Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery

Joydeep Ray, James C. Hoe and Babak Falsafi
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
{jray, jhoe, babak}@ece.cmu.edu

CALCM Technical Report, #2002-002
June, 2001

Abstract

Diminutive devices and high clock frequency of future microprocessor generations are causing increased concerns for transient soft failures in hardware, necessitating fault detection and recovery mechanisms even in commodity processors. In this paper, we propose a fault-tolerant extension for modern superscalar out-of-order datapath. We argue that a single processor that can selectively deliver fault-tolerance when required and can otherwise revert to full performance will be an important design point in the transitional phase when transient failure rates are just becoming unacceptable for some applications. This dual-functionality will enable both vendors and end-users to trade-off performance for fault-protection depending on applications. In the proposed extensions, error-detection is achieved by verifying the redundant results of dynamically replicated threads of executions, while the error-recovery scheme employs the branch-rewind mechanism to restart at a failed instruction. The proposed scheme can be supported by only modest additional hardware. Nevertheless, it delivers comparable performance and fault coverage as static fully-replicated solutions.

We study the performance impact of augmenting superscalar microarchitectures with fault tolerance capabilities. An analytical performance model is used in conjunction with a performance simulator. The simulation results of 11 SPEC95 and SPEC2000 benchmarks show that in the absence of faults, error detection causes a 2% to 45% reduction in throughput, which is comparable to the previously proposed schemes. In the presence of transient faults, the fast error recovery scheme contributes very little additional slowdown. The low-impact in performance and the low hardware overhead make our proposed scheme a viable extension to current superscalar processors for transient-fault tolerance.

keywords: *fault-tolerance, superscalar, transient fault, detection, recovery*

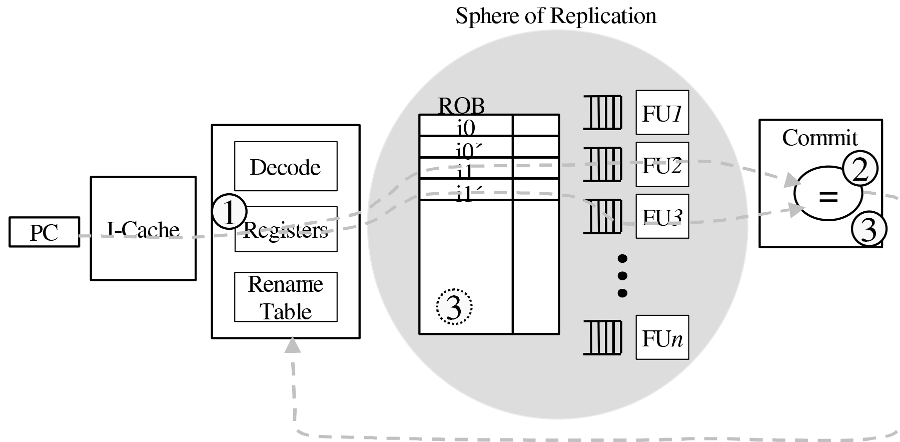

Figure 1. The three steps of fine-grain concurrent error detection and recovery on a superscalar processor.

1 Introduction

Following the current trends in transistor size, voltage and clock frequency, future microprocessors will become increasingly susceptible to transient hardware failures (a.k.a. single-event upsets (SEU) or soft errors) [3, 7]. Recently, researchers have proposed techniques to make use of inherent hardware redundancies of multithreaded and chip-multiprocessor architectures in concurrent error detection [14, 15, 19]. In this paper we present a transient-fault tolerant design that takes advantage of the redundancies found in superscalar processors. In addition to concurrent error detection, we propose to make use of pre-existing "branch-rewind" mechanisms for recovery. A fault-tolerant superscalar processor is in an important design space because a microprocessor that can deliver maximum single-thread performance at a given cost will continue to be the mainstay in commodity PC and embedded processor markets. Backed by tremendous economic momentum, a single design that can deliver maximum everyday-use performance and alternatively provide mission-critical reliability will have a profound impact on the affordability of reliable computing. On the one hand, high-reliability servers and mainframes could leverage the economy of scale of mass-market PC processors. While on the other hand, everyday PC users can have the option of selectively trading off performance for reliability depending on their customized usage and requirements.

The same mechanisms for speculative out-of-order superscalar execution, common in the current generation of microprocessors, can also be applied to *both* detection and recovery of transient faults. Figure 1 illustrates the actions of error detection and recovery in a superscalar datapath. (1) By adapting register renaming capabilities, instructions fetched from a single stream can be issued redundantly as two or more data-independent threads in the dynamic execution path. (2) For error detection, the redundantly computed results from multiple threads can be checked against each other prior to committing the effect of an instruction. (3) Any inconsistency between the redundant results triggers the branch-rewind mechanism to restart program execution from the failed instruction; the execution can be continued in a seamless, timely fashion if the error is non-recurring.

This paper explores the above fault-tolerant framework in depth and pays particular attention to the issues of transient-fault recovery. Earlier work on fault-tolerant superscalar design (such as [8]) concentrates on concurrent error detection of irrecoverable hard failures and does not consider the possibility of using branch-rewind as a recovery mechanism. The more recent efforts in the multithreaded/multiprocessor contexts also do not discuss the mechanisms for efficiently recovering from a transient fault and instead rely on coarse-grain checkpointing. Coarse-grain recovery schemes severely disrupt program progress and are only acceptable for non-interactive data-processing applications. Fine-grain recovery is a must for both real-time or interactive applications. Furthermore, in our proposed fault-tolerant superscalar design, once concurrent error detection is in place, recovery can be implemented at nearly no hardware or performance cost.

