Assertion-Based Microarchitecture Design for Improved Fault Tolerance

Vimal K. Reddy Ahmed S. Al-Zawawi Eric Rotenberg
Department of Electrical and Computer Engineering
North Carolina State University
{vkreddy, aalzawa, ericro}@ece.ncsu.edu

***Abstract*—Protection against transient faults is an important constraint in high-performance processor design. One strategy for achieving efficient reliability is to apply targeted fault checking/masking techniques to different units within an overall reliability regimen. In this spirit, we propose a novel class of targeted fault checks that verify the functioning of the microarchitecture itself, as opposed to the broader challenge of verifying overall architectural correctness of a running program. That is, the checks focus on verifying the mechanics of executing the program. Long term, discriminating between machinery and state may lead to highly efficient reliability solutions with high coverage.**

The key idea is to identify and exploit opportunities to assert microarchitectural "truths". We explore two examples, Register Name Authentication (RNA) for the rename unit and Timestamp-Based Assertion Checking (TAC) for the issue unit of a contemporary out-of-order superscalar processor. Thousands of fault injection experiments show that RNA and TAC microarchitectural assertions detect most unmasked faults for which they are designed.

I. INTRODUCTION

Reliability is an important concern in modern microprocessor design. Efficient solutions to reliability are equally important, so that power and performance overheads do not counteract benefits of high-performance processors.

One strategy to keep reliability overheads contained is to focus on key vulnerable areas of the pipeline and apply a regimen of diverse targeted fault detection techniques. We propose a new methodology that adds to the suite of fault detection techniques available to designers of reliable high-performance processors.

The idea is to insert microarchitectural assertion checks into the processor pipeline. The checks verify microarchitectural “truths” to confirm correct functioning of the machinery. When a fault causes an inconsistency in a microarchitecture mechanism, an assertion check associated with that mechanism fails. Depending on the check and mechanism involved, safe recovery may be possible by rolling back to the architectural state or using more extensive checkpointing schemes. In this paper, we only focus on error detection capabilities of assertion checks.

A program can be viewed as a specification to the microarchitecture. Mechanisms within the microarchitecture work towards the plan laid out in the specification. A transient error in either the specification (i.e., instructions and data) or the microarchitecture machinery can cause unplanned deviations. High-level reliability solutions like redundant multithreading [7, 8, 9, 10, 11, 13, 14] detect deviations from the plan proposed by the specification. In this way, errors in either the specification or the machinery can be detected. The cost is that of reproducing the original plan.

Our class of solutions is geared towards detecting faults in the microarchitecture machinery and not the specification that drives it. Microarchitectural assertion checks can lead to very low overhead fault-tolerant solutions, since a few key assertion checks can provide broad fault coverage of a particular microarchitectural unit and possibly other collateral microarchitectural units.

Since coverage is limited to the microarchitecture machinery, faults that directly instill errors into a program are not detected by microarchitectural assertions. For example, to the microarchitecture, a fault that flips a bit in the source value of an instruction is indistinguishable from a logical bug in the program, i.e., a program value does not present a microarchitectural truth that can be verified. Nonetheless, assertion-checks can be integrated with other solutions aimed at providing coverage over program state. In an overall framework of targeted fault-tolerant solutions, assertion checks can provide coverage over the most diverse part of the architecture, namely the microarchitecture.

The basic approach in assertion-based design is to identify opportunities for making assertions that can detect anomalies in the functioning of a microarchitectural unit. For the issue unit, for example, a microarchitectural assertion can check if dependent instructions issued in sequential order. While there may not be assertion opportunities for all the functionality within a microarchitectural unit, even a few key assertions can provide good coverage of a unit.

As examples of assertion-based design, assertion checks are proposed for two major units of a superscalar processor – Timestamp-based Assertion Checking (TAC) for the issue unit and Register Name Authentication (RNA) for the rename unit. Their fault detection capabilities are evaluated using targeted, random fault injection experiments on a detailed timing simulator.

