Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors

Moinuddin K. Qureshi Onur Mutlu Yale N. Patt
Department of Electrical and Computer Engineering
The University of Texas at Austin
{*moin, onur, patt*}*@hps.utexas.edu*

Abstract

The increasing transient fault rate will necessitate on-chip fault tolerance techniques in future processors. The speed gap between the processor and the memory is also increasing, causing the processor to stay idle for hundreds of cycles while waiting for a long-latency cache miss to be serviced. Even in the presence of aggressive prefetching techniques, future processors are expected to waste significant processing bandwidth waiting for main memory.

This paper proposes Microarchitecture-Based Introspection (MBI), a transient-fault detection technique, which utilizes the wasted processing bandwidth during long-latency cache misses for redundant execution of the instruction stream. MBI has modest hardware cost, requires minimal modifications to the existing microarchitecture, and is particularly well suited for memory-intensive applications. Our evaluation reveals that the time redundancy of MBI results in an average IPC reduction of only 7.1% for memory-intensive benchmarks in the SPEC CPU2000 suite. The average IPC reduction for the entire suite is 14.5%.

1. Introduction

Transient faults present a serious challenge to the correct operation of future processors. A transient fault occurs in a processor when a high-energy cosmic particle strikes and inverts the logical state of a transistor. Technology trends such as reduction in operating voltage, increase in processor frequency, and increase in the number of on-chip transistors indicate that the transient fault rate is likely to increase by orders of magnitude [15]. In order to address this problem, future processors will need to incorporate on-chip fault tolerance techniques. Fault-tolerant techniques with *low hardware overhead* are desirable because they make fault-tolerance pervasive by giving the users an option of fault-tolerance without committing to a heavy hardware cost. Thus, if the user chooses not to have a fault-tolerant processor, then only the small hardware resources dedicated solely to fault tolerance will go unused. This paper investigates a low hardware overhead fault tolerance mechanism that is relatively simple and can be easily incorporated in an existing microarchitecture.

Fault tolerance has traditionally been achieved using two techniques: *redundant code* and *redundant execution*. Storage structures have regular patterns, which lend themselves to redundant codes. These structures can easily be protected with well-understood techniques such as parity and Error Correcting Codes (ECC). In contrast, combinational logic structures have irregular patterns, which often necessitate redundant execution for fault tolerance. Redundant execution can further be classified into *space redundancy* and *time redundancy*. Space redundancy is achieved by executing a task on multiple disjoint structures. Space redundancy has low performance overhead but requires hardware in proportion to the number of redundant structures. Time redundancy is achieved by executing a task on the same hardware multiple times. Time redundancy has low hardware overhead but high performance overhead. The performance overhead of time redundancy can be reduced if the redundant execution is scheduled during idle cycles in which the processing bandwidth is otherwise wasted. This paper focuses on utilizing the wasted processing bandwidth due to long-latency cache misses for time redundancy.

Processor frequency has increased at a much faster rate than DRAM memory speed, which has led to a widening speed gap between the processor and the memory. This problem is partially addressed in current processors through multi-level on-chip caches, providing fast access to recently-accessed data. Unfortunately, a cache miss at the highest level on-chip cache almost always stalls the processor [5] because the latency to main memory is very long (in the order of hundreds of cycles [21]). The stalled processor cannot execute instructions until the long-latency cache miss is completely serviced. Therefore, these cache misses translate into idle cycles for the processor, resulting in wasted processing bandwidth. The problem is even worse for memory-intensive applications because these applications have large working sets, which cause frequent cache misses. Applications that have traditionally required high fault tolerance, such as transaction processing and database workloads, tend to be memory-intensive [4]. In fact, a recent study [3] showed that database workloads spend about two-thirds of their execution time waiting for memory.

The processing bandwidth wasted due to long-latency cache misses can be utilized for different purposes. Runa-

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 IEEE

head execution [8] pre-executes the application during long-latency cache misses in order to generate prefetches. Li et al. [6] use the spare bandwidth to reduce power by reducing the processor voltage while a cache miss is being serviced. Similarly, the wasted processing bandwidth can be used for increasing the fault tolerance of a processor by re-executing the application during the idle cycles.

In this paper, we propose *Microarchitecture-Based Introspection (MBI)*, a time redundancy technique, to detect transient faults. MBI achieves time redundancy by scheduling the redundant execution of a program during idle cycles in which a long-latency cache miss is being serviced. MBI provides transient fault coverage for the entire pipeline starting from instruction fetch to instruction commit. This technique is completely microarchitecture based and requires no support from the compiler or the ISA. It requires modest hardware overhead and minimal changes to the existing microarchitecture. Because the technique attempts to utilize idle cycles caused by long-latency cache misses, it is particularly well-suited for memory-intensive applications. Our experiments show that using MBI for 13 memory-intensive benchmarks from SPEC CPU2000 results in an average IPC reduction of only 7.1%. The average IPC reduction for the entire SPEC CPU2000 suite is 14.5%.