When the proposed fault-tolerant mechanism is in effect, some fraction of the original processor throughput is lost to redundant processing, but when protection is not needed, the modified datapath can still be returned to the performance of an optimally-tuned superscalar design because our extension requires only small deviations from a standard design. To better understand the performance cost of reliability, we have developed both a simple analytical model and a SimpleScalar-derived performance simulator. In most cases, the performance loss in a two-way redundant execution is much less than 50% compared to normal non-redundant executions on comparable hardware resources. For the 11 SPEC95 and SPEC2000 benchmarks we studied, the throughput (IPC) penalty due to two-way redundant execution ranges from 2% to 45% (32% average) on an 8-way superscalar datapath. This throughput penalty is inline with other reported results for concurrent error detection by redundant instruction processing. We have determined that the overall throughput remains unaffected by even a high frequency of faults because of the low cost of rewind-based recovery. In this paper, we also discuss the trade-offs and the range of applicability of different implementation decisions.

Paper Outline: The remainder of this paper is organized as follows. Section 2 provides additional background on the nature of transient faults and relates our effort to prior work in fault-tolerance (both hard and soft errors). Section 3 presents the details of our proposed transient fault detection and recovery mechanisms. Section 4 presents a simple analytical performance model for a fault-tolerant superscalar processor and applies it in an analysis. Section 5 presents the results of our simulation-based performance evaluation. Section 6 summarizes the contributions of this paper and discusses future directions of our efforts.

2 Background

2.1 Transient Hardware Faults

Transient faults have traditionally been associated with the corruption of stored data values. This phenomenon has been reported as early as 1954 in adverse operating conditions such as near nuclear bomb test sites and later in space applications [22, 12]. Since 1978, dense memory circuits, both DRAM and SRAM, have been known to be susceptible to soft errors caused by alpha-particles from IC packagings [11] and cosmic rays [21]. By definition, a hardware device can recover its full capability following a transient failure, but such failures are no less catastrophic for the correct execution of a program because a corrupted intermediate value, if not handled, can corrupt all subsequent computations. Over the years, measures to protect against soft errors in memory devices have evolved to include physical techniques in cell/gate design [1] and packaging materials [9], as well as error correction codes (ECC). Today, these techniques are commonplace even in commodity PC memories, but, except in extremely critical applications, protection against transient failures has received little commercial attention outside of the memory subsystem.

In an effort to keep up with Moore's Law, microprocessor implementations have required ever decreasing feature size and supply voltage. As a consequence of the reduced capacitive node charge and noise margin, even flip-flop circuits will inevitably become susceptible to soft-errors [6]. The high clock rate of modern processors further exacerbates the problem by increasing the probability of a new failure mechanism where a momentarily corrupted *combinational* signal is latched by a flip-flop. These necessary evils of continually pushing the processor performance envelope will shortly[1] place us in an unfamiliar realm where logically correct implementations alone cannot ensure correct program execution with sufficient confidence.

2.2 Fault-Tolerant Computing

It should not be surprising that even today no commercial ICs are guaranteed to operate perfectly. In fact, international standards exist to prescribe what is an acceptable, albeit a very low, frequency of failure [18]. When additional confidence in reliability is called for, vendors of high-availability platforms have long incorporated explicit error detection and correction techniques in their architectures. The basic techniques involve information redundancy, space redundancy and time redundancy.

Protecting data words with information redundant coding, such as parity or Hamming code, allows some number of bit errors to be detectable or correctable. Information redundancy comes at the cost of additional storage for the coding overhead, as well as the cost of the encoding and checking logic. Memory arrays can be ECC-protected relatively efficiently because the cost of the coding logic can be amortized over the array. Applying ECC to individual registers in a processor can involve an exorbitant amount of overhead and also increases the critical path delay. Typically, information redundancy is reserved for memory, caches and perhaps register files, whereas space and time redundant techniques are employed elsewhere in the processor.

Space redundancy is achieved by carrying out the same computation on multiple independent hardware at the same time. Errors are exposed by corroborating the redundant results. For systems with triple (or higher) redundancy, a correct answer can be obtained by a majority election scheme in certain failure modes. For duplex system, computation must be restartable in order to recover from an error. To avoid the large hardware overhead of space redundancy, an alternative is time redundancy where redundant computation is obtained by repeating the same operations multiple times on the same hardware. Time redundancy has the shortcoming that persistent hardware faults may introduce identical errors to all redundant results, making

[1] Intel speculates that by 2007 they would have production microprocessors with 0.045μ technology operating at less than 1 volt and clocked at 20GHz [10].

errors indiscernible. A proposed workaround involves transforming the input operands (such as rotating the operands for bit-wise logical operations) between redundant executions to expose a persistent fault [13]. Space redundancy has a complementary shortcoming that a transient failure mechanism may affect the space redundant hardware identically, again making errors indiscernible.

Due to the high cost of fault tolerance and the relatively low likelihood for errors in present-day technologies, fault-tolerant processor designs have only been justified in specialty systems and very high-end mainframes/servers intended for mission-critical applications. Examples of commercial fault-tolerant processing systems are IBM z900 [16] and Compaq NonStop Himalaya [5]; both employ a combination of redundancy techniques described above.

IBM z900 (previously S/390) employs extensive fault-tolerant mechanisms throughout the system, accounting for approximately 20% to 30% of all logic. In particular, since G4, microprocessors for IBM mainframes have employed two fully-duplicated lock-step pipelines. When the two pipelines disagree in an instruction's result, the processor reverts to *millicode* to carry out extensive hardware checks, and, on transient errors, it can restore the program state from a special hardware checkpoint module. The whole process can take up to several thousand processor cycles. Although these processors offer superb fault-tolerance, they are high-end specialty items because their design trade-offs are substantially different from commodity microprocessors.

