Coverage of a Microarchitecture-level Fault Check Regimen in a Superscalar Processor

Vimal Reddy
Qualcomm Inc.
vreddy@qualcomm.com

Eric Rotenberg
North Carolina State University
ericro@ece.ncsu.edu

Abstract

Conventional processor fault tolerance based on time/space redundancy is robust but prohibitively expensive for commodity processors. This paper explores an unconventional approach to designing a cost-effective fault-tolerant superscalar processor. The idea is to engage a regimen of microarchitecture-level fault checks. A few simple microarchitecture-level fault checks can detect many arbitrary faults in large units, by observing microarchitecture-level behavior and anomalies in this behavior. Previously, we separately proposed checks for the fetch and decode stages, rename stage, and issue stage of a contemporary superscalar processor. While each piece hinted at the possibility of a complete regimen – for an overall fault-tolerant superscalar processor – this totality was not explored. This paper provides the culmination by building a full regimen into a superscalar processor. We show for the first time that the regimen-based approach provides substantial coverage of an entire superscalar processor. Analysis reveals vulnerable areas which should be the focus for regimen additions.

1. Introduction

Conventional approaches to processor fault tolerance use space or time redundancy, providing robust fault tolerance but incurring high costs (in terms of performance, area, and power). Explicit redundancy is suitable for high-end computing systems, but may not be viable in commodity systems. These systems demand a more cost-effective fault tolerance solution, that provides less coverage than explicit duplication but substantial coverage nonetheless.

This paper explores a new, unconventional approach to designing a cost-effective fault-tolerant superscalar processor. The idea is to engage a regimen of microarchitecture-level fault checks. A microarchitecture-level fault check indirectly and broadly detects low-level transient faults, by observing the microarchitecture-level anomalies they cause. Thereby, a few simple checks can detect many arbitrary faults in large units.

In prior work, we separately proposed checks for covering the fetch and decode stages [1], the rename stage, and the issue stage of a contemporary superscalar processor [2]. While each piece hinted at the possibility of a complete regimen – for an overall fault-tolerant superscalar processor – this totality was not explored or evaluated. This paper provides the culmination by building a full regimen into a superscalar processor, and thoroughly evaluating the coverage offered by a regimen-based approach through extensive fault-injection experiments.

This paper makes the following key contributions:

- We describe an overall microarchitecture-level fault check regimen. The regimen is a composition of our previous separate microarchitecture-level fault checks [1,2], key improvements to these checks, and some new checks.
- We develop a high-level fault injection strategy for use with cycle-accurate simulators. All pipeline stages of a contemporary superscalar processor are analyzed to understand how faults are ultimately manifested. This analysis is used to enumerate a list of faults to inject in the cycle-accurate simulator. The enumerated faults achieve high fault modeling coverage of the pipeline, without resorting to RTL or gate-level fault injection.
- We demonstrate for the first time that a microarchitecture-level fault check regimen can provide substantial coverage of an overall superscalar processor pipeline. Extensive fault injection experiments show that the regimen detects 83% of non-masked faults, on average. This confirmation could only be obtained by assembling a full regimen and injecting a broad spectrum of fault types throughout the pipeline, whereas our precursor work used rather localized and targeted fault injection. For example, our holistic study exposes cases where faults are detected by checks in ways that were not anticipated or intended.

- We present detailed breakdowns of fault outcomes and fault checks, (i) across all pipestages, (ii) per pipestage, and (iii) per fault type within pipestages. These results provide insight into which faults are chief culprits for which fault outcomes, and which fault checks detect them. The results also identify remaining vulnerable areas of the regimen, for guiding regimen additions.

2. Fault check regimen

This section describes the checks that make up the fault check regimen used in this paper.

2.1. Inherent Time Redundancy (ITR)

Inherent time redundancy (ITR) exploits program repetition to detect faults in decode signals [1]. Decode signals of instruction traces are combined into signatures and stored in an ITR cache. When an instruction trace repeats, its signature is re-created and checked with the ITR cache for a match. Misses do not directly lead to loss in fault detection coverage, because faults in instruction traces that miss can be detected by future hits to their signatures. In the previous study [1], we showed that an ITR cache can effectively detect faults in the fetch and decode stages of the pipeline.

In this paper, we make ITR more effective by moving the ITR cache to the retirement stage. This extends fault coverage to decode signals across all pipeline stages, in addition to protecting the fetch and decode units. The pipeline is modified such that, when certain decode bits are used for the last time and discarded, they update a signature which flows with the instruction. When instructions drain from the pipeline, their signatures are written into the reorder buffer (ROB). Finally, at retirement, the ITR cache is accessed to compare instruction trace signatures for faults.

2.2. Register Name Authentication (RNA)

Register Name Authentication (RNA) [2] exploits redundancy among renaming structures used in out-of-order superscalar processors, to detect faults in the destination register mappings of instructions. RNA includes two checks, the previous mapping check (referred to as RNA1) and the writeback state check (RNA2). RNA1 is based on the insight that, when an instruction’s logical destination register is renamed, the previous physical register mapping in the rename map table corresponds to the previous producer of that logical destination register, and the same previous physical register mapping should be in the architectural map table when the instruction commits. By comparing the previous physical register mapping recorded at the register rename stage to the corresponding mapping in the architectural map table at the commit stage, RNA1 can detect faults in several rename structures: the rename map table, architectural map table, shadow map tables (branch checkpoints), and the previous and current mapping fields of the ROB. However, RNA1 cannot detect faults in which an erroneous mapping appears consistent among all the structures. Faults in the free list and the destination renaming logic (which assigns new mappings from the free list) fall into this class. Nor can RNA1 detect faults in instructions’ destination register mappings after the dispatch stage, as these mappings fall outside the scope of the overall renaming logic.

RNA2 aims to detect such faults using the insight that they cause invalid register conflicts between instructions. Basically, an erroneously mapped physical register might already be in use by another active instruction, committed to architectural state, or still available in the free list. Asserting the state of a physical register at the register writeback stage (confirming that it is not already ready and not in the free list) exposes conflicts, hence, detects these faults.

