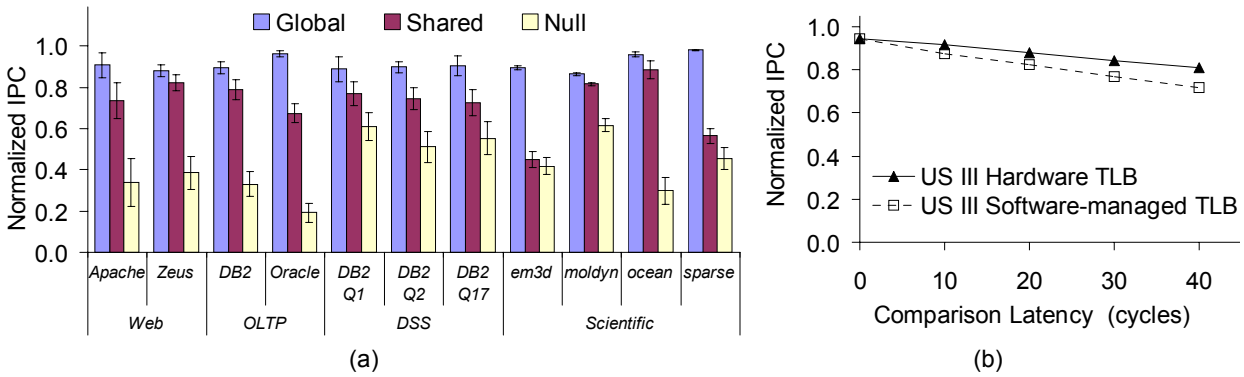

Figure 7. (a) Reunion performance with different phantom request strengths and (b) Reunion average performance for commercial workloads with hardware and software-managed TLBs.

orders of magnitude more frequent than with *global* phantom request strengths and in both cases are more frequent than TLB misses. These high frequencies indicate that recovery from input incoherence can become a bottleneck with weaker phantom requests.

Figure 7(a) compares the performance of the three phantom request strengths with a 10-cycle comparison latency, normalized to the non-redundant baseline. Both *shared* and *null* phantom requests incur a severe performance impact from frequent recoveries. *Shared* phantom requests capture most mute L1 misses because of the high shared-cache hit rate. One notable exception is *em3d*, whose working set exceeds the shared cache and therefore frequently reads arbitrary data instead of initiating off-chip reads. The *null* phantom request policy has a severe performance impact for all workloads because each L1 read miss is followed by input incoherence rollbacks.

The analysis of input incoherence in this section shows that *global* phantom requests are effective in reducing input incoherence events to negligible levels in our workloads. Furthermore, the weaker phantom request strengths increase the frequency of input incoherence to levels that cause a severe performance impact.

5.5. Serialization Overhead

Finally, we identify the importance of architecturally-defined serializing instructions to redundant execution performance. In the system we evaluate, serializing instructions are traps, memory barriers, and non-idempotent memory requests. Many of these events are inherent in the workloads, such as memory barriers needed to protect critical sections, while others, such as system traps, are specified by the instruction set architecture.

The results presented up to this point in the paper have eliminated the dominant source of system-specific traps in our baseline UltraSPARC III architecture: the software-managed TLB miss handler. In commercial workloads, the "fast TLB miss handler" is invoked frequently (see Table 3), due to their large instruction and data footprints. The handler function includes two traps, for entry and exit, and executes three non-idempotent memory requests to the memory management unit (MMU).

Figure 7(b) contrasts the average performance of commercial workloads with a hardware-managed TLB model and the architecturally-defined UltraSPARC III software-managed TLB handler. As the comparison interval increases, the contribution of the serializing checks is readily apparent—increasing the performance impact to 28% at a forty-cycle comparison latency. While this result is from Reunion, a comparable impact also occurs with strict input replication.

Strong memory consistency models can also affect checking performance. In contrast with Sun TSO, the Sequential Consistency (SC) memory consistency model places memory barrier semantics on every store (5-10% of the dynamic instructions in our workloads). Hence, every store serializes retirement. We observe an average performance loss of over 60% at 40 cycles due to store serialization with SC.

The results in this section underscore the importance of considering serializing instructions, especially architecture-specific ones, in the performance of redundant execution microarchitectures.