Compaq NonStop Himalaya comprises of two stock Alpha processors running the same program in complete locked step. Faults are detected by comparing the output of the two processors at the external pins on every clock cycle. The two processors are halted immediately if they disagree to prevent errors from corrupting the memory and storage subsystems. Although Compaq is able to leverage their commodity workstation processors in their NonStop Himalaya systems, they are not able to provide hardware support for seamless recovery following a transient failure.

2.3 Related Work in Transient-Fault Tolerance

In the coming paragraphs, we describe some recent work in addressing the problem of transient-faults in future commodity processors. Our work shares many of the common elements examined by these closely-related efforts. However, our investigation, based on a superscalar datapath, attempts to provide a design that can optionally divert its full resources toward **single-thread performance** or **reliability**. Our work also integrates recovery into fault tolerance with very little additional cost.

The inherent hardware redundancy in simultaneous multithreading (SMT) [20] and the chip multi-processor (CMP) architectures make them ideal bases for space and time redundant fault-tolerant designs. However, in normal operation, these multithreaded/multiprocessor architectures are throughput-optimized and depend on the existence of multiple threads for maximum performance. We believe the ratio of single-thread performance over cost will remain an important factor in PC and embedded markets. Rotenberg is the first to suggest using an SMT architecture to execute two copies of the same program for fault-tolerance [15]. The two copies, known as the A-thread and the R-thread, proceed with a slight lag. Mechanisms are introduced to verify the outcome of the two threads on an instruction-by-instruction basis. In the same paper, Rotenberg also describes mechanisms to reduce the throughput penalty of redundant execution by expediting the R-thread using A-thread-assisted control and data predictions. The paper did not discuss the means for recovery. A later paper develops similar concepts in the context of CMPs [19].

Reinhardt and Mukherjee have improved upon SMT-based fault-tolerant designs by checking only instructions that exit the processor core, permitting looser coupling between the redundant threads [14]. They have also suggested mechanisms to speedup the delayed threads using information from the leading thread. No recovery scheme is suggested to complement this coarse-grain detection scheme. Their paper presents a thorough examination of the issues in SMT-based fault-tolerance. In particular, they describe the concept of

sphere of replication for aiding the design and discussion of fault-tolerant processors. In short, the parts of the processor that fall outside of the sphere are not replicated and must be protected via means such as information redundancy; components inside the sphere must either be physically replicated for space redundancy or logically replicated via time redundancy.

Austin proposes a very different fault-tolerant scheme in the DIVA architecture [2]. DIVA comprises of an aggressive out-of-order superscalar processor and, on the same die, a simple in-order *checker* processor. The checker processor verifies the output of the complex out-of-order processor and triggers a recovery action when an inconsistency is found. Besides transient fault-tolerance, assuming the simple DIVA checker processor is free of errors, it is also able to correct errors due to design mistakes in the aggressive front-end processor. DIVA offers an interesting solution to combat processor design complexity, but the static nature of its hardware redundancy prevents it from regaining additional performance when reliability is not warranted.

3 Transient-Fault Tolerant Superscalar

This section presents our proposal for a fault-tolerant superscalar processor. (In the rest of this paper, we use fault-tolerance to mean transient-fault tolerance.) Our objective is to develop a set of extensions that leverage pre-existing speculative out-of-order hardware and minimizes disturbances to modern superscalar designs.

3.1 Baseline Processor and Assumptions

Earlier proposals for aggressive superscalar designs have employed centralized bookkeeping structures such as RUU [17]. Most recent implementations have adopted a more decentralized organization with distributed reservation stations, reorder buffer (ROB) and rename register file. Our proposal is compatible with both schemes. In the following paragraphs, we briefly describe the main architectural assumptions for the baseline superscalar processor.

For concurrent error detection, we rely on register renaming hardware to temporarily split an instruction stream into multiple data-independent threads. In our baseline architecture, rename registers reside with ROB entries; speculative values are transferred to a separate committed register file upon an instruction's retirement. A *map table* maintains mapping from logical register names to physical register locations. We will also consider the alternative where renaming is carried out by associatively searching the "logical destination" column of ROB and the case where committed registers and rename registers are held in a unified pool of physical registers.

For error recovery, we rely on speculative execution hardware to maintain both in-order (or committed) and out-of-order (or speculative) states such that the processor can always revert to the known-to-be-good committed state after encountering an exception. In our baseline architecture, ROB records all outstanding instructions in program order and tracks their execution status. While in ROB, instructions are considered speculative; their side-effects on committed program states are delayed in ROB until retirement. Although instruction executions are dynamically reordered, instructions retire from ROB in strict program order.

In coming microprocessor generations, memory cell arrays are likely to experience an unacceptable rate of transient failures sooner than the rest of the processor. Therefore we anticipate all or most of the on-chip storage arrays to be ECC protected. In this paper, we assume ECC protection is viable for all simple array structures, i.e., ones that only support basic indexed read and write access to a whole entry. In particular, we assume all committed program states (including register files, caches, main memory and TLBs) are ECC protected. Internal state registers and complex-access structures like ROB cannot be efficiently ECC protected. Thus, speculative state and intermediate execution results must be covered by either space or time redundant processing. In short, the *sphere of replication* in our design separates committed program states from speculative execution states.

3.2 Fault Tolerance Mechanisms

The proposed fault tolerance extensions consist of three parts: instruction injection (replication), fault detection and transient fault recovery.

Instruction Injection
The instruction injection logic in the decode stage temporarily creates multiple redundant threads from a single instruction stream. Standard superscalar designs can decode and dispatch multiple instructions concurrently. The same capability could be borrowed to redundantly decode and dispatch the same instruction

R number of times, where R is the desired degree of redundancy. The R decoded copies of the same instruction are allocated to consecutive ROB entries.