2.3. Timestamp-based Assertion Check (TAC)

Timestamp-based Assertion Checking (TAC) [2] exploits the insight that an instruction should execute only after all of its producers have executed. This invariant is true even in an out-of-order superscalar processor, where instructions that do not depend on each other issue in parallel or out-of-order. To confirm time-orderliness within a data dependence chain, TAC assigns timestamps to instructions when they issue, and compares timestamps in the retirement stage to assert that instructions issued only after their producers. The out-of-order scheduler is comprised of complex hardware structures for waking up and selecting ready instructions for issuing. Transient faults in these structures or any associated logic can cause instructions to issue prematurely. TAC detects all of these faults with one simple assertion check.

2.4. Sequential PC Check (SPC)

With the SPC check [1], the idea is to maintain a retirement program counter (PC) and assert that a committing instruction’s PC matches the retirement PC. The retirement PC is updated as follows. Non-branch instructions add their length (which can be recorded at decode for variable-length ISAs) to the retirement PC and branches update the retirement PC with their calculated PC. Comparing a committing instruction’s PC with the retirement PC will detect a discontinuity between two otherwise sequential instructions. The SPC check can detect faults that affect sequential control-flow, for example, faults on the PC or a ROB bookkeeping fault that might cause some valid instructions in the ROB to be overwritten.

2.5. Register Consumer Counter (CC)

Source register mappings originate in the source rename logic and the rename map table (of which the shadow map tables are an extension), and propagate with the instruction after renaming.

The consumer counter (CC) check – newly proposed in this paper – aims to detect faults in source mappings *after* renaming. The CC check detects such faults by maintaining a counter per physical register, which indicates the number of unissued consumers of the physical register, and asserting that the consumer counter is non-zero when the register is read. The counter is incremented in the rename stage, when source mappings are initially determined. In the register read stage, it is asserted that the consumer counter of a register being read is non-zero, following which, the counter is decremented. Squashing a consumer, like issuing a consumer, causes its source registers' counters to be decremented as well.

If a source mapping in the rename map table (or a shadow map table) is modified by a transient fault, the now faulty source mapping will be detected by RNA1, albeit after a consumer of the faulty mapping commits. Specifically, when the next producer of the same logical register commits, its recorded previous mapping will differ from the corresponding mapping in the architectural map table.

Neither the CC check nor RNA1 check can detect faults in the source rename logic, however. In this case, the wrong counter is incremented and decremented consistently (CC) and the previous mapping recorded by the next producer is correct (RNA1). Such faults can be detected by re-renaming instructions in the retirement stage, using the architectural map table, which should yield the same source mappings as the rename stage unless there was a fault in either the source renaming logic or the source re-renaming logic. The extra cost of re-renaming is moderate considering that it can be embedded in the cost of the TAC check, which reads timestamps of logical source registers from the architectural map table. We did not include re-renaming in the regimen, however. It is left for future work.

Finally, note that some source mapping faults are ultimately detected by TAC (if the source mapping links a consumer to a wrong producer that executes before the right producer) or the watchdog timer (if the source mapping causes the consumer to never issue).

2.6. Confident Branch Misprediction (ConfBr)

Confidently-predicted branches have been shown to be useful for detecting transient faults [9,10]. Mispredictions among confidently-predicted branches are considered symptoms of transient faults. Upon detecting the misprediction of a confidently-predicted branch, the pipeline is flushed and restarted from the instruction at the head of the ROB. The ConfBr check detects some faults affecting uncommitted (speculative) values that directly or indirectly feed into a branch.

2.7. BTB Verify (BTBV)

The BTB Verify (BTBV) check, newly proposed in this paper, exploits inherent redundancy between the fetch and decode stages.

The BTB (branch target buffer) in the fetch stage is used to identify branches and provide their taken target addresses, earlier than the decode stage (for fast next-PC prediction). If there is a BTB miss, the branch is decoded one or a few cycles later in the decode stage, providing the same information, only late.

We leverage the inherent redundancy between the branch information generated by a BTB hit and the branch information generated a cycle later in the decode stage. They should be consistent. If not, there is a fault in either the BTB logic or the decode logic.

3. Evaluating coverage

3.1. Fault injection strategy

A particle can flip a bit stored in a latch, flip-flop, or SRAM cell, or cause a transient pulse in a net that might lead to incorrect outputs from combinational logic blocks which in turn may get latched. Directly modeling these low-level faults requires a gate-level implementation of the processor. Although this method is highly accurate, it has several drawbacks. First, a gate-level model may not be available early in the design. Second, fault simulation at the gate level is very time consuming. Compounding this problem is the need for long observation windows following injection of a fault, to determine if the fault is eventually detected by microarchitecture-level checks. Third, faults injected at the gate level are often masked at the logic level or architectural level [3,4,5]. While inherent fault masking should be part of an accurate reliability estimate, it is not useful in evaluating the effectiveness of a fault-checking regimen.

Therefore, it is desirable to have a reasonably accurate fault injection strategy that can be used with a C/C++ cycle-accurate microarchitecture simulator, which we refer to as a timing simulator throughout. The challenge in using a timing simulator for fault injection is ensuring that the modeled faults comprehensively cover low level faults. We take a new approach to ensure high fault modeling coverage. The idea is to characterize the high level effects of low level faults in each pipeline stage, and aggregate many low level faults into fewer high level fault manifestations that can be modeled in a timing simulator.

In hierarchical simulation [6], the effects of low level faults are propagated to higher levels of abstraction using fault dictionaries. Essentially, our novel contribution is a fault dictionary for use in a timing simulator of a superscalar pipeline, derived from a manual and comprehensive analysis of the pipeline.

Section 3.2 presents fault analysis of a detailed 2-way superscalar pipeline model. Each pipeline stage is thoroughly analyzed to aggregate faults into manifestations which can be modeled in a timing simulator. The outcome of this exercise is a comprehensive list of high level faults in each pipeline stage.