2. Motivation and Overview

2.1. Motivation for Using Cache Miss Cycles for Redundant Execution

To measure the amount of processing bandwidth wasted due to long-latency cache misses, and its potential for redundant execution, we perform three experiments.[1] In the first experiment, we simulate the baseline machine with a perfect instruction cache, a perfect data cache, and a perfect branch predictor. We measure the Cycles Per Instruction[2] (CPI) of this configuration (CPI-PERF). Redundant execution can leverage the load values and the resolved branch directions from the first execution of the instruction stream and therefore does not need to incur data cache misses or branch mispredictions. Hence, the CPI of the redundant execution can be expected to be very close to CPI-PERF. In the second experiment, we measure the CPI of the baseline machine with a perfect L2 cache, but real first-level (L1) caches and a real branch predictor. The CPI increase for the second experiment with respect to the CPI of the first experiment represents the CPI added due to real L1 caches and branch predictor (CPI-L1-BP). In the third experiment, we measure the CPI of the baseline machine with real L1 and L2 caches and a real branch predictor. The CPI increase for the third experiment with respect to the CPI of the second experiment represents the CPI added due to a real L2 cache (CPI-L2). Figure 1 shows the CPI breakdown in terms of CPI-PERF, CPI-L1-BP, and CPI-L2 for each benchmark.

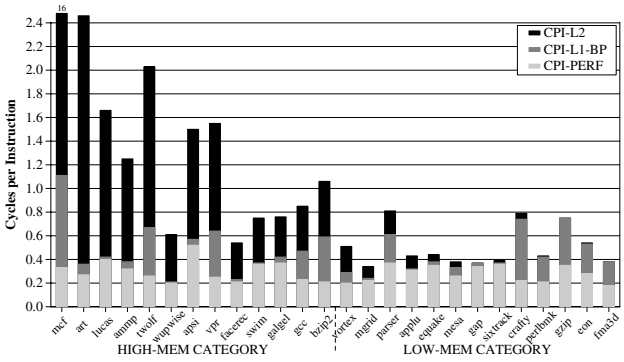

Figure 1. CPI breakdown.

For each benchmark, the proportion of CPI-L2 in total CPI represents the fraction of processing bandwidth that gets wasted due to L2 misses. We sort the benchmarks with respect to this proportion and classify them into two categories: HIGH-MEM and LOW-MEM. The thirteen benchmarks that have the highest proportion of CPI-L2 in total CPI are grouped in the HIGH-MEM category, and the remaining benchmarks are grouped in the LOW-MEM category. The benchmarks in the LOW-MEM category have a small CPI-L2 component in total CPI. These benchmarks either have memory access patterns that are easily prefetchable or have working sets that fit in the L2 cache.

All benchmarks in the HIGH-MEM category, except galgel, have CPI-L2 more than CPI-PERF. These benchmarks have sufficient idle processing bandwidth to re-execute the program with negligible performance degradation. Though our technique is primarily targeted towards such memory-intensive applications, we perform our studies for all the benchmarks in the SPEC CPU2000 suite.

2.2. Overview

In this section, we provide a brief overview of MBI. The operation of a machine implementing MBI has two modes: *performance mode* and *introspection mode*. Instructions are processed for the first time in performance mode. Later, for fault tolerance, instructions are processed again in introspection mode. The architectural state (i.e., architectural registers and memory) is updated only during the performance mode. The purpose of introspection mode is to verify the results produced during performance mode. Thus, the architectural state is not modified during introspection mode unless an error is found. Figure 2 shows the state machine for the MBI mechanism.

[1] Section 4 describes the baseline configuration and benchmarks. The memory system includes a 1MB L2 cache and an aggressive prefetcher.

[2] We use CPI for the experiments in this section because CPI provides a more intuitive breakdown of execution time due to the different components in a processor. For experiments in all other sections, we use Instructions Per Cycle (IPC) to measure performance.

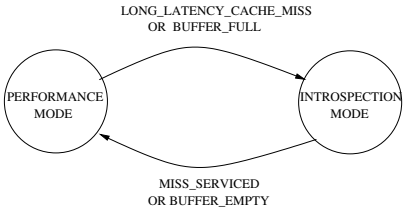

Figure 2. State machine for the MBI mechanism.

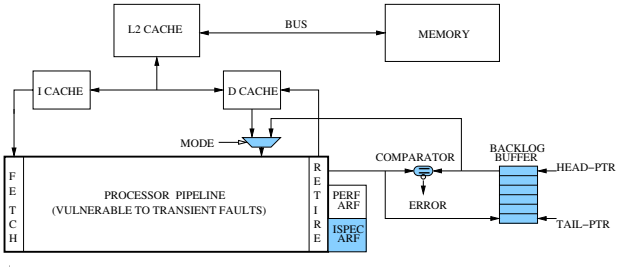

Figure 3. Microarchitecture support for MBI.

In performance mode, instruction processing proceeds without paying attention to incorrect operation resulting from transient faults. Instructions are fetched, executed, and retired. The result of each retired instruction is stored in a structure, called the *backlog buffer*. Writing results to the backlog buffer is a post-retirement operation and does not affect the instruction processing speed in performance mode. When a long-latency cache miss occurs in performance mode, the mechanism switches the state to introspection mode. In introspection mode, the processor re-executes each instruction that was retired in performance mode and compares the new result with the result of the previous execution stored in the backlog buffer. If both results match, then instruction processing for that instruction is assumed to be fault-free and the respective entry is removed from the backlog buffer. A mismatch between the two results signals an error.

If there are no long-latency cache misses for a long period, the processor will not enter introspection mode and the backlog buffer will become full. To guarantee redundant execution for all retired instructions, the processor is forced to enter introspection mode when the backlog buffer becomes full.[3] The processor returns from introspection mode to performance mode when either the backlog buffer becomes empty or when the long-latency miss gets serviced.

3. Design

In this section, we provide details about the implementation and operation of MBI. Figure 3 shows the baseline system and the extensions required to incorporate MBI into an existing microarchitecture. The structures unique to MBI are shaded.