The complication in this step is in separating the data dependence between the R threads. By always storing redundant copies of an instruction in consecutive ROB entries, if the ROB size is a multiple of R, we can insist two active ROB entries with indices i and j belong to the same thread only if $i \equiv j \pmod R$. Thus, if renaming is achieved by associatively searching the "logical destination" column of ROB then the above condition can be added to the match criteria. If an operand of entry i is renamed to the instruction result in entry j, and $i \equiv j \equiv 0 \ (mod\ R)$, then the corresponding operand of entry $i + k$ must be renamed to the entry $j + k$ for $0 < k < R$. In other words, for each new instruction, we only need to rename the operands for the first copy, and the rename tags for the remaining copies can be deduced by adding an offset k. Thus, only one map table is needed regardless of R. The contents of the sole rename table must be protected by ECC, however. In some architectures, both the committed registers and rename registers are held in a unified pool of physical registers. In this case, copies of the same instruction need to be allocated an aligned block of consecutive rename registers.

When redundant instruction injection is in effect, the effective dispatch bandwidth of a processor is reduced by a factor of R, and the effective capacities of ROB and the rename register file are similarly reduced by a factor of R. Furthermore, the peak execution stage throughput is also reduced by approximately a factor of R because each instruction is executed R times. The effective throughput, however, degrades by a lesser amount if some functional unit types (integer ALUs, memory port, etc.) were not fully utilized in normal (non-redundant) operation. In general, increasing R reduces performance but provides better coverage against transient faults. We expect R to be either 2 or 3 in most designs. Sections 4 and 5 discuss the performance impact of redundant instruction injection in greater detail.

Fault Detection

The redundancy between threads only exists temporarily during speculative execution — the threads are re-merged into a single committing instruction stream before updating the committed program state. After redundant copies of the same instruction are injected during decode, their executions proceed normally until the commit stage. When all copies of the same instruction have been executed and are the oldest entries in ROB, the R entries are cross-checked. If all entries agree, then they are freed from ROB, retiring a single instruction. If any fields of the entries disagree, then an error has occurred and recovery is required. Under our current proposal, checks are only performed when the entries are retiring. There is a small performance advantage to detect and recover from a fault sooner, such as right after all copies of an instruction have been executed but are not yet the oldest. However, the improvement is too small at any reasonable error rate to justify the drastic increase in hardware complexity, and furthermore, the copies of an instruction must still be rechecked at commit time in case a value becomes corrupted while waiting to commit.

Control flow instructions are also redundantly issued, but as soon as one copy of a branch instruction evaluates and disagrees with the predicted branch direction or target, branch rewind is triggered immediately based on this singular result. Only redundant copies of a correctly predicted branch, or a corrected branch, will eventually reach the commit stage. At which point, the redundantly evaluated branch decisions and targets are cross-checked. To ensure the correctness of control-flow, an ECC-protected register must hold the next-PC of the last committed instruction as part of the committed program state. Every retiring instruction's PC must be checked against the last committed next-PC.

To corroborate the retiring results, R accesses to ROB is needed to retire a single instruction, and thus the effective commit/retire bandwidth is reduced by a factor of R. The memory and register file write ports may become under-utilized relative to the rest of the system because we only perform one write per R retiring ROB entries. However, the number register file and memory ports cannot be reduced since the overall processor design must remain balanced for normal operation in the unprotected mode. When a unified pool

of physical registers is used for both committed registers and rename registers, corroborating the results of different threads requires two additional register file read accesses per retiring instruction. In addition, the redundant rename registers cannot be easily coalesced into one committed register. Thus, the performance of fault-tolerant superscalar derived from a microarchitecture with a unified register file will be lower then reported in Section 5.

Recovery

Once the concurrent error detection mechanism is in place, recovery can be achieved at nearly no cost, in terms of both hardware and performance, using the pre-existing execution rewind mechanism. After an inconsistency is detected between redundantly executed copies of a retiring instruction, the default action is to completely rewind the ROB, i.e. discard the entire ROB contents and restart execution by refetching from the committed next-PC register. If the error is non-persistent, the program execution can proceed correctly on the next attempt. A rewind-based recovery only introduces a penalty on the order of tens of cycles and has a negligible effect on throughput for even the highest expected error rate.

For designs with $R \geq 3$, inconsistency between redundant copies of a retiring instruction could also be resolved by a majority election. Under this scheme, an extra degree of design choice is the correctness acceptance threshold, i.e. how many copies must agree before one accepts the majority result as correct. If an acceptable majority exists, then an instruction is allowed to commit even if discrepancies are detected; otherwise a complete rewind is invoked. In Section 5, we will show that an 'R=3' design with majority election performs better than an 'R=2' rewind-based design only when the error rate is exceedingly high. Thus we conclude 'R>2' designs are only useful in increasing the confidence in fault coverage.

3.3 Fault Coverage

This fault-tolerant superscalar design assumes the committed processor states are protected by traditional information redundant techniques. Assuming the committed states are adequately protected, a single-event upset within the speculative portions of a processor can always be corrected by reverting to a known-to-be-correct committed previous state.

Instruction replication begins at decode, and independence between the redundant copies is maintained throughout until the temporary threads are re-merged into a single committing stream in the commit stage. All information between the decode stage and the commit stage are R-redundant in storage and computation. In other words, every intermediate result is represented by and stored as R separate copies. The R copies are the results of R space-independent or time-independent computations, and the redundant computations do not share any common input. A single-event upset that causes an intermediate error with an observable end effect will be detected during corroboration against other unaffected threads in the commit stage. Only "proven"-to-be-good results are allowed to update the committed states.

An instruction leaves the protection of ECC-protected I-cache at fetch time. There is a window of vulnerability between fetch and decode, while an instruction resides in the fetch queue. Due to the simple nature of fetch queue accesses and its essentially RAM-based implementation, it is feasible to use ECC to protect the buffer contents. Trace caches also falls in the same category and must be ECC protected as well. Another point of vulnerability is the PC register which is not duplicated. However, any error in a program's control flow sequence would be detected at retirement because we check the control flow relationship between every pair of consecutive retiring instructions. For the same reason, BTB arrays do not need to be protected.