3.2. Fault analysis of a superscalar pipeline

A typical superscalar processor pipeline is comprised of several stages that process instructions in parallel, as depicted in Figure 1 (a) and (b). The instruction fetch (IF) stage predicts branches and fetches instructions from the instruction cache. The instruction decode stage (ID) generates decode signals for processing the instruction. The rename stage (REN) eliminates output and anti-dependencies in the static program representation. The dispatch stage (DISP) inserts instructions into the reservation stations (i.e., the issue queue) and the reorder buffer, and also the load/store queues, in the case of memory instructions. The issue stage (IS) dynamically schedules instructions for execution based on data availability and issue bandwidth. After being issued, instructions read source values from the register file in the register read stage (RR) and begin executing in the execution stage (EX). Loads and stores go through the address generation (AGEN) stage instead of the EX stage, followed by disambiguation (store-load dependence checking), store-to-load forwarding, and data cache access (M). After execution or data cache access, instructions write their results into the register file and bypass them to dependent instructions in the writeback stage (WB). Instructions are finally retired in original program order from the reorder buffer in the retirement stage (RE).

IF	ID	REN	DISP	IS	RR	EX	WB	RE

(a)

IF	ID	REN	DISP	IS	RR	AGEN	M	WB	RE

(b)

Figure 1. Superscalar processor pipeline for (a) non-memory and (b) memory instructions.

For fault analysis, each pipeline stage is examined to see how low level faults in that stage would manifest. The goal is to aggregate as many faults into as few manifestations as possible, without losing fault modeling coverage of the pipeline.

3.2.1. Example: fault analysis of fetch & decode. Due to limited space, we only describe fault analysis of the fetch and decode stages of the pipeline, as an example. For comprehensive analysis of all stages, the reader is referred to the corresponding Ph.D. thesis [7].

Figure 2 is a detailed illustration of the fetch and decode stages of a 2-way superscalar pipeline. There are three fetch stages, Fetch0, Fetch1 and Fetch2. In Fetch0, the instruction cache unit is accessed using the program counter (PC). The instruction cache unit (ICU) is comprised of the instruction translation lookaside buffer (I-TLB), instruction cache, and instruction alignment logic. The outputs of the ICU are a maximum of two instructions, the fetch bandwidth of the example processor. In some cases, the output could be only one or no instructions. To signify the scenario, instructions are marked with valid bits (v1 and v2). In parallel with accessing the ICU, the PC is also fed to the Next PC Prediction Unit for predicting the PC for the next cycle. The prediction is based on inputs from a branch target buffer (BTB), conditional branch predictor, and return address stack (these are not explicitly shown in the figure). The predicted next PC is fed back as a candidate PC for the next cycle.

The Fetch1 stage checks misfetches due to wrong branch information from the Next PC Prediction Unit. Inputs to Fetch1 are the instruction packet from the Fetch0 stage, BTB information that was assumed by the Next PC Prediction Unit to predict the next PC (in the case of a BTB miss, the Next PC Prediction Unit assumes that the fetched packet contains no branches), and the predicted next PC itself. The fetched packet is decoded to extract branch information, which is then compared with the BTB information used by the Next PC Prediction Unit. On detecting an inconsistency, the Fetch0 stage is redirected to the override PC, and instructions in the Fetch0 stage are flushed. The Next PC Prediction Unit is also updated with correct information (not shown in the figure).

In the Fetch2 stage, the fetched instructions are entered into a fetch queue. The fetch queue decouples the decode unit from the fetch unit. The FetchQ Allocator tracks the number of entries in the fetch queue, and if space is available, enters the new instructions at the tail of the queue (tail and tail+1). The FetchQ Deallocator reads instructions from the head of the fetch queue (head and head+1) and provides them to the Decode stage.

In the Decode stage, instructions are fully decoded to establish signals that govern the processing of the instructions in later pipeline stages.

In Figure 2, the numbered circles indicate the fault manifestations modeled in the fetch and decode stages.

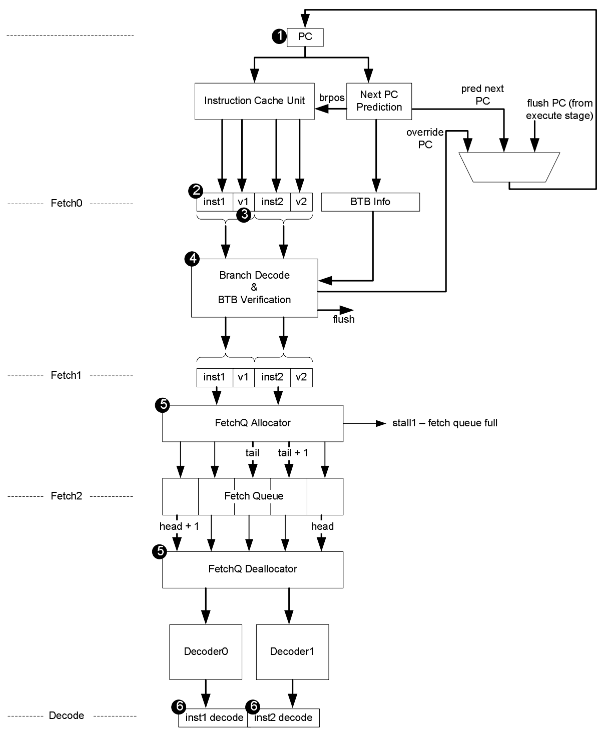


Figure 2. Fault analysis of the fetch and decode stages.

(1) corresponds to the PC. Many faults in the fetch unit ultimately manifest as a faulty PC. This includes faults in the Next PC Prediction Unit, selection of a wrong PC due to faulty signals to the PC mux, and faults in the PC register itself. This manifestation is modeled in the simulator by flipping a random bit in the PC.

(2) corresponds to the instruction stream returned by the ICU. Many faults in the ICU manifest as a wrong instruction stream. This includes faults affecting the I-cache access, instruction alignment, and branch location. This manifestation is modeled by arbitrarily masking off instructions from the fetched packet.