In particular, the main contributions of this paper are as follows:

- A new fault detection methodology is proposed, based on the idea of microarchitectural assertion checks to detect faults in the microarchitecture machinery. This may lead to very low-overhead, high-coverage fault-tolerant solutions.
- Microarchitectural assertion checks are proposed for the rename unit and issue unit of a contemporary out-of-order superscalar processor. RNA indirectly checks for problems with the destination register mapping of an instruction, based on expected states of physical registers and the correspondence between the rename map table and architectural map table. TAC checks for sequential order among instructions in a data dependence chain, by assigning timestamps to instructions when they issue and comparing consumer timestamps to producer timestamps at retirement.
- Fault detection capabilities of RNA and TAC are evaluated using targeted, random fault injection. Fault injection is performed only on microarchitectural state corresponding to the rename unit and the issue unit.

The rest of the paper is organized as follows. Section II discusses the fault model assumed in this study. Section III discusses fault coverage offered by assertion checks. Section IV.A discusses Timestamp-based Assertion Checking (TAC) and Section IV.B discusses Register Name Authentication (RNA). Section V gives a proof-of-concept evaluation of TAC and RNA using targeted, random fault injection. Section VI discusses related work and Section VII concludes the paper.

II. FAULT MODEL

The fault model assumed in this study is a transient fault that causes a single-bit error in a sequential element or an arbitrary circuit node. The cause of a transient fault may be a particle strike or other noise sources.

We assume there is only one fault in the system that needs to be detected. Previous studies have indicated single-event upsets are a reasonable model for reliability studies [1, 2, 4].

III. FAULT COVERAGE

An assertion check provides fault coverage for combinational and sequential logic which, if afflicted with a transient fault, will cause the assertion check to fire.

Microarchitectural assertions check operational aspects of the microarchitecture. Thus, these assertions cannot check for program correctness. For example, the correctness of a program value cannot be checked. The only natural way to check the value is by redoing the program and confirming the value, as done by techniques like redundant multithreading and N-modular redundancy. Thus, microarchitectural assertion checks do not provide fault coverage for logic that computes and stores program values.

IV. ASSERTION-BASED MICROARCHITECTURE DESIGN

In the following sections, we discuss targeted fault detection solutions within the microarchitecture based on assertion checks. Timestamp-based Assertion Checking (TAC) is discussed in Section IV.A and Register Name Authentication (RNA) is discussed in Section IV.B.

A. Timestamp-Based Assertion Checking

1) TAC Operation

Timestamp-based Assertion Checking (TAC) provides fault coverage for the out-of-order issue unit in a superscalar processor. TAC reasons that an instruction executes only after all its producers have issued and executed. This time-orderliness within a data dependence chain is captured by assigning timestamps to instructions at issue. At retirement, an instruction's timestamp is compared with its producers' timestamps and an inconsistency indicates a violation in issuing order, and hence an error in the issue unit.

To illustrate how TAC works, consider the instruction sequence (and the corresponding data-flow graph) shown in Figure 1. Frames A to F in Figure 2 show how TAC checks the issue logic by assigning timestamps to these instructions.

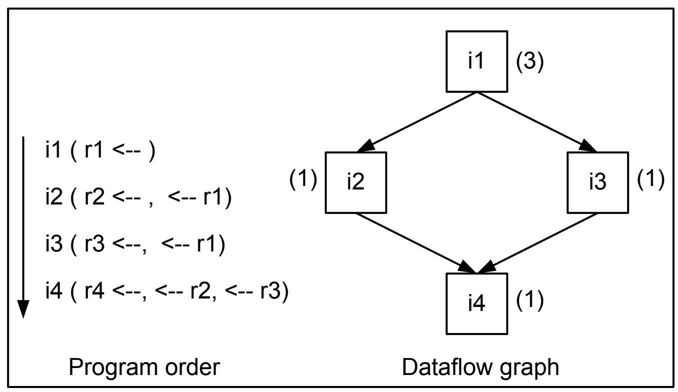

Figure 1. Example data-flow graph with latencies.

Boxes in dark grey indicate new additions to the existing microarchitecture to support TAC. The new additions are: 1) a timestamp counter incremented each cycle and 2) fields in the ROB and architecture map table to store timestamp and execution latency information.