Figure **??** shows the additional hardware required to implement our scheme. The dark gray blocks are new to superscalar architecture while the light gray blocks requires ECC bits for protection.

3.4 Other Design Considerations

Below we discuss some of the issues that are considered but not incorporated into our design.

Granularity of Error Checking: The proposed fault-tolerant superscalar design is based on fine-grain checking at the commit stage of every instruction such that the execution of redundant threads are tightly coupled. In the context of an SMT architecture, Reinhardt and Mukherjee proposed to let the redundant threads run unchecked except when information is about to exit the processor core [14]. The advantage of coarse-grain checking is that threads do not need to be synchronized in the pipeline and the register files do not need to be ECC protected. However, much additional hardware is needed to reconcile the results between two far flung threads. Recovery (not specified in the paper) for this scheme would conceivably require an elaborate process involving a large amount of history. The coarse-grain model is natural for SMT-based designs, but the extra hardware requirement to support multiple truly-independent thread contexts conflicts with the goal to preserve maximum normal single-thread performance.

Staggered Thread Executions: Reinhardt and Mukherjee also suggested that allowing the execution of the redundant threads to be staggered over time can improve performance over more tightly coupled executions. The performance gain of staggered execution does not come from smoothing contentions for a particular functional unit type; a sufficiently large ROB can serve that purpose. Rather, in an SMT-based redundant scheme, the improvement is due to reduced cache misses and branch mispredictions in the delayed thread. A fault-tolerant superscalar does not need to recover the cost associated with redundant instruction fetches, and, thus, it does not need to use prioritized scheduling to create an artificial stagger between the redundant threads.

Multi-cycle and Correlated Faults: Transient faults that last multiple cycles could cause identical errors to time-redundant operations, leaving the error indiscernible. Increasing the stagger between time-redundant operations gives additional safe-guards against this failure mode. Although radiation and noise-related faults typically do not lead to this failure mode, this is a conceivable scenario for failure mechanisms (such as overheating) with slower transients. Our design limits the amount of drift between the redundant threads, but this coupled execution style can be used to our advantage by co-scheduling redundant copies of the same instruction such that they are executed on different physical functional units whenever possible.

4 Performance Implications

The IPC penalty of fault-tolerant superscalar executions can be attributed to two sources:

1. Steady state penalty due to redundant instruction processing
2. Error recovery overhead that is a function of transient failure frequency

In this section, we develop analytical models to understand and predict the performance cost of reliability. In the models, R is the degree of redundancy; IPC_o is IPC of the unmodified superscalar datapath; and IPC_R is IPC of the same datapath modified to support R redundant threads. CPI_o and CPI_R are defined correspondingly.

4.1 Steady State Penalty

To the first order, IPC_R can be approximated by $\frac{IPC_o}{R}$. In other words, the throughput of the processor is reduced by a factor of R when each instruction must be processed R times and therefore consumes R times as much resources and bandwidth. In practice, IPC_R sometimes fares better than $\frac{IPC_o}{R}$ because IPC_o is not always limited by the peak throughput of the processor. IPC_o achieved by an application is also strongly determined by the application's instruction-level parallelism (ILP). When we create R data-independent threads from a single instruction stream, we have effectively increased the available ILP by a factor of R. Ideally, until the processor resources become saturated, the extra data independent operations consume the previously unused capacities and incur little cost. This can be captured by the following equation,

$$IPC_R = IPC_o - max\left\{\frac{R \cdot IPC_o - B}{R}, 0\right\}$$

where the subtrahend corresponds to the redundant processing penalty.[2] The term B represents the first resource bottleneck exercised by an application, typically the number of functional units of a particular type.

4.2 Recovery Penalty

In the following discussion, let f be the average transient failure frequency (in terms of failures per instruction[3]) for an unmodified superscalar datapath. Converting to an R-redundant superscalar datapath, the average frequency at which one of the R copies of an instruction becomes corrupted is $R \cdot f$. Whenever the redundant copies disagree, the processor must discard its speculative state and rewind to the known-to-be-good committed state. Let p be the average rewind penalty in terms of the number of cycles added to the execution. On average, a superscalar processor in fault-tolerant mode will take p more cycles to commit $\frac{1}{R \cdot f}$ instructions. Thus, CPI_R as a function of f can be approximated by

$$CPI_R(f) = CPI_{R-ss} + \frac{p}{\frac{1}{R \cdot f}}$$

where CPI_{R-ss} is the error-free steady-state CPI discussed in Section 4.1. Converting $CPI_R(f)$ to $IPC_R(f)$, we get,

$$IPC_R(f) = \frac{IPC_{R-ss}}{1 + p \cdot R \cdot f \cdot IPC_{R-ss}}$$

These equations are not accurate for very high error frequency (i.e. $\frac{1}{f} \approx p$) because at such frequencies, rapid successions of faults may only incur one rewind penalty.

[2] Simply put, if $IPC_o < \frac{B}{R}$ then $IPC_R = IPC_o$ else $IPC_R = \frac{B}{R}$.

[3] In other words, we expect 1 instruction execution to produce an incorrect result in $\frac{1}{f}$ instructions.

4.3 Applying the Models

Figure 2 plots idealized $IPC_{R=2}(f)$ and $IPC_{R=3}(f)$ where we assume IPC_o=B and is normalized to 1. IPC_o=B represents the case when single-thread executions already have sufficient ILP to saturate the bottleneck B and thus IPC_R=$\frac{IPC_o}{R}$. Both R=2 and R=3 use rewind for recovery, and p is assumed to be 20 cycles in both cases. From Figure 2, we see that IPC of 'R=2' and 'R=3' stays relative constant until $\frac{1}{f}$ is within two orders of magnitude of p, at which point rewind penalties constitutes a significant fraction of the total execution time. We do not intend our design to be operated under such high error rates. A third curve shows IPC_R vs. f for a 'R=3' design that uses majority-election as well as rewind.