(3) corresponds to instruction validity. If valid bits are faulty, bad instructions can appear or good instructions can disappear from the pipeline. This is similar in spirit to (2), but is modeled by applying it to only one instruction.

(4) corresponds to the Override PC inferred by the branch pre-decode stage (Fetch1) to confirm BTB predictions made in the Fetch0 stage. Faults in the branch pre-decode/BTB verification stage manifest as an incorrectly inferred Override PC. This includes faults that cause false detection of a branch, overlooking of a branch, an incorrect branch position in the instruction packet, or an incorrect branch target address. This manifestation is modeled by flipping a random bit in the inferred Override PC.

(5) corresponds to the fetch queue. Many faults in the fetch queue manifest as reading and writing wrong entries. This includes faults in allocation and de-allocation of entries, bookkeeping, and stall signals. This manifestation is modeled by flipping a bit in the fetch queue head or tail pointers.

(6) corresponds to decode signals. Many faults in the fetch and decode stages manifest as the production of wrong decode signals. These include faults in the registers holding instructions, the decode logic, and the Decode stage's output registers. These are modeled by flipping an arbitrary bit in one of the many decode signals, shown in Table 1.

3.2.2. Fault analysis of all pipeline stages. Similar analysis was performed for all pipeline stages [7], and the resulting list of fault manifestations is shown in Table 1.

4. Methodology

The faults identified in Section 3.2 are modeled in a timing simulator. The simulator models a microarchitecture similar to the MIPS R10000 [8] and outlined in Section 3.2. Key parameters of the microarchitecture are: 4-way superscalar pipeline with 64-entry ROB; 64KB 4-way set-associative L1 instruction and data caches; 1 MB 4-way set-associative unified L2 cache with 10-cycle hit time and 100-cycle miss time; 2^{16}-entry gshare branch predictor with confidence threshold of 64. The ITR cache is 16KB (1,024 entries) 2-way set-associative.

A subset of the SPEC2K benchmark suite is used for evaluation. For each benchmark, one thousand faults are randomly injected. Fault injection involves randomly selecting a fault to inject from the list of faults. A separate "golden" (fault-free) simulator is run in parallel with the faulty timing simulator. When an instruction is committed to the architectural state in the timing simulator, it is compared with its golden counterpart to determine whether or not the architectural state is being corrupted. Any fault that leads to corruption of architectural state is classified as a potential silent data corruption (SDC) fault. Likewise, if no corruption of architectural state is observed for a set period of time after a fault is injected (the observation window), it is classified as a masked fault. In this study, an observation window of one million cycles is used. A watchdog timer (shown as WDOG in the results) is included in the experiments to check for deadlocks.

An injected fault leads to one of many possible outcomes, based on the combination of (1) the effect of the fault on the application, and (2) whether or not the fault is detected by the fault check regimen and how. Figure 3 shows the possible fault outcomes as a chart.

Table 1. Table of faults for all pipeline stages.

Pipe Stage	Fault	Description
Fetch	FETCH_PC	Flip a random bit in the program counter
Fetch	WRONG_INSTR	Remove an arbitrary number of fetched instructions
Fetch	NEXT_PC	Flip a random bit in the override PC from the branch pre-decode/BTB verification stage
Fetch	INSTR_DISAPP	Mask a randomly selected instruction from fetched instructions
Fetch	FETCHQ	Flip a randomly selected bit in the tail/head pointer of the fetch queue
Decode	OPCODE	Flip a random bit in an instruction's opcode
Decode	FLAGS	Flip a random bit in an instruction's decode flags
Decode	SHAMT	Flip a random bit in an instruction's logical/arithmetic shift quantity
Decode	SRC_LOG_REG	Flip a random bit in an instruction's logical source register specifier
Decode	SRCA_LOG_REG	Flip a random bit in an instruction's logical address source register specifier
Decode	RDST_LOG_REG	Flip a random bit in an instruction's logical destination register specifier
Decode	LAT	Flip a random bit in an instruction's latency
Decode	IMM	Flip a random bit in an instruction's signed immediate value field
Decode	UIMM	Flip a random bit in an instruction's unsigned immediate value field
Decode	TARG	Flip a random bit in an instruction's branch target address
Decode	NUM_RSRC	Flip a random bit in an instruction's source operand count
Decode	NUM_RSRCA	Flip a random bit in an instruction's source operand count, address operand
Decode	NUM_RDST	Flip a random bit in an instruction's destination operand count
Decode	IS_DECISION	Flip the bit which indicates whether an instruction is a control-flow decision instruction
Decode	LEFT	Flip the bit indicating left shift of data (LWL/SWLinstructions)
Decode	RIGHT	Flip the bit indicating right shift of data LWR/SWR instructions)
Decode	SIZE	Flip a random bit indicating the size of data (load/store instructions)
Rename	REN_MAP_TABLE	Flip a random bit of a random mapping in the rename map table
Rename	ARCH_MAP_TABLE	Flip a random bit of a random mapping in the architecture map table
Rename	SHADOW_MAP_TABLE	Flip a random bit of a random mapping in a shadow map table
Rename	FREE_LIST	Flip a random bit of an entry in the physical register free list
Rename	FREE_LIST_TAIL	Flip a random bit of the physical register free list's tail pointer
Rename	CHKPT	Randomly pick a shadow map table and flip its availability (used-->free)
Rename	REN_MAP_DEST_INDEX	Flip a random bit in the index used to write a new mapping to the rename map table
Rename	REN_MAP_SRC_INDEX	Flip a random bit in the index used to read a source mapping from the rename map table
Rename	REN_MAP_OLD_DEST_INDEX	Flip a random bit in the index used to read the old register mapping from the rename map table
Rename	DEST_PHYS_REG	Flip a random bit in the destination physical register mapping carried by an instruction
Rename	SRC_PHYS_REG	Flip a random bit in the source physical register mapping carried by an instruction
Rename	OLD_DEST_PHYS_REG	Flip a random bit in the old destination physical register mapping carried by an instruction
Dispatch	ROB_WRITE	Flip a random bit in the tail pointer to the reorder buffer causing a write to a wrong entry
Dispatch	RS_WRITE	Overwrite a randomly selected (occupied) entry in the reservation station (RS)
Dispatch	LSQ_WRITE	Flip a random bit in the tail pointer to the Load Store Queue (LSQ) causing a write to a wrong entry
Backend	READY_BIT	Flip the ready bit of a random physical register from 0 to 1, causing its dependents to execute prematurely
Backend	SPEC_LOAD	Flip a cancel signal sent by a speculatively issued load that misses in the data cache, causing dependents to execute with wrong data
Backend	WAKEUP_TAG	Flip a random bit in a physical register tag that is broadcasted to wakeup dependent instructions
Backend	ROB_ID	Flip a random bit in the ROB index stored with an instruction, causing it to index a wrong ROB entry
Backend	LSQ_TAG	Flip a random bit in the LSQ index stored with an instruction, causing it to index a wrong LSQ entry
Backend	SRCA_VALUE	Flip a random bit in an instruction's effective address (load/store instr.)
Backend	SRC_VALUE	Flip a random bit in an instruction's source value
Backend	DST_VALUE	Flip a random bit in an instruction's computed destination value
Backend	DCACHE_VALUE	Flip a random bit in a load/store instruction's value (going to or coming from the data cache)
Backend	COMPLETE_BIT	Flip an instruction's completion status in the ROB to true before it completes