We assume that the storage structures such as caches, register files, and the backlog buffer are protected using ECC and buses are protected using parity. The pipeline is unprotected and vulnerable to transient faults.

[3] An alternative policy does not force the processor into introspection mode when the backlog buffer is full. However, this policy will not provide redundant execution for all retired instructions and, therefore, will not guarantee transient fault coverage for *all* instructions. The percentage of instructions that are redundantly executed during an L2 miss (i.e., the coverage of the instruction stream of this alternative policy) is shown in Section 5.4.

3.1. Structures

MBI requires the following changes to the microarchitecture:

1. Extra register file: The design includes two architectural register files, PERF ARF and ISPEC ARF. Each ARF contains the general purpose registers, control registers, and the Program Counter (PC). The PERF ARF is updated only in performance mode and the ISPEC ARF is updated only in introspection mode.

2. Backlog buffer: This storage structure keeps track of the instructions that were retired in performance mode but have not yet been processed in introspection mode. It is implemented as a circular FIFO buffer and contains two pointers: a tail pointer (TAIL-PTR) and a head pointer (HEAD-PTR). The TAIL-PTR determines the entry in the backlog buffer where the result of the next retiring instruction will be written. Results are written in the backlog buffer only during performance mode. When an instruction writes its result in the backlog buffer, the TAIL-PTR is incremented to point to the next entry. The HEAD-PTR determines the location of the entry in the backlog buffer that will provide the result to be compared with the next completed instruction in introspection mode. The results are read from the backlog buffer only during introspection mode. In our experiments, we assume that the backlog buffer can store the results of 2K instructions. Section 5.2 analyzes the impact of varying the size of the backlog buffer and Section 5.3 analyzes the hardware cost of the backlog buffer.

3. Selection mechanism for load instructions: Because store instructions update the D-cache in performance mode, load instructions read their memory values from the backlog buffer in introspection mode. A newly added mux selects between the D-cache output and the backlog buffer output, based on the mode of the processor.

4. Comparator: A comparator is required to compare the results of redundant execution with the results stored in

the backlog buffer. The comparator logic is accessed after the retirement stage of the pipeline. If the inputs to the comparator are different, an error is signaled in introspection mode.

In addition, depending on the error handling policy, logic for handling errors may be required.

3.2. Operation

A processor implementing MBI can be in performance mode, introspection mode, or switching from one mode to another. We describe the operation for all these cases.

- Operation in performance mode

  In performance mode, the processor performs its normal operation without checking for errors resulting from transient faults. When an instruction retires, it updates the PERF ARF register file and its results are also copied into the backlog buffer. Store instructions update the D-cache and load instructions read their values from the D-cache.

- Operation in introspection mode

  Instruction processing in introspection mode begins with the oldest instruction that was retired in performance mode but has not yet been processed in introspection mode. In introspection mode, the processor fetches the instructions from the I-cache and processes them in the normal manner with four exceptions. First, retired instructions update the ISPEC ARF register file. Second, a load instruction reads its memory value from the backlog buffer. Third, a store instruction does not update the D-cache (memory system is not updated in introspection mode). Finally, the prediction for a conditional branch instruction is provided by the resolved branch direction stored in the backlog buffer. In introspection mode, the processor does not update the branch predictor. This allows the branch predictor to retain its history information for performance mode.

  When an instruction retires in introspection mode, its result is compared with the result stored in the backlog buffer. A match denotes correct operation, whereas a mismatch is signaled as an error. We discuss error handling policies in Section 6.

- Switching from performance mode to introspection mode

  A processor in performance mode can switch to introspection mode due to any of the following reasons:

  1. Long-latency L2 cache miss

     When there is a long-latency L2 cache miss, the instruction causing the cache miss will reach the head of the reorder-buffer and stall retirement until the cache miss gets serviced. When the instruction causing the cache miss reaches the head of the reorder-buffer, the processor waits for 30 cycles and switches to introspection mode. The 30 cycle wait allows the in-flight instructions to execute and to possibly generate additional L2 misses.

  2. Backlog buffer full

     When the backlog buffer is full, the processor is forced to enter introspection mode. This is done to guarantee redundant execution, and, therefore, transient fault coverage, for *all* instructions.

  3. System call

     Instructions retired in performance mode need to go through redundant execution before the processor executes a system call.[4] This prevents external devices from accessing the possibly incorrect results of the instructions that have not been verified through redundant execution. As system calls are infrequent, entering introspection mode before them does not cause a significant performance degradation. For SPEC CPU2000 benchmarks, only 0.0004% of the introspection mode episodes were caused by system calls.

  4. Self-modifying code

     When self-modifying code is detected, the processor is forced to enter introspection mode before modifying any of the instructions. Otherwise, the two executions of the same instruction will likely give different results in performance mode and introspection mode, even if there is no transient fault. We do not have any self-modifying code in our experiments because we model the Alpha ISA, which disallows self-modifying code.

  If the processor switches from performance mode to introspection mode because of a long-latency cache miss, the entry into introspection mode is called *normal-introspection*. Entry into introspection mode for any other reason is called *forced-introspection*. The process of switching to introspection mode comprises two activities. First, the mode bit is switched to indicate introspection mode so that subsequent writes to the ARF update the ISPEC ARF. Second, the pipeline is flushed and the PC from the ISPEC ARF is copied into the PC of the fetch unit so that the processor begins fetching the redundant instructions from the I-cache.

[4] For critical system calls, such as the machine check exception, the processor can transfer control without completing redundant execution of all retired instructions.