In frame A, all instructions have been fetched and renamed into the ROB. For convenience, we assume the timestamp counter is at 1 at this point. In frame B, we show that all instructions have issued. The timestamps recorded at issue are shown in their ROB entries. In frames C through F, the instructions are committed. When a given instruction I commits, the following main consistency check is performed with its two producers S1 and S2 (the TAC-LAT check):

I_ts >= S1_ts + S1_lat

AND

I_ts >= S2_ts + S2_lat

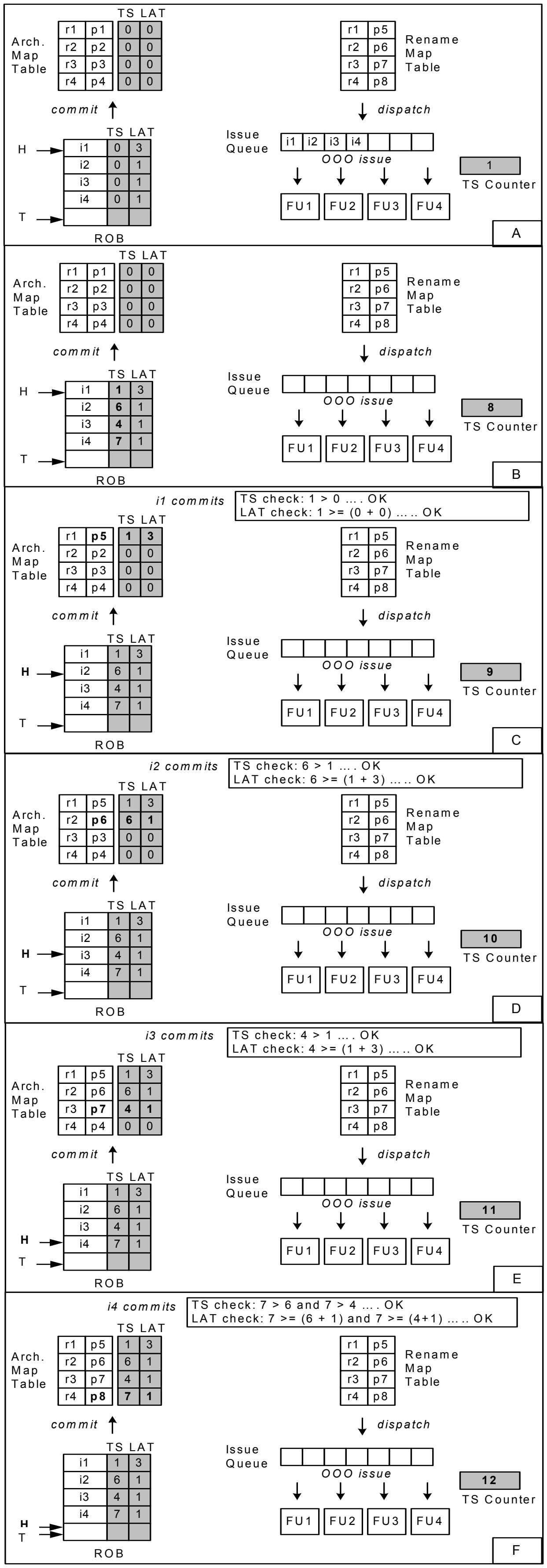

Figure 2. Illustration of TAC checks.

Here, I_ts is the timestamp of instruction I and I_lat is its latency (same for S1 and S2).

A slightly less rigorous check is possible at a lower implementation cost (the TAC-TS check):

I_ts > S1_ts AND I_ts > S2_ts

This consistency check determines whether or not the issue order of instruction I was correct with respect to its producers. However, it is less rigorous, because even if the issue order of instruction I was correct with respect to its producers, it may be that instruction I read its source operands before its producers finished executing. The TAC-LAT check detects even such premature issuing errors, and hence subsumes the TAC-TS check.

In frame C, i1 commits and, as shown, both consistency checks confirm it issued correctly. Frames D and E confirm the same for i2 and i3, respectively. In frame F, since i4 has two sources, consistency checks are performed with respect to both of them, which also succeed.

If i4 had issued with a {timestamp, latency} of {5, 1}, then in frame F, either of the TAC check would compare timestamps and fail (specifically, 5 > 6 and 5 > (6+1)), detecting an issue order violation (i4 issuing before i2). Similarly, if i3 issued with a {timestamp, latency} of {3, 1}, then in frame E, the TAC-TS check would succeed (because i3 issued after i1) but the TAC-LAT check would fail (because 3 < (3 + 1)) and detect a premature issuing violation.