A fault is shown in a black box, its effects on an application in grey boxes, and the final outcomes in white boxes. The possible effects of a fault on an application are, 1) control-flow deviation (CFD), 2) silent data corruption (SDC), 3) application deadlock (Deadlock/WDOG), and 4) masking of the fault (Mask), i.e., it does not have any of the previous effects. If an injected fault is detected, the fault outcome is indicated by prefixing the effect of the fault with the letter 'A', signifying an assertion. If an injected fault is undetected, the fault outcome is indicated by prefixing the effect of the fault with the letter 'U'. Based on this, the list of possible fault injection outcomes are (also shown in Figure 3):

- ACFD: The fault caused a control-flow deviation, and was detected by the fault-checking regimen.
- UCFD: The fault caused a control-flow deviation, and was not detected by the fault-checking regimen.
- ASDC: The fault caused a silent data corruption, and was detected by the fault-checking regimen.
- USDC: The fault caused a silent data corruption, and was not detected by the fault-checking regimen.
- AWDOG: The fault caused a deadlock, but was detected by the fault-checking regimen before the watchdog timeout occurred.
- UWDOG: The fault caused a deadlock, and was not detected by the fault-checking regimen before the watchdog timeout occurred.
- USDCWDOG: The fault caused a silent data corruption and then a deadlock, and was not detected by the fault-checking regimen.
- AMASK: The fault was architecturally masked, and yet was detected by the fault-checking regimen.
- UMASK: The fault was architecturally masked, and was not detected by the fault checking regimen.

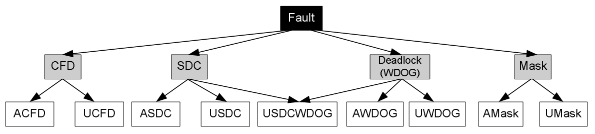


Figure 3. Classification chart for fault injection outcomes.

5. Results

5.1. Distribution of injected faults

Figure 4 shows the distribution of injected faults across all pipeline stages. Across benchmarks, the fault distribution is fairly uniform. On average, the percentage of faults injected in each stage are: fetch – 9%, decode – 39%, rename – 24%, dispatch – 7%, and backend stages – 21%. Note that the 'decode' category in Figure 4 includes faults injected in all pipeline stages where decode bits are used, not just the decode stage itself (this is not reflected in Table 1).

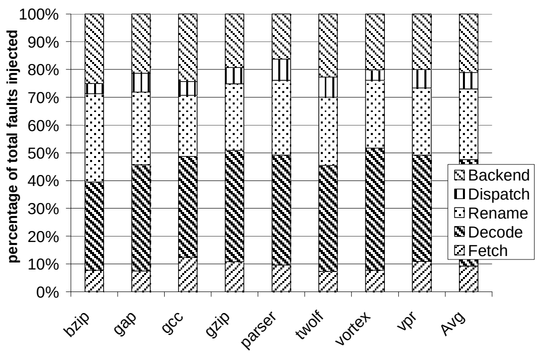


Figure 4. Distribution of injected faults.

Because every fault has the same random chance of being injected, faults are distributed in proportion to the number of faults modeled in each stage. A pipeline stage that models more faults has a larger fraction of faults injected in it. We are currently developing a synthesizable verilog model of the superscalar pipeline to assign non-uniform weights to faults, weighting each pipeline stage based on its total area, its total flip-flop count, or combinations of these. With a TSMC 180nm technology, using areas for weights yields an average fault distribution as follows: fetch – 47% (includes BTB but not I-cache), decode – 22%, rename – 8%, dispatch – 1%, backend – 22%. This area-weighted fault distribution yields similar results to the distribution above. Most of the results in this section are not sensitive to fault distribution: fault outcomes per fault type and fault checks per fault type are influenced by fault *type*. The distribution of fault outcomes, which includes coverage of unmasked faults by the regimen, is influenced by the distribution of injected faults, but even in this case the coverage results hold for the area-weighted distributions (81% reduction in vulnerability compared to 83%).

5.2. Distribution of fault outcomes

Figure 5 shows the overall fault outcome distribution. The distribution pattern across all benchmarks is fairly uniform, and discussion is focused on average results. The average breakdown of fault outcomes is as follows: those detected by the regimen is 60%, those not detected by the regimen but detected by the watchdog timer is 9%, and those undetected is 31%. Several interesting conclusions can be drawn from the results. Around 40% of faults (sum of all CFD, SDC, and WDOG causing faults) cause harm to the application being run, either by corrupting the architectural state, committing wrong-path instructions, or causing a deadlock. This fairly large percentage is motivation to protect processors from transient faults. Among all faults detected by the fault-checking regimen (ACFD + ASDC + AWDOG + AMASK), the part that causes harmful effects is 24% (ACFD + ASDC + AWDOG). So, with the fault-checking regimen in place, the processor is vulnerable to only 40 – 24 = 16% of faults, a 60% reduction in vulnerability. If a watchdog timer is included in the processor, then an additional 9% of faults (those that cause deadlocks) can be detected through timeouts (UWDOG + USDCWDOG). This further reduces vulnerability to harmful faults from 16% to 7%. The overall reduction in vulnerability to harmful faults by combining the fault-checking regimen with the watchdog timer is 40% to 7%, an 83% reduction. This is a substantial result for considering a regimen-based approach to processor fault tolerance.