- Switching from introspection mode to performance mode.

  For normal-introspection, the processor returns to performance mode when the long-latency cache miss gets serviced or when the backlog buffer becomes empty, whichever is earlier. On the other hand, for forced-introspection, the processor returns to performance mode only when the backlog buffer becomes empty. The process of returning to performance mode comprises two activities. First, the mode bit is switched to indicate performance mode so that subsequent writes to the ARF update the PERF ARF. Second, the pipeline is flushed and the PC of the PERF ARF is copied into the PC of the fetch unit.

4. Experimental Methodology

The experiments were performed with SPEC CPU2000 benchmarks compiled for the Alpha ISA with the `-fast` optimizations and profiling feedback enabled. We perform our studies with the reference input set by fast-forwarding up to 15 billion instructions and simulating up to 200M instructions.

We evaluate MBI by modeling it in an execution-driven performance simulator. Our baseline processor is an aggressive eight-wide out-of-order superscalar processor that implements the Alpha ISA. Table 1 describes the baseline configuration. Because MBI is primarily targeted towards utilizing the idle cycles during long-latency cache misses, we model the memory system in detail. We faithfully model port contention, bank conflicts, bandwidth, and queuing delays. The cache hierarchy includes a relatively large, 1MB, L2 cache. Because some of the cache misses can be easily prefetched using a prefetcher, our baseline also contains a state-of-the-art streaming prefetcher [18].

Table 1. Baseline configuration.

Instruction cache	16KB, 64B line-size, 4-way with LRU replacement; 8-wide fetch with 2 cycle access latency.
Branch Predictor	64K-entry gshare/64K-entry PAs hybrid with 64K-entry selector; 4K-entry, 4-way BTB; minimum branch misprediction penalty is 24 cycles.
Decode/Issue	8-wide; reservation station contains 128 entries.
Execution units	8 general purpose functional units; All INT instructions, except multiply, take 1 cycle; INT multiply takes 8 cycles. All FP operations, except FP divide, take 4 cycles; FP divide takes 16 cycles.
Data Cache	16KB, 64B line-size, 4-way with LRU replacement, 2-cycle hit latency, 128-entry MSHR.
Unified L2 cache	1MB, 64B line-size, 8-way with LRU replacement, 15-cycle hit latency, 128-entry MSHR.
Memory	400-cycle minimum access latency; 32 banks.
Bus	16B-wide split-transaction bus at 4:1 speed ratio.
Prefetcher	Stream-based prefetcher with 32 stream buffers.

5. Results and Analysis

5.1. Performance Overhead

MBI has two sources of performance overhead. The first source is the latency involved in filling the pipeline after switching from one mode to another. We call this overhead the *pipeline-fill penalty*. The second source is forced-introspection, which forces the processor to perform redundant execution at the expense of performing its normal operation. We call this overhead the *forced-introspection penalty*.

Figure 4 shows the decrease in IPC of the baseline processor when MBI is incorporated. To measure IPC, we only count the instructions retired in performance mode. We measure the IPC overhead for a suite by first calculating the harmonic mean IPC of the baseline for the suite, then calculating the harmonic mean IPC for the suite when MBI is incorporated, and taking the percentage difference between the two harmonic means.

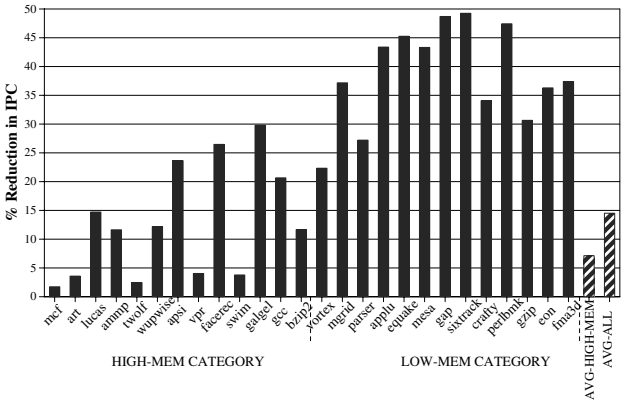

Figure 4. IPC reduction due to the MBI mechanism.

For benchmarks in the HIGH-MEM category, the IPC overhead due to MBI is fairly low, averaging only 7.1%. The average IPC overhead for all the 26 benchmarks is 14.5%. For mcf, art, twolf, vpr and swim, the IPC reduction is well below 5%. These benchmarks are memory intensive and have frequent stalls due to long-latency cache misses. For lucas, ammp, and wupwise, the IPC reduction ranges from 12% to 16%. The bursty L2 cache miss behavior of these three benchmarks results in a high IPC overhead even though these benchmarks spend more than half of their execution time waiting for memory. In some of these benchmarks, the program passes through two phases, P1 and P2, each phase containing many more instructions than the size of the backlog buffer. Phase P1 causes a lot of long-latency cache misses and therefore results in a lot of spare processing bandwidth, whereas phase P2 does not contain any long-latency cache misses. Phases P1 and P2 repeat in an alternating pattern. Even though there is significant spare processing bandwidth during P1, instructions in P2

cannot utilize it for redundant execution and the processor is frequently forced to enter forced-introspection during P2, which causes the reduction in IPC.

For benchmarks in the LOW-MEM category, the performance overhead is high. All benchmarks, with the exception of vortex and parser, incur an IPC reduction of more than 30%. For the LOW-MEM benchmarks when the processor enters introspection mode, it is almost always because the backlog buffer is full. This results in frequent incurrence of the forced-introspection penalty.