2) Handling Wraparound of Timestamp Counter

TAC uses a timestamp counter incremented each cycle. When the counter wraps around, it may cause false error alarms. Hence, the wraparound case needs special handling.

To deal with wraparounds, each new counter wraparound is viewed as entering a new phase. If we assume the timestamp counter is big enough that only one wraparound can occur within the scope of the ROB, we can be assured each committing instruction has a timestamp from either the current phase or the previous phase (i.e., immediately before the current phase). To identify the current phase, a toggling current phase bit is maintained along with the timestamp counter. A wraparound toggles the current phase bit. To remember the phase that an instruction's timestamp belongs to, a phase bit is also appended to each ROB entry.

At commit, an instruction's phase can be identified by comparing its phase bit with the current phase bit: if equal, it issued in the current phase, else it issued in the previous phase.

To identify the phase of an instruction's producer, two bits are added to the architecture map table – a current-phase production bit and a previous-phase production bit. When a producer instruction commits, it sets either the current-phase production bit or previous-phase production bit corresponding to its logical destination register, depending on the phase of its timestamp. At counter wraparound, current-phase production bits are flash copied into the previous-phase production bits (and then all current-phase production bits are reset). If both bits are unset, it indicates a past-phase (i.e., neither current phase nor previous phase) production.

At commit, a current-phase instruction is compared only with its current-phase producers, i.e., comparisons with previous-phase or past-phase producers are skipped to avoid false alarms. Moreover, skipping previous-phase and past-phase timestamp comparisons is safe, because the fact that previous-phase and past-phase producers are identified confirms that the current-phase consumer instruction issued after these producers (implicitly implements the TAC-TS check without timestamp comparisons). There is a slight residual vulnerability since the latencies of previous-phase and past-phase producers are not accounted for (no implicit TAC-LAT check).

Similarly, at commit, a previous-phase instruction is compared only with previous-phase producers, i.e., comparisons with past-phase producers are skipped using the same reasoning as before. Moreover, the previous-phase instruction is in issue order violation if it has any current-phase producers.

A sufficiently large timestamp counter enables handling counter wraparounds elegantly and prevents false alarms, as discussed above. For a 64-entry ROB and a L2-miss latency of 100 cycles, a 13-bit counter is sufficient to ensure at most one wraparound within the scope of the ROB, in the worst case of chained L2 misses.

B. *Register Name Authentication (RNA)*

Register Name Authentication aims to detect faults in the destination register renaming logic and various renaming structures used in a contemporary superscalar processor [21], i.e., rename map table, architecture map table, shadow map tables (branch checkpoints), active list, and free list. We present some low-overhead assertion checks that can be introduced into the pipeline to detect errors in destination register mappings, caused by faults in the rename logic and structures. The RNA checks presented in this paper are not intended to detect all possible faults. Rather, the intention is to highlight that low-overhead microarchitectural assertion checks have significant potential for fault detection.

1) *Previous mapping check*

When an instruction's logical destination register is renamed, the rename map table is updated with the new mapping. However, before updating the mapping, the previous mapping can be recorded in the instruction's entry in the active list. Thus, the instruction's entry in the active list has both the current and previous mappings of its logical destination register. Some superscalar processors already record the previous mapping in the active list, to facilitate freeing the previous mapping at retirement [21]. At retirement, before committing the instruction's current mapping to the architecture map table, the instruction's previous mapping can be confirmed to be the same as the corresponding mapping in the architecture map table.

This first RNA check can detect many faults that affect mappings in the rename map table, architecture map table, shadow map tables, and active list.