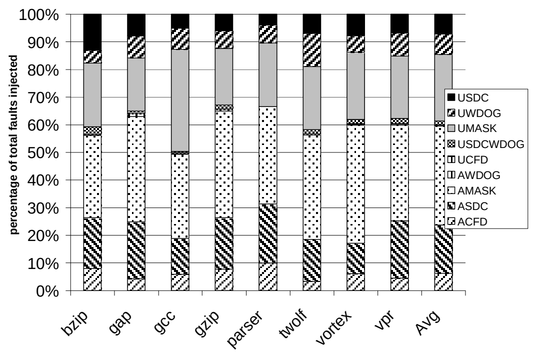


Figure 5. Distribution of fault outcomes.

5.3. Distributions per pipeline stage

To further analyze fault outcomes, the outcome distribution in each pipeline stage is considered, as shown in Figure 6. This provides a high-level reference to analyze fault outcomes, showing the pipeline stage(s) where a given fault outcome tends to occur. It will be used as the first step in investigating a fault outcome, followed by looking at a further breakdown of fault outcomes per fault type injected in a pipeline stage, which are shown in Figure 7 through Figure 11 (left side, marked as (a)). To provide more insight into fault detection, a breakdown of fault checks per fault type in a pipeline stage is shown in Figure 7 through Figure 11 (right side, marked as (b)).

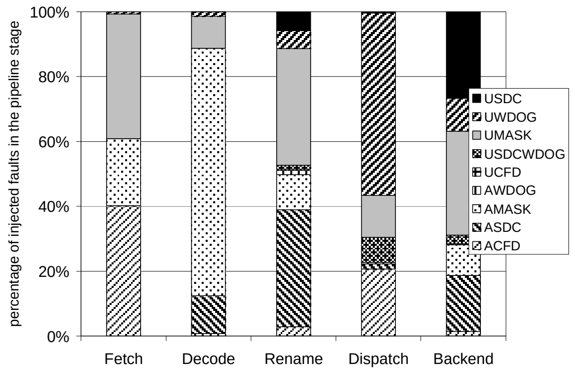
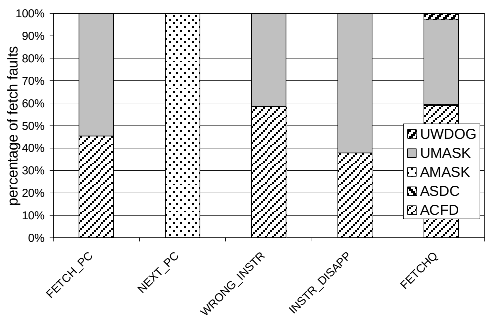


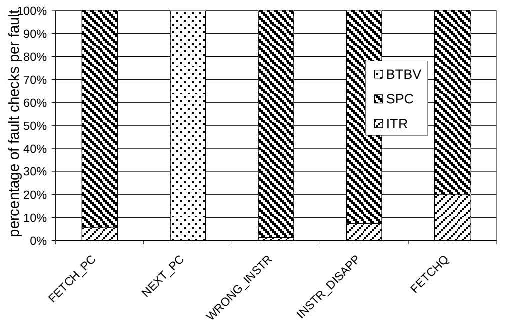
Figure 6. Distribution of fault outcomes per pipe stage.

5.3.1. CFD outcomes. As seen in Figure 6, fault outcomes that cause control-flow deviation (*CFD) mainly occur in the fetch and dispatch stages, with minor incidences in the decode, rename and backend stages. For insight, we refer to the fault outcome breakdown per fault type in the various pipeline stages. Looking at Figure 7(a) (for fetch), we see that four among the five fault types (FETCH_PC, WRONG_INSTR, INSTR_DISAPP, and FETCHQ) are capable of causing invalid breaks in control-flow. As expected, instances of these four fault types are the main contributors to causing a control-flow deviation (see ACFD contribution). All instances of the predicted next PC fault (NEXT_PC) get detected early in the pipeline by the BTBV check. Looking at Figure 10(a) (for dispatch), we find that only the ROB_WRITE fault type contributes to CFD. This is expected, because writing to wrong ROB entries, breaks control-flow. Notice from the fault check distributions in Figure 7(b) (for fetch) and Figure 10(b) (for dispatch), no instance of CFD-causing faults are undetected. Moreover, the majority of them are detected through the SPC check, and some, through the ITR check. A small number of CFDs are also caused by faults injected in the decode, rename and backend stages. Referring to the respective fault outcome breakdowns of these pipeline stages (Figure 8(a) for decode, Figure 9(a) for rename, and Figure 11(a) for backend) reveals that these instances of CFD are caused by faults leading to incorrect branch computation (e.g., SRC_LOG_REG, IMM etc. in decode, REN_MAP_TABLE etc. in rename, and READY_BIT, SPEC_LOAD, etc. in the backend). Most of the faults are aptly detected (see ACFD) by ITR in decode, RNA in rename, and TAC in backend, and a very small fraction goes undetected (see UCFD).