Figure 3 plots the same information as Figure 2 except p has been increased to 2000 cycles to reflect a much coarser grain error detection and recovery scheme. Comparing Figures 2 and 3, p has only a minimal effect on the average IPC for any reasonable values of f. However, p has another important effect if a processor needs to maintain a real-time guarantee of executing a certain number of instructions within some window of time. A large p can only be amortized over a corresponding large window, making fine-grain real-time guarantees impossible.

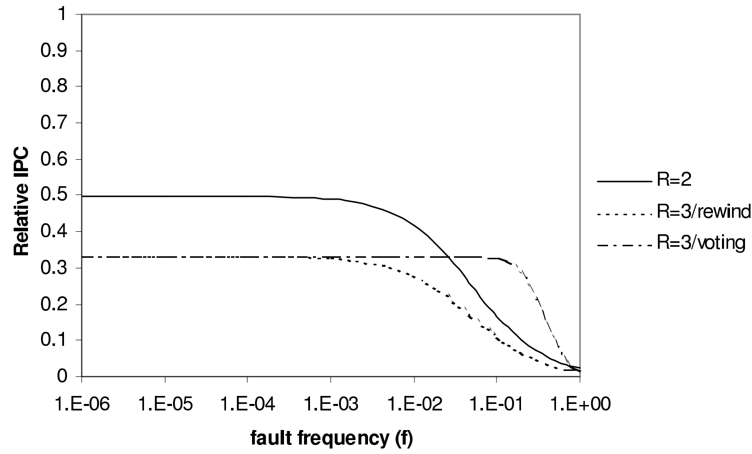

Figure 2. IPC vs. fault frequency for $p = 20$

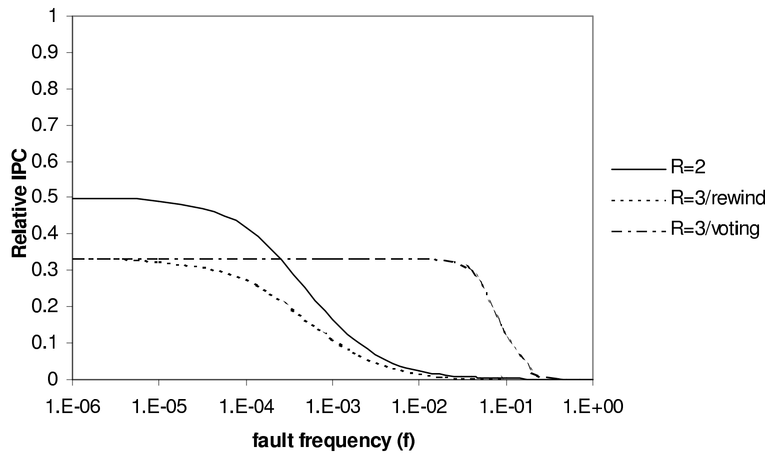

Figure 3. IPC vs. fault frequency for $p = 2000$

5 Performance Simulation and Evaluation

In this section, we present a more detailed performance evaluation based on cycle-accurate performance simulations of 11 benchmarks from the SPEC95 and SPEC2000. We begin by describing the simulation environment, simulated microarchitectures and selected benchmarks.

5.1 Experimental Setup

5.1.1 Performance/Functional Simulator

We use a modified version of SimpleScalar [4] for both performance and functional simulations. We modify the stock out-of-order simulator to implement instruction replication, fault detection and recovery schemes described in Sections 3. We also introduce a "fault injection" module that can randomly corrupt some instructions based on a user-specified probability distribution function. Because our fault injection module may decide to corrupt some part of an instruction at any stage of the pipeline, significant changes have to be made to the stock *sim-outorder*'s branch rewind code to allow rewinds to be decided later than the decode stage.

Our modified simulator maintains two sets of committed register file and memory state. One set is updated by instructions that have successfully graduated through the commit stage after faithfully passing through the out-of-order pipeline. The other set, concurrently maintained as a sanity check, is updated by executing the program in an in-order, non-speculative manner. During simulations, we have the option to periodically drain the pipeline to compare the two sets of states to ensure our error detection scheme has captured the randomly injected faults and the recovery scheme has correctly restored the processor to a good state after an injected fault.

5.1.2 Simulated Machine Parameters

For our main performance results, we simulated three machine models. The baseline machine (SS-1) is a single-thread out-of-order, superscalar processor simulated using the stock *sim-outorder* simulator. The salient machine parameters are summarized in Table 1. The parameters are chosen to reflect an aggressive but contemporary processor design point. The same machine descriptions are given to our modified simulator to simulate a 2-way-redundant fault-tolerant superscalar processor (SS-2). To ensure a fair comparison, extra resource usages are carefully accounted for as to not exceed the capacity available to the baseline model. In the redundant execution of memory instructions (both loads and stores), the memory addresses are computed redundantly, but only one memory access is performed. The last model (Static-2) in our study reflects a statically-redundant processor with two identical and independent pipelines that run two copies of the program in locked-step (such as IBM G4/G5/G6). Static-2 is modeled using the stock *sim-outorder* simulator with half of the resources (except caches and BP hardware) as in Table 1. This corresponds to statically dividing the baseline resources equally into two pipelines.[4]

5.1.3 Benchmarks

The simulation results are based on 11 benchmarks selected from SPEC95 and SPEC2000. The benchmarks are compiled for PISA ISA using `gcc -O2 -funroll-loops`. The reference inputs are used for each benchmark. For each benchmark measurement, the results typically are averaged over a 1-billion-instruction simulation (after skipping the first 1 billion instructions when appropriate). Table 2 gives a summary of the benchmarks in terms of their dynamic instruction profile.