1. A fault that changes a mapping in the rename map table will be detected by the next instruction to update the mapping. The previous mapping recorded with this instruction (incorrect) will differ from the corresponding mapping in the architecture map table (correct). Detecting this fault is valuable for a couple of reasons, specifically, the fault may cause the wrong mapping to be freed and may cause consumers to receive a wrong source mapping.
2. A fault that changes a mapping in the architecture map table will be detected by the next instruction to update the mapping at retirement. This instruction's previous mapping (correct) will not match the corresponding mapping in the architecture map table (incorrect). If there is an exception before the instruction commits, the fault will not be detected and may have consequences for precise state. Otherwise the fault is detected and is not distinguishable from a fault in the rename map table.
3. A fault that changes a mapping in a shadow map survives if the shadow map is copied to the rename map table during a branch misprediction recovery. This scenario is then similar to a fault in the rename map table.
4. A fault that changes the previous mapping of an instruction in the active list will be detected when the instruction reaches retirement. Its previous mapping (incorrect) will not match the corresponding mapping in the architecture map table (correct). Detecting this fault is valuable because it causes the wrong mapping to be freed.
5. A fault that changes the current mapping of an instruction X in the active list will be detected by the next instruction Y with the same logical destination register. X will commit the wrong mapping to the architecture map table. When Y commits, its previous mapping (correct) will not match the corresponding mapping in the architecture map table (incorrect). This scenario is basically the same as a fault in the architecture map table and is also vulnerable to the intervening exception case.

Summing up, there is inherent redundancy among the rename structures, and the first RNA check exploits this fact to detect inconsistencies in redundant destination register mappings that point to underlying faults.

The first fault scenario (a fault changes a mapping in the rename map table) is illustrated in Figure 3. Various events are indicated by numbers, indicating the order of events in time. Events 1 through 7 are for instruction I1 and events 8 through 12 are for instruction I2. At time 4, a fault causes the mapping r4 – p4 to change to r4 – p5, in the rename map table. When a later instruction I2 with logical destination register r4 is renamed, it records p5 as the old mapping of r4, instead of p4. At retirement of I2, the RNA check fails to match I2's previous mapping of r4 (p5) to the mapping of r4 in the architecture map table (p4), hence detecting the fault. Again, catching this fault is valuable because the fault in the rename map table may have caused consumers of I1 to obtain a wrong source mapping (p5 instead of p4) and also causes I2 to free p5 instead of p4.

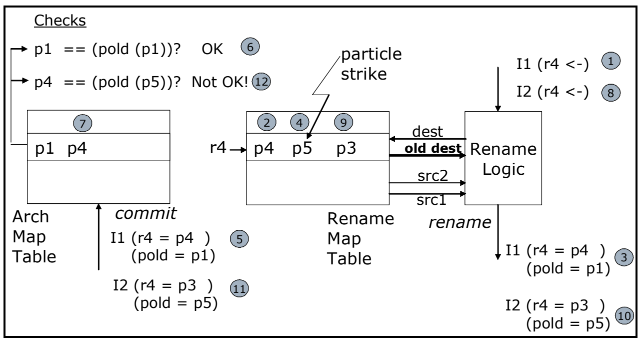

Figure 3. Fault detection using RNA's "previous mapping check".

2) *Writeback state check*

RNA's previous mapping check can detect faults that cause inconsistencies between the architecture map table and other structures. However, it cannot detect faults in the destination renaming logic itself, i.e., the logic that presents a new mapping to the rename map table. An erroneous mapping this early is not distinguishable from a correct mapping. The erroneous mapping will be consistent among all the structures (rename map, architecture map, shadow maps, and active list).

To detect faults in the destination rename logic (including the rename logic and free list), we exploit the insight that such faults cause register conflicts. That is, an instruction may be assigned a physical register that is in use by another active instruction, or committed, or still free (in the middle of the free list).

Fortunately, some superscalar designs already associate ready and free bits with each physical register [21], which can be leveraged to detect conflicts, hence, faults of the nature described above. The ready bit of a physical register is set when an instruction writes (or is about to write) to the physical register and is cleared when the physical register is added back to the free list. The free bit of a physical register is set when it is added back to the free list and is cleared when it is popped from the free list.

The RNA writeback state check confirms neither of the bits is set before writing to a physical register. The intuition behind the RNA writeback state check is as follows. Suppose that the destination renaming logic produces a faulty physical register tag. There are two possibilities regarding the faulty tag[1], (1) it is in the free list, or (2) it is already being used by another instruction or is part of the architectural state. A set free bit at writeback detects the case of writing to a free register and a set ready bit detects the case of writing to a physical register that is already being used or part of the architectural state. For the case where a register is assigned to two instructions, the assertion check will fire when either the correct instruction or the faulty instruction writes back, depending on who writes last (the last writer observes a set ready bit).