5.2. Impact of the Size of the Backlog Buffer

In the previous section, we assumed that the backlog buffer can store the results of 2K instructions. In this section, we analyze the sensitivity of the IPC overhead of MBI to the capacity of the backlog buffer. We vary the size of the backlog buffer from 1K entries to 4K entries and measure the IPC overhead. Figure 5 shows the IPC variations when the size of the backlog buffer is changed.

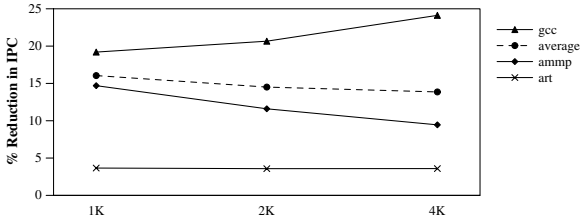

Figure 5. Sensitivity of the IPC overhead to the size of the backlog buffer. Y-axis represents the IPC overhead and X-axis represents the number of entries in the backlog buffer.

The IPC-overhead for ammp decreases from 14.9% to 9.2% as the backlog buffer size is increased from 1K-entry to 4K-entry because a larger backlog buffer can reduce the number of costly forced-introspection episodes. Benchmark art has a steady IPC overhead of 3.6% for all three sizes of the backlog buffer. Art has frequent long-latency cache misses, which obviates the need for a large backlog buffer. Benchmark gcc has a large instruction working-set size, and therefore it shows a slight increase in the IPC overhead when the size of the backlog buffer is increased. A large backlog buffer can increase the I-cache miss rate in introspection mode, which in turn can increase the I-cache miss rate in performance mode (because different instructions may need to be fetched from the I-cache in the two different modes). The average IPC-overhead decreases from 16.1% for a 1K-entry backlog buffer, to 13.9% for a 4K-entry backlog buffer. A backlog buffer of 2K entries provides a good trade-off between IPC overhead (14.5%) and hardware overhead. Therefore, we chose 2K entries as the default size of the backlog buffer throughout our experiments.

5.3. Storage Cost of the MBI Mechanism

The additional hardware required for the MBI mechanism includes the backlog buffer, the extra register file (ISPEC ARF) and combinational logic such as mux and comparators. In this section, we estimate the storage cost of the MBI mechanism. A straightforward, *unoptimized* implementation of the backlog buffer is to store the full instruction results in the backlog buffer entry. We assume that each result value stored in the backlog buffer takes eight bytes. Table 2 calculates the storage cost of the MBI mechanism in terms of Register Bit Equivalents (RBE's).

Table 2. Storage Cost of the MBI Mechanism.

ARF contains: 32 INT Regs	32*8B=256B
32 FP Regs	32*8B=256B
2 CNTL Regs	2*8B= 16B
1 PC	1*8B= 8B
Size of ARF	536B
Backlog buffer contains	2000 entries
Size of each backlog buffer entry	8 B
Size of backlog buffer	2000*8B= 16000B
Total storage cost of MBI	16536B

The storage cost of 16KB is fairly small, especially considering that the baseline contains a 1MB cache. This cost can further be reduced by using compression schemes and taking advantage of result values that require fewer than 8 bytes. It should be noted that the above calculations do not quantify the cost of control logic for the backlog buffer (such as ECC bits) and the cost of glue logic (such as selection mux and comparator). The hardware cost of the control logic is fairly small in comparison to the storage cost tabulated in Table 2. However, quantifying the exact cost of the control logic is beyond the scope of this paper.

5.4. Reasons for Entering Introspection Mode

The processor enters introspection mode either because of a long-latency cache miss (normal-introspection) or because it is forced to enter introspection mode (forced-introspection). An episode of normal-introspection does not cause a significant reduction in performance because it uses idle processing bandwidth for redundant execution. On the other hand, an episode of forced-introspection reduces the performance of the processor because the processor is forced to perform redundant execution at the expense of normal execution. In this section, we present results on the distribution of introspection episodes. Figure 5.4(A) shows the breakdown of the introspection episodes into normal-introspection and forced-introspection for a subset of the studied benchmarks.

For mcf, art, twolf, and vpr, more than 96% of the introspection episodes occur because of normal-introspection. These benchmarks are memory intensive and frequently experience long-latency cache misses. For apsi, approximately half of the introspection episodes occur because of

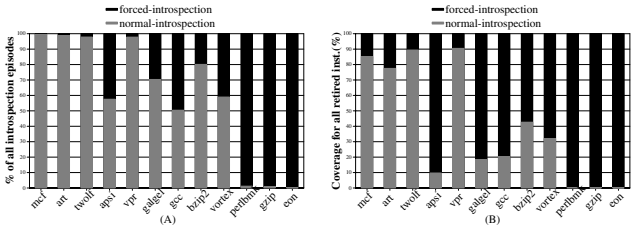

Figure 6. (A) Breakdown of introspection episodes into normal-introspection and forced-introspection. (B) Redundant execution coverage of normal-introspection and forced-introspection.

forced-introspection. Although apsi generates a significant number of long-latency cache misses, these misses tend to be clustered. For perlbmk, gzip, and eon almost all the introspection episodes occur because the backlog buffer is full. These benchmarks do not have a substantial number of long-latency cache misses and therefore require forced-introspection.