6. Conclusion

Designs for redundant execution in multiprocessors must address input replication and output comparison. Strict input replication requires significant changes to the pipeline. We observe that the input incoherence targeted by strict input replication is infrequent. We propose the Reunion execution model, which uses relaxed input replication, backed by light-

weight error recovery within the processor's speculative window. The model preserves the existing memory system, including the coherence protocol and memory consistency model. We show that the overhead of relaxed input replication in Reunion is only 5% and 6% for commercial and scientific workloads, respectively.

Acknowledgements

We thank Stephen Somogyi for his help with workload checkpoints, members of the SimFlex research group at Carnegie Mellon and the anonymous reviewers for their helpful feedback. This work is supported by NSF CAREER award CCF-0347568, NSF award ACI-0325802, Sloan and DoD/NDSEG fellowships, the Center for Circuit and System Solutions (C2S2), MARCO, CyLab, and by grants and equipment from Intel Corporation.

References

[1] H. Akkary et al. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proc. of the 36th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, Dec 2003.

[2] G. Albertengo and R. Sisto. Parallel CRC generation. *IEEE Micro*, 10(5):63–71, Oct. 1990.

[3] L. Barroso et al. Piranha: A scalable architecture base on single-chip multiprocessing. In *Proc. of the 27th Annual Intl. Symp. on Computer Architecture*, June 2000.

[4] J. Bartlett et al. Fault tolerance in tandem computer systems. Technical Report TR-86.7, HP Labs, 1986.

[5] D. Bernick et al. Nonstop advanced architecture. In *Proc. Intl. Conf. Dependable Systems and Networks*, Jun. 2005.

[6] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov-Dec 2005.

[7] K. Chakabarty and J. P. Hayes. Test response compaction using multiplexed parity trees. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 15(11), Nov 1996.

[8] S. Chatterjee et al. Efficient checker processor design. In *Proc. of the 33rd Annual IEEE/ACM Intl. Symp. on Microarchitecture*, Dec 2000, 87–97.

[9] M. Gomaa et al. Transient-fault recovery for chip multiprocessors. In *Proc. of the 30th Annual Intl. Symp. on Computer Architecture*, Jun. 2003.

[10] 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, Jul. 1995.

[11] C. McNairy and R. Bhatia. Montecito: A dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(2), Mar. 2005.

[12] P. J. Meaney et al. IBM z990 soft error detection and recovery. *IEEE Trans. device and materials reliability*, 5(3):419–427, Sept. 2005.

[13] S. S. Mukherjee et al. The soft error problem: An architectural perspective. In *Proc. of the Eleventh IEEE Symp. on High-Performance Computer Architecture*, Feb 2005.

[14] S. S. Mukherjee et al. Detailed design and evaluation of redundant multithreading alternatives. In *Proc. of the 29th Annual Intl. Symp. on Computer Architecture*, May 2002.

[15] S. S. Mukherjee et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proc. of the 36th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, Dec 2003.

[16] K. Olukotun et al. The case for a single-chip multiprocessor. In *Proc. of the Seventh Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[17] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proc. of the 27th Annual Intl. Symp. on Computer Architecture*, Jun. 2000.

[18] J. P. Shen and M. H. Lipasti. *Modern processor design: fundamentals of superscalar processors*. McGraw Hill, 2005.

[19] P. Shivakumar et al. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. of the 2002 Intl. Conf. on Dependable Systems and Networks*, Jun. 2002.

[20] T. Slegel et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, Mar-Apr. 1999.

[21] J. C. Smolens et al. Fingerprinting: Bounding soft-error detection latency and bandwidth. In *Proc. of the Eleventh Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004, 224–234.

[22] K. Sundaramoorthy et al. Slipstream processors: improving both performance and fault tolerance. In *Proc. of the Ninth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[23] T. N. Vijaykumar et al. Transient-fault recovery using simultaneous multithreading. In *Proc. of the 29th Annual Intl. Symp. on Computer Architecture*, May 2002.

[24] T. F. Wenisch et al. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, Jul.-Aug. 2006.

[25] H. Zhou. A case for fault tolerance and performance enhancement using chip multi-processors. In *IEEE Computer Architecture Letters*, Jan. 2006.