5.3.2. SDC outcomes. Next, we explore outcomes that lead to silent data corruption (*SDC). As seen in Figure 6, the majority of USDC instances occur in the rename and backend stages. For insight we refer to their respective breakdowns per fault type in Figure 9(a) (for rename) and Figure 11(a) (for backend). In the rename stage, USDC instances dominantly occur due to the REN_MAP_SRC_INDEX fault type. The anomaly modeled here is a fault while indexing into the rename map table, which is highly likely to cause an instruction to produce a wrong value (due to renaming to an incorrect source register) and cause a SDC. There is no specific fault check in the regimen that is targeted to detect it (the consumer counter check only applies to reading from wrong source registers after correctly renaming). Partial coverage is provided by TAC, which can detect faults if the fault-afflicted instruction issues before its actual producer, indicated by its non-faulty logical source registers. But a large fraction of the faults are undetected and end up causing an SDC. Referring to Figure 11(a), USDC instances in the backend often occur due to faults that directly or indirectly affect an output value (e.g., SRC_VALUE, DST_VALUE, etc.). Such faults are very likely to corrupt an output value that can influence the architectural state, hence, cause SDC. The regimen does not provide comprehensive coverage to values through any of the fault checks. Partial coverage is provided by misprediction detection among confident branches. But the number of confident branches is limited, hence, a large number of such faults go undetected (e.g., all SRC_VALUE faults are undetected) and end up causing SDC.

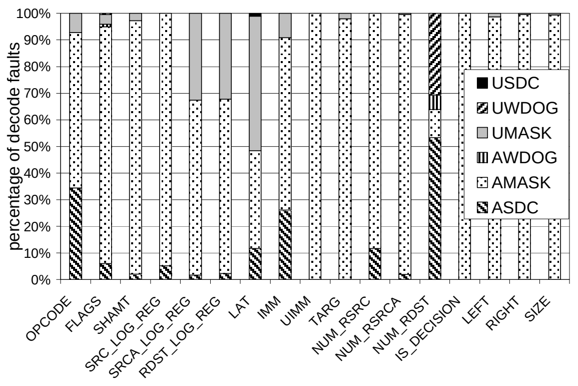
5.3.3. WDOG outcomes. Next, fault outcomes that lead to deadlocks are analyzed (*WDOG). We observe from Figure 6, that almost all deadlocks are detected by the watchdog timer (UWDOG and USDCWDOG), and only a small fraction is detected first by regimen-based checks (AWDOG). This underscores the advantage of including a watchdog timer in a processor. From Figure 6, deadlocks are mostly caused by faults in the dispatch stage. For insight, we look at the per fault type breakdown for the dispatch stage in Figure 10(a). As seen there, deadlocks are caused by all the fault types. This is expected. The ROB_WRITE and RS_WRITE faults result in writing instructions to wrong locations in the ROB and reservation stations, respectively. If instructions wrongly overwrite other instructions, their dependents end up waiting endlessly for results. Some faults in the backend also contribute noticeably to deadlocks, mainly faults impacting the wakeup mechanism (WAKEUP_TAG, LSQ_TAG, etc., as shown in Figure 11(a)).

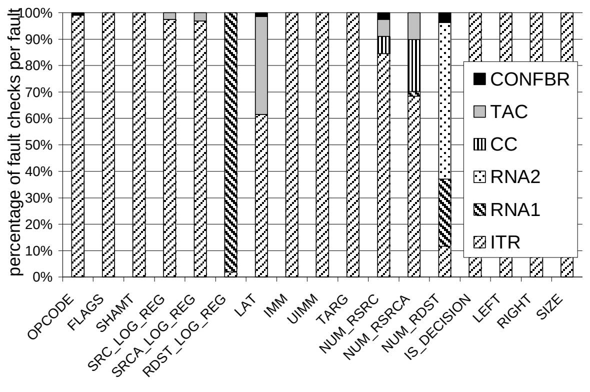




(a) (b)

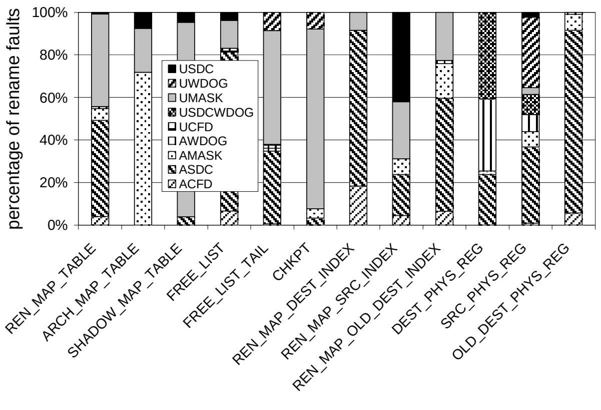
Figure 7. Fetch stage: (a) Distribution of fault outcomes. (b) Distribution of fault checks.

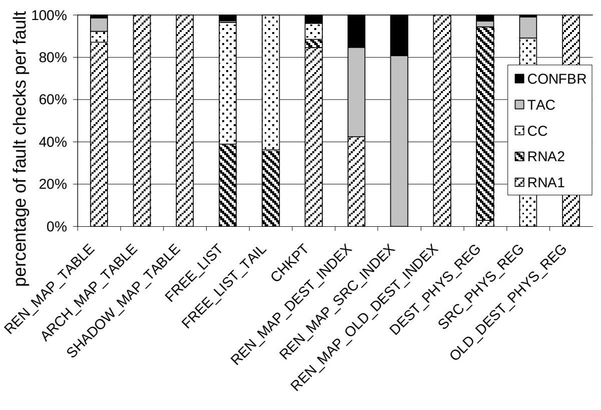




(a) (b)

Figure 8. Decode stage: (a) Distribution of fault outcomes. (b) Distribution of fault checks.





(a) (b)

Figure 9. Rename stage: (a) Distribution of fault outcomes. (b) Distribution of fault checks.

6. Related work

Other low-cost, high-coverage solutions for contemporary superscalar processors include ReStore [9] and Dynamic Dataflow Verification (DDFV) [12]. ReStore exploits atypical behavior (exceptions, confident mispredictions, etc.) as possible symptoms of faults. DDFV uses dataflow signatures to confirm that producers and consumers communicate correctly, exhaustively and implicitly detecting any missteps that can occur in the complex machinery that performs this communication, including rename, issue, register read, and bypass. This rigor comes at the cost of passing and storing signatures everywhere that operands exist in the datapath, and incurs static and dynamic instruction overheads for conveying signatures to the hardware. Moreover, dataflow checking is limited to within blocks for which the dataflow is statically known (not influenced by branches). Argus [11] combines DDFV with control-flow and computation checking in a very simple core, for comprehensive coverage with very low overhead, as such simple cores offer too few resources for time or space redundancy.