When the processor is forced to enter introspection mode because the backlog buffer is full, the processor always performs redundant execution until the backlog buffer becomes empty. In contrast, during normal-introspection, the processor performs redundant execution either until the long-latency cache miss gets serviced or until the backlog buffer becomes empty, whichever is earlier. Thus, a typical forced-introspection episode results in the execution of many more instructions than a typical normal-introspection episode. Figure 5.4(B) shows the redundant-execution coverage of the instruction stream provided by normal-introspection and forced-introspection.

For mcf, 12% of the instructions go through their redundant execution due to forced-introspection. However, this costs only 1.4% in terms of IPC because the benchmark is memory bound. For twolf and vpr, less than 10% of the instructions go through their redundant execution due to forced-introspection. This translates to an IPC reduction of only 3% for these benchmarks. In contrast, for apsi, galgel, and gcc almost 80% of the instructions go through their redundant execution due to forced-introspection. Consequently, these benchmarks incur an IPC overhead of more than 20% (refer to Figure 4). Although these benchmarks have a lot of CPI-L2 (theoretically enough to re-execute the program without any performance loss), this idle processing bandwidth comes in bursts and the MBI technique is not able to exploit it.

5.5. Error Detection Latency

A redundant execution mechanism can have a delay between the first execution and the redundant execution of an instruction. If there is an error in the first execution of an instruction, then this error will not be detected until the instruction completes its redundant execution. The delay between the first execution and the redundant execution determines the error detection latency of the fault tolerance mechanism. MBI has a variable error detection latency. Table 3 shows, for each benchmark, the average and worst-case error detection latency of MBI.

Table 3. Error Detection Latency (in cycles).

HIGH-MEM Benchmarks			LOW-MEM Benchmarks		
Name	Avg	Worst-Case	Name	Avg	Worst-Case
mcf	465	6976	vortex	467	18157
art	304	3552	mgrid	514	17401
lucas	307	7066	parser	957	8593
ammp	500	7829	applu	680	25941
twolf	481	10785	equake	795	5366
wupwise	457	7711	mesa	646	11449
apsi	1143	36183	gap	744	12187
vpr	487	5043	sixtrack	770	20978
facerec	475	9279	crafty	1190	19480
swim	807	16546	perlbmk	832	14330
galgel	790	8372	gzip	1106	7339
gcc	902	20075	eon	877	17954
bzip2	680	4793	fma3d	615	10936
Overall: Average = 692, Worst-Case = 36183					

For all benchmarks except apsi, crafty, and gzip, the average error detection latency is less than 1000 cycles. Over the entire SPEC CPU2000 suite, the average error detection latency is 692 cycles, and the worst-case error detection latency is 36183 cycles. The impact of the error detection latency on the system operation is dependent on the fault handling policy. For example, if the fault handling policy is to terminate the faulting application, then the average error detection latency of 692 cycles is clearly acceptable. However, if the fault-handling policy is to correct the error, then the time it takes to correct the error may increase with the error detection latency. The next section discusses error handling policies.

6. Handling Errors

A fault in the processor can be detected only during introspection mode. When a fault is detected, the faulting instruction is re-executed one more time to ensure that the fault was indeed during the first execution of that instruction. If the result produced during this re-execution matches the result in the backlog buffer, then the fault is ignored. However, if the result of the re-execution does not match the result in the backlog buffer, then the fault is considered an error and is dealt with in accordance with the error handling policy. We discuss some of the error handling policies that can be combined with MBI.

6.1. Fail-Safe Operation

The simplest error handling mechanism is to terminate the error-causing application and generate a machine check

exception. This mechanism avoids any correction but allows the processor to fail in a safe manner.

6.2. Restore a Checkpointed State

Another error handling mechanism, also known as Backward Error Recovery (BER), is based on the concept of checkpointing. Both memory state and processor state are checkpointed at pre-determined intervals. When an error is detected, the system is restored to the most-recent, error-free checkpointed state. The BER scheme provides fast recovery from an error but has a high storage cost. Proposals such as Multiversion memory [16] and ReVive [11] provide cost-effective implementations of the BER mechanism.

6.3. Recovery with Undo Mechanism

The system can recover from an error if it can be restored to a state where the effects of all the instructions prior to the error-causing instruction are architecturally visible and no effects of the error-causing instruction, or any subsequent instruction, are architecturally visible. The MBI mechanism allows a unique error recovery technique that can restore the system to the state of the last error-free instruction. Error recovery is performed by undoing the store operations that were performed during performance mode after the last error-free instruction. Figure 7(a) shows the extensions added to the backlog buffer to facilitate error recovery.

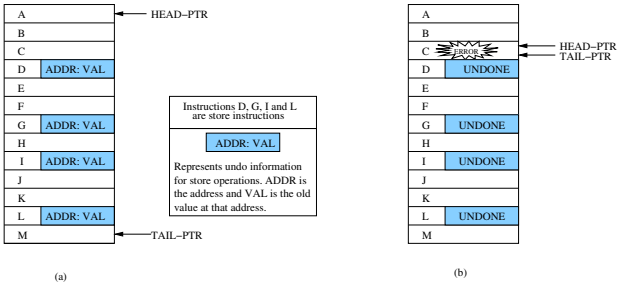

Figure 7. (a) Extending the backlog buffer to keep track of undo information for store instructions. (b) An example of error correction with the undo mechanism.

6.3.1. Structure. The backlog buffer tracks undo information for each store instruction. The undo information consists of the effective address of the store instruction and the old value at that address. In Figure 7(a), instructions labeled D, G, I, and L are store instructions and the shaded regions represent the undo information associated with each store instruction. The backlog buffer does not contain any undo information for non-store instructions.