[4]Each redundant pipeline has an FP Mult/Div unit. Thus, Static-2 in effect has the advantage of an extra FP Mult/Div unit.

Fetch/Decode/Dispatch Width	8
Issue width	8
RUU/LSQ size	128/64
Branch Predictor	Combined predictor that selects between a 2K bimodal and a 2-level predictor. The 2-level predictor consists of a 2-entry L1 (10-bit history), an 1024-entry L2, and 1-bit *xor*. **One prediction per cycle.**
Instruction L1 cache	64 KBytes, 2 way associative.
Data L1 cache	32 KBytes, 2-way associative, 2 R/W ports.
Unified L2 cache	512 KBytes, 4-way associative.
Functional Unit Mix	4 Int ALU, 2 Int Mult, 2 FP Add and 1 FP Mult/Div. **All FU operations are pipelined except for division.**

Table 1. ***sim-outorder*** **machine parameters for the baseline superscalar model**

Benchmark	Inst Skipped	Inst Simulated	% Mem Ops	% Int Ops.	% FP Add	% FP Mult	% FP Div
gcc	1073741824	1073741824	74.55	25.45	0	0	0
vortex	1073741824	1073741824	54.56	45.44	0	0	0
go	1073741824	1073741824	29.49	70.50	0	0	0
bzip	1073741824	1073741824	29.84	70.16	0	0	0
ijpeg	0	1073741824	26.06	73.94	0	0	0
vpr	1073741824	1073741824	31.30	63.61	3.57	1.38	0.15
equake	1073741824	1073741824	34.55	52.82	6.06	6.41	0.16
ammp	1073741824	1073741824	41.35	56.64	1.49	0.50	0.02
fpppp	1073741824	1073741824	52.43	15.03	15.53	16.84	0.16
swim	1073741824	1073741824	32.71	37.41	19.31	10.12	0.47
art	1073741824	268435456	35.29	43.50	11.07	8.39	1.36

Table 2. Summary of Benchmark Characteristics

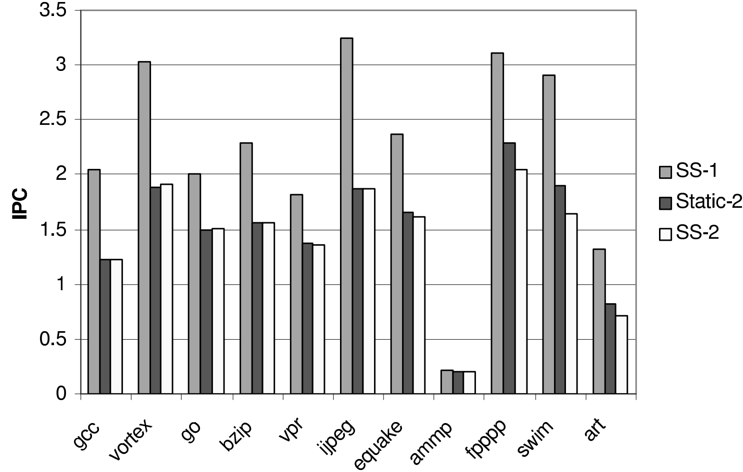

Figure 4. Steady-State IPC Comparison

5.2 IPC Performance Comparisons

Figure 4 compares IPC of the three processor models (SS-1, Static-2 and SS-2) for each of the 11 benchmarks. On average, the 2-way dynamic redundant superscalar (SS-2) achieves 30% lower IPC than the baseline processor (SS-1). For comparable hardware configurations, SS-2 should perform comparably to the results reported for the SMT-based fault-tolerance with *slack-fetch* and *branch-outcome queue* [14]. Overall, the 2-way dynamic redundant superscalar (SS-2) performs comparably to the static two-pipeline processor (Static-2). For *fppp*, *swim*, and *art* Static-2 significantly out performs SS-2 due to the extra FP Mult/Div unit.

The benchmarks *ammp*, *go* and *vpr* suffer less IPC penalty in SS-2 then the rest of the benchmarks. To help understand this, we test the benchmark's sensitivity to varying numbers of functional units (0.5x, 2x, infinite) and RUU sizes (0.5x, 2x, infinite). From the results of these experiments, we are able to determine that these other benchmarks with higher IPC penalties are already functional-unit limited in the baseline configuration. (*swim* is also RUU-limited.) In other words, the single-thread throughput (IPC_o) is already exercising some hardware bottleneck (B). Therefore, injecting a second thread has a big impact on the effective IPC (IPC_R). On the other hand, *go* and *vpr* are almost insensitive to the amount of resources available. This is an indication that their IPC_o is determined by ILP, and therefore the extra ILP from the second thread has free use of previously under-utilized resources. *ammp* is an extreme case where its IPC_o is limited by a large number of divisions in its critical path.

5.3 Fault Recovery Performance

This section presents the recovery costs of fault-tolerant superscalar execution. For these experiments, the fault-injection module in our modified simulator is enabled to randomly corrupt some instructions at varying frequencies. Using the same machine parameters as in Table 1, we have simulated two designs with R=2 and R=3, respectively. For the 'R=3' design, majority-election is used to recover from a corrupted instruction whenever possible.