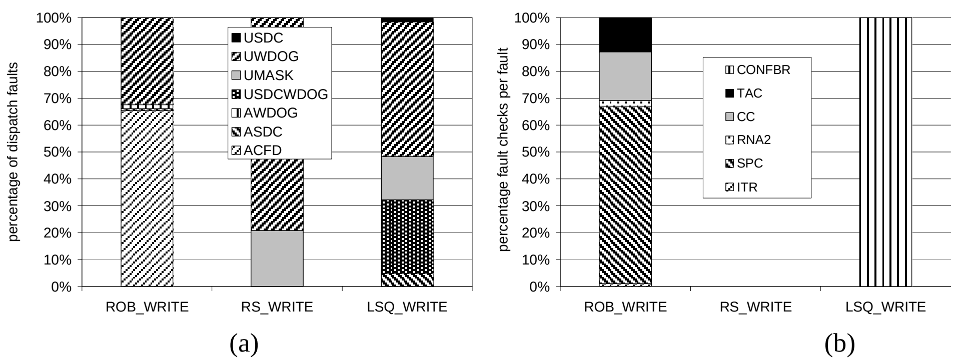


Figure 10. Dispatch stage: (a) Distribution of fault outcomes. (b) Distribution of fault checks.

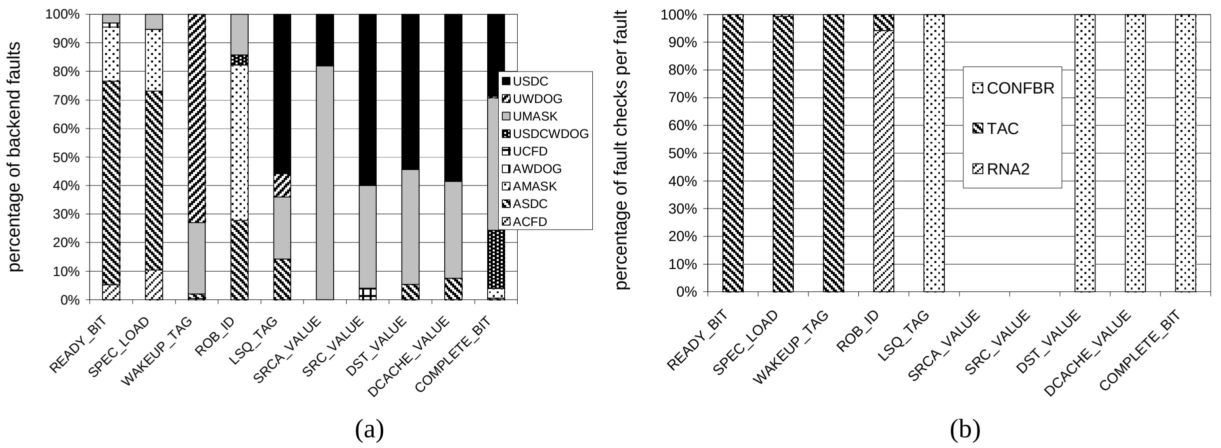


Figure 11. Backend stages: (a) Distribution of fault outcomes. (b) Distribution of fault checks.

7. Summary and future work

Conventional processor fault tolerance based on time/space redundancy is robust but prohibitively expensive for commodity processors. This paper explored an unconventional approach to designing a cost-effective fault-tolerant superscalar processor. The idea is to engage a regimen of microarchitecture-level fault checks. A few simple checks can detect many arbitrary faults in large units, by observing microarchitecture-level behavior and anomalies in this behavior. We showed for the first time that the regimen-based approach provides substantial coverage of an entire superscalar processor. Analysis revealed vulnerable areas which will be the focus for regimen additions. For future work, we are developing a synthesizable verilog superscalar model, which will be leveraged for weighting fault distributions, prototyping the regimen, and evaluating its area/power overheads.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. We also thank Niket Choudhary for early area estimates and Ahmed Al-Zawawi for valuable discussions. This research was supported in part by NSF grants CCF-0429843 and CCF-0702632, and an Intel grant. Any opinions, findings, and conclusions or recommendations herein are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] V. Reddy and E. Rotenberg. Inherent Time Redundancy (ITR): Using program repetition for low-overhead fault tolerance. IEEE/IFIP DSN-37, June 2007.
[2] V. Reddy, A. Al-Zawawi, E. Rotenberg. Assertion-based microarchitecture design for improved fault tolerance. ICCD 2006.
[3] G. Saggese et al. An Experimental Study of Soft Errors in Microprocessors. IEEE Micro., Nov 2005.
[4] S. S. Mukherjee et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. IEEE MICRO-36, Dec 2003.
[5] P. Shivakumar et al. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. DSN-32, 2002.
[6] Z. Kalbarczyk et al. Hierarchical simulation approach to accurate fault modeling for system dependability evaluation. IEEE Transactions on Software Engineering, Sep/Oct 1999.
[7] V. Reddy. Exploiting Microarchitecture Insights for Efficient Fault Tolerance. PhD thesis, ECE Dept., NCSU, July 2007.
[8] K. C. Yeager. The MIPS R10000 superscalar processor. IEEE Micro, April 1996.
[9] N. Wang and S. Patel. ReStore: Symptom based soft error detection in microprocessors. IEEE DSN-35, Jun 2005.
[10] V. Reddy, S. Parthasarathy and E. Rotenberg. Understanding prediction-based partial redundant threading for low-overhead, high-Coverage fault tolerance. ACM ASPLOS-12, Oct 2006.
[11] A. Meixner, M. Bauer, D. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. MICRO-40, 2007.
[12] A. Meixner and D. Sorin. Error Detection Using Dynamic Dataflow Verification. PACT-16, Sep. 2007.