6.3.2. Operation. Error recovery consist of two parts: (1) recovering the register state, and (2) recovering the memory state. Recovering the register state is relatively easy because the register state in introspection mode is updated only after the instruction is found to be fault-free. Thus, during introspection mode, the ISPEC ARF register file always corresponds to the last instruction that was found to be correct. When an error is detected, the register state of ISPEC ARF is copied to PERF ARF.

Recovery of the memory state is more challenging because, at the time the error is detected, the store instructions younger than the error-causing instruction have already updated the memory. Therefore, memory updates caused by these store instructions must be undone. We explain the undo operation with an example. Figure 7(b) shows the state of the backlog buffer for the instruction sequence A to M. Instructions D, G, I, and L are store instructions and the backlog buffer entries corresponding to the stores contain undo information. While checking the result of instruction C in introspection mode, an error is detected. To recover the memory state, all stores younger than instruction C in the backlog buffer (stores D, G, I, and L) must be undone. To accomplish this, the backlog buffer is traversed backwards starting with the TAIL-PTR. Any store instruction that is encountered during this backward traversal is undone. The undo operation simply copies the old value stored in the backlog buffer entry to the memory address indicated in the backlog buffer entry. Backward traversal of the backlog buffer continues until the the error-causing instruction is reached. At that point, no effect of the error-causing instruction, or any subsequent instruction, is architecturally visible. Thus the system has recovered from the error detected at instruction C.

It should be noted that the undo mechanism assumes that the store values are not read by other devices (such as other processors if a multiprocessor system is used) before the undo process takes place. For multiprocessor systems, the undo mechanism can be used if other processors are not allowed to read the values produced by a store instruction until that instruction is found to be fault-free in introspection mode. An alternative is to transfer the control to software so that the processors that have consumed incorrect values can be recovered to a consistent state. Design requirements of the undo mechanism in multiprocessor systems is outside the scope of this paper and is part of our future work.

6.3.3. Error Correction Latency. The duration between when the error is detected and when the system is restored to an error-free state is called the *error correction latency*. The error correction latency of the undo mechanism is variable and depends on the number of instructions between the HEAD-PTR and TAIL-PTR. In the worst-case where every entry in the backlog buffer is a store instruction, the TAIL-PTR traverses through, and performs undo operation for, every entry in the backlog buffer. Thus, the worst-case error correction latency with a backlog buffer containing 2K entries is as high as 2K undo operations. However, this latency

is very low compared to the time between errors and will not significantly affect the availability of the system. For example, even with an error rate of 1 error/hour, a 5GHz processor that can perform 1 undo operation per cycle will have an availability of 99.99999% ($Availability = (T_E - T_R)/T_E$, where T_E is the mean time between errors, and T_R is the time the machine is not available due to an error).

7. Related Work

Commercial high reliability systems, such as the Tandem Computer [2], the Compaq NonStop Himalaya [22], and the IBM S/390 [17] use lock-step execution for detecting faults in processors. Fault tolerance in these systems is achieved at the expense of hardware replication.

An alternative to replication was proposed by Austin in the form of DIVA [1]. For redundant execution, DIVA uses a simple checker processor after the retirement stage of the main processor. The assumption that the instruction fetch stage and the instruction decode stage of the main processor are fault-free allows the checker processor to use the instruction dependency information computed in the main processor. Unlike DIVA, MBI provides fault coverage for the *entire* pipeline, including the fetch and decode stages. DIVA uses a physically separate processor for redundant execution and can therefore detect both permanent and transient faults, whereas MBI provides coverage only for transient faults and would need to be combined with techniques like RESO [10] to provide coverage for permanent faults. The additional processor in DIVA, although simple, requires considerable hardware overhead compared to MBI. A low hardware overhead is desirable because it provides the users with a choice to use or not to use the features of fault tolerance. With MBI, if the user chooses not to have fault tolerance, then only the hardware solely dedicated to fault tolerance, which is relatively small, will go unused.

Rotenberg [14] proposed AR-SMT, which provides fault tolerance by executing the application using two separate threads. The primary thread inserts its results into a delay buffer and the redundant thread uses these results for speculative execution and fault detection. Both threads run concurrently in the processor pipeline and have different memory images. The approach of using SMT-based machines for fault tolerance was generalized in [13]. Both [14] and [13] require a fine-grain multi-threaded machine capable of concurrently fetching, decoding, and executing from more than one thread. Redundant execution, in both cases, halves the fetch bandwidth, reduces the effective size of the storage structures (e.g. reservation stations, caches) visible to each thread, and increases the contention for execution units. In contrast, operation in MBI is in either performance mode or introspection mode. Therefore, in MBI, redundant execution does not compete with the primary execution for hardware resources at a fine-grain level. The MBI design is also less intrusive than the SMT-based designs because it does not require widespread modifications throughout the processor pipeline.

Both MBI and SMT-based techniques target idle processing bandwidth for fault tolerance. However, they target fundamentally different types of idle processing bandwidth. SMT leverages the fine-grain idle processing slots that remain unused due to limited ILP in each cycle. On the other hand, MBI utilizes the coarse-grain idle processing bandwidth that remains unused due to long-latency cache misses. As such, MBI is well suited for memory-intensive applications. SMT-based techniques, on the other hand, are well suited for applications that are not significantly limited in their performance by long-latency cache misses.