Figure 5 compares the observed IPC_R for *fpppp* on the two designs over different fault frequencies. The X-axis is the average fault frequency given in faults per one million instructions, while the Y-axis is the corresponding IPC_R. This plot corresponds closely to the analytical model in Section 4.2. As expected,

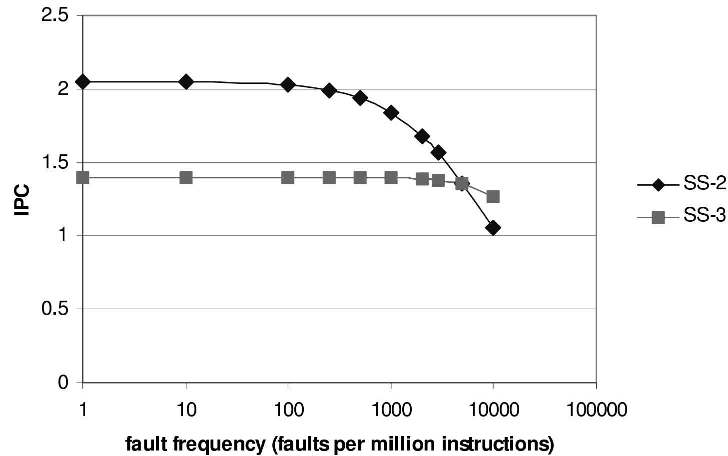

Figure 5. IPC vs. fault-frequency for *fpppp*

IPC of the 'R=2' design drops sharply when faults are sufficiently frequent for recovery penalties to be a significant part of the execution time. Typical recovery costs observed in *fppp* simulations are around 30 cycles. IPC of the 'R=3' design is lower but remains unaffected until much higher frequencies because there is no rewind penalty until 2 out of 3 copies of an instruction are corrupted. IPC of the more efficient 'R=2' design eventually drops below the 'R=3' design, but the cross-over occurs at a much higher fault frequency than what our design is intended for. (At such high fault-frequencies, fundamentally different solutions in terms of architecture or implementation technology is called for.) In our intended range of error frequency, 'R=2' offers a clear performance advantage. 'R=3' designs are only applicable if extra redundancy is desired for higher confidence in fault-coverage, or if the application cannot tolerate even the small performance *hiccups* of rewind recovery.

6 Conclusions

The push toward deep submicron devices is a key enabler in the continued exponential increase in microprocessor performance. Given the anticipated feature size, noise margin and clock rate, it is inevitable that processors will begin to experience an unacceptable level of transient hardware faults in both logic and memory. An effective microarchitectural counter-measure must include both detection and recovery.

In this paper, we have presented an efficient fault-tolerance technique for current out-of-order superscalar microarchitectures. The proposal relies on three key elements to provide transient-fault tolerance through hardware redundancy: (1) dynamic instruction injection that creates redundant threads of executions, (2) value synchronization that compares redundantly executed instruction results to detect errors, and (3) recovery by reverting to a known-to-be-correct previous state using the same mechanism as branch rewind. These new functionalities require only minimal extensions to pre-existing mechanisms that already serve other useful purposes. Redundant instruction execution incurs a noticeable performance penalty but the results are inline with other redundant execution techniques. The performance penalty of rewind-based error recovery is negligible until exceedingly high error rates.

This current design is most useful during a transitional period when transient failure rates are only beginning to be unacceptable for some applications. The flexibility of this fault-tolerant approach allows increased protection when reliability is absolutely critical but at the same time allows the hardware to return to full performance for gaming and multimedia scenarios. We plan to expand our investigation to include fault-tolerance techniques that are applicable to superscalar microarchitectures and threaded architectures such as SMT and CMP. The goal is to develop a family of soft-error protection mechanisms that can be combined or applied independently to ensure optimal instruction throughput under different reliability requirements and varying fault frequencies.

References

[1] L. Anghel, D. Alexandrescu, and I. Alzaher-noufal. Self-checking circuits versus realistic faults in very deep submicron. In *Proceedings of the 18th IEEE VLSI Test Symposium*, April/May 2000.

[2] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd International Symposium on Microarchitecture*, November 1999.

[3] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, July-August 1999.

[4] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, University of Wisconsin-Madison, Computer Sciences Technical Report, 1997.

[5] Compaq Computer Corporation. Data integrity for Compaq NonStop Himalaya servers. http://nonstop.compaq.com, 1999.

[6] F. Faccio, et al. Single event effects in static and dynamic registers in a $0.25\mu m$ CMOS technology. *IEEE Transactions on Nuclear Science*, 46(6):1434–1439, December 1999.

[7] M. J. Flynn, P. Hung, and K. W. Rudd. Deep-submicron microprocessor design issues. *IEEE Micro*, 19(4):11–22, July-August 1999.

[8] M. Franklin. A study of time redundant fault tolerant techniques for superscalar processors. In *Proceedings of IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, 1995.

[9] A. Hasnain and A. Ditali. Building-in reliability: Soft errors - a case study. In *Proceedings of the 30th International Reliability Physics Symposium*, April 1978.

[10] Intel Corporation. Intel researchers build world's fastest silicon transistors. Press Release, June 2001.

[11] T. C. May and M. H. Woods. A new physical mechanism for soft errors in dynamic memories. In *Proceedings of the 16th International Reliability Physics Symposium*, April 1978.

[12] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43:2742–2750, 1996.

[13] J. H. Patel and L. T. Fung. Concurrent error detection in ALU's by recomputing with shifted operands. *IEEE Transactions on Computers*, 31(7):589–595, July 1982.

[14] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[15] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing Systems*, June 1999.

[16] T. J. Slegel, et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, March/April 1999.

[17] G. S. Sohi. Instruction issue logic for high-performance interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3), March 1990.

[18] STACK International. *General Requirements for Integrated Circuits*, 2001.

[19] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 33rd International Symposium on Microarchitecture*, December 2000.

[20] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[21] J. F. Ziegler and W. A. Landford. Effect of cosmic rays on computer machines. *Science*, 206, 1979.

[22] J. F. Ziegler, et al. IBM's experiments in soft fails in computer. *IBM Journal of Research and Development*, 40(1), 1996.