Mendelson and Suri proposed O3RS [7], which provides transient-fault tolerance to only the out-of-order portion of the processor pipeline. After an instruction is renamed, it occupies two entries in the reservation stations. The two results obtained in this manner are compared for fault detection. The mechanism proposed by Ray et al. [12] replicates instructions in the rename stage and provides transient-fault coverage for all stages after the rename stage. Both [7] and [12] assume fault protection for the stages before the rename stage and require extra logic in the processor to handle the simultaneous existence of primary and redundant instructions.

The related fault tolerance techniques described thus far are hardware based. Fault tolerance can also be incorporated with software support. Wu et al. [23] proposed a technique to arrange the code such that redundant instructions are statically mapped to use empty slots in execution units. However, only applications with regular code behavior and with latencies predictable at compile-time lend themselves to static scheduling. The mechanism proposed in [9] executes two different versions of the same program (with the same functionality) and compares the outputs to detect transient and some permanent faults.

A transient fault may or may not cause an error depending on whether it affects the final outcome of the program. A study of the effects of transient faults on the performance of a superscalar processor is provided in [19]. Weaver et al. [20] describe a mechanism to avoid signaling errors that occur during the processing of dynamically dead instructions. All retired instructions are inserted into a FIFO buffer. A faulty instruction is marked as possibly incorrect before insertion into the FIFO buffer. Faults in a possibly incorrect instruction are ignored if the instruction is found to be dynamically dead while it is in the FIFO buffer. Currently, the MBI mechanism detects errors for both dynamically live and dynamically dead instructions. However, the technique proposed in [20] can easily be incorporated in the MBI mechanism by making minor modifications to the backlog buffer.

8. Conclusion and Future Work

Future processors will need on-chip fault tolerance techniques to tolerate the increasing transient fault rate. Future processors will also be challenged by the speed gap between the processor and the memory and will waste significant processing bandwidth waiting for memory. Based on these observations, this paper makes the following two contributions:

(1) A transient-fault detection technique, Microarchitecture-Based Introspection (MBI), which uses the wasted processing bandwidth during long-latency cache misses for redundant execution. This technique has small hardware cost and provides redundant execution coverage for the entire pipeline (from instruction fetch to retirement).

(2) A fault recovery scheme for MBI that has a negligible effect on system availability.

The time redundancy of MBI results in an average IPC reduction of only 7.1% for memory-intensive benchmarks and an average IPC reduction of 14.5% over the entire SPEC CPU2000 suite.

MBI can be combined with runahead execution [8] to improve both the reliability and the performance of memory-intensive applications. MBI can also be combined with SMT to utilize both fine-grain and coarse-grain idle processing bandwidth for redundant execution. Exploring these hybrid mechanisms is a part of our future work.

Acknowledgments

We thank Pradip Bose for the early discussions and continued feedback on this work. We also thank Sanjay Patel, the anonymous reviewers, and the members of the HPS research group for their helpful comments. This work was supported by gifts from IBM, Intel, and the Cockrell foundation. Moinuddin Qureshi is supported by an IBM fellowship. Onur Mutlu is supported by an Intel fellowship.

References

[1] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 196–207, 1999.

[2] J. Bartlett, J. Gray, and B. Horst. Fault tolerance in Tandem computer systems. Technical Report 86.2, Tandem Computers, Mar. 1986.

[3] R. Hankins, T. Diep, M. Annavaram, B. Hirano, H. Eri, H. Nueckel, and J. Shen. Scaling and characterizing database workloads: Bridging the gap between research and practice. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 151–163, 2003.

[4] W. W. Hsu, A. J. Smith, and H. C. Young. Characteristics of production database workloads and the TPC benchmarks. *IBM Journal of Research and Development*, 40(3):781–802, Mar. 2001.

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

[6] H. Li, Chen-Yong Cher, T. N. Vijaykumar, and K. Roy. VSV: L2-miss-driven variable supply-voltage scaling for low power. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 19–28, 2003.

[7] A. Mendelson and N. Suri. Designing high-performance and reliable superscalar architectures: The out of order reliable superscalar (o3rs) approach. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2000.

[8] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the Ninth IEEE International Symposium on High Performance Computer Architecture*, pages 129–140, 2003.

[9] N. Oh, S. Mitra, and E. J. McCluskey. ED4I: Error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers*, 51(2):180–199, Feb. 2002.

[10] J. H. Patel and L. Y. Fung. Concurrent Error Detection in ALUs by REcomputing with Shifted Operands. *IEEE Transactions on Computers*, 31(7):589–595, July 1982.

[11] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 111–122, 2002.

[12] J. Ray, J. C. Hoe, and B. Falsafi . Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 214–224, 2001.

[13] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, 2000.

[14] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pages 84–91, 1999.

[15] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 389–398, 2002.

[16] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Fast checkpoint/recovery to support kilo-instruction speculation and hardware fault tolerance. In *Dept. of Computer Sciences Technical Report CS-TR-2000-1420, October 2000*.

[17] T. J. Slegal et al. IBM's S/390 G5 Microprocessor Design. *IEEE micro*, pages 12–23, Mar. 1999.

[18] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.

[19] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 61–70, 2004.

[20] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31th Annual International Symposium on Computer Architecture*, pages 264–273, 2004.

[21] M. V. Wilkes. The memory gap and the future of high performance memories. *ACM Computer Architecture News*, 29(1):2–7, Mar. 2001.

[22] A. Wood. Data integrity concepts, features, and technology. *White Paper, Tandem division, Compaq Computer Corporation*.

[23] K. Wu and R. Karri. Selectively breaking data dependences to improve the utilization of idle cycles in algorithm level re-computing data paths. *IEEE Transactions on Reliability*, 52(4):501–511, Dec. 2003.