The RNA writeback state check is illustrated in Figure 4. Again, various events are indicated by numbers in order of occurrence. At time 3, a fault causes the destination renaming logic to produce a wrong physical register tag p6 instead of p5, for instruction I1. P6 was last used by an instruction that finished and committed to the architectural state, and p6 is still part of the architectural state (shown by r10 – p6 in the architecture map table). Hence, p6 has {ready, free} of {1, 0}. After I1 executes and before it writes to p6, the ready and free bits of p6 are checked for consistency. At time 6, the check fails because the ready bit of p6 appears to have been set already. Hence, the fault is detected.

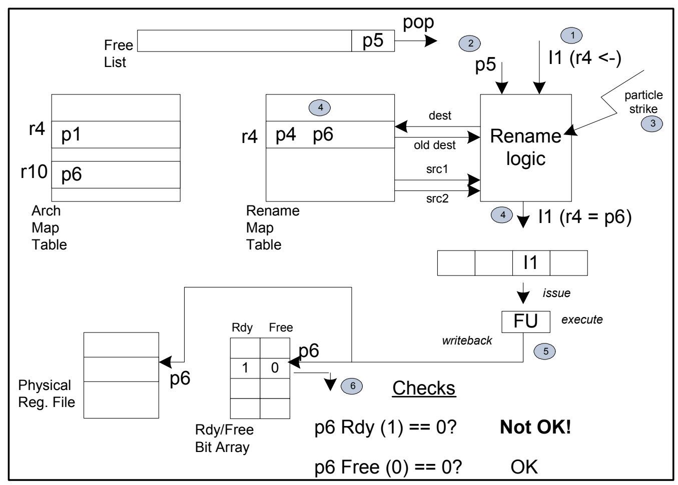

Figure 4. Fault detection using RNA-writeback.

3) *Source Register Renaming Faults*

We have discussed faults in the rename unit that affect destination registers. For faults that cause only source register renaming errors, we are investigating low-overhead assertion checks along the lines of current RNA checks. However, for some faults a timeout mechanism, like a watchdog timer [6], could be an effective assertion check. A watchdog timer can detect some pure source renaming errors, if they cause a timeout by blocking retirement, waiting for phantom producers to issue. Examples are faulty source registers in the free list that never get popped, faulty source registers in the forward slice of the faulty instruction that cause a cyclic dependency, etc.

[1] Faults can also cause invalid physical register tags, if the number of physical registers is not a power of two. For example, for a 126-entry physical register file, the tags 127 and 128 are invalid. Detecting faulty, invalid tags is straightforward.

V. Results and Evaluation

A. Methodology

We implemented the TAC and RNA checks in a cycle-level simulator to evaluate their fault detection capability. Faults are randomly injected into the microarchitectural state of the simulator pertaining to the rename and issue units.

The modeled microarchitecture is similar to the MIPS R10000 [21]. The microarchitecture configuration is shown in Table 1.

TABLE 1. MICROARCHITECTURE CONFIGURATION.

L1 I & D Caches	64KB, 4-way, 64B line, LRU, L1hit = 1 cycle, L1miss/L2hit = 10 cycles
L2 Unified Cache	1MB, 8-way, 64B line, LRU, L1miss/L2miss = 100 cycles
Branch Predictor	gshare, 16-bit history, 2^{20} entries
Superscalar Core	reorder buffer (ROB): 64 dispatch/issue/retire bandwidth: 4 per cycle

A separate, "golden" (fault-free) simulator is run in-sync with the faulty simulator. When an instruction is committed to the architectural state in the faulty 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, we use an observation window of one million cycles. Results are similar for a five million cycle observation window.

As a preliminary test of TAC and RNA, the simulator is first run without fault injection. None of the assertion checks fire, indicating that there are no false alarms during correct operation.

We conduct two fault injection campaigns, targeting TAC and RNA separately. Each campaign consists of 1,000 faults. Moreover, the two campaigns are repeated for nine SPEC2K benchmarks. The benchmarks are compiled with the Simplescalar gcc compiler [19] for the PISA ISA.

A fault may lead to one of four possible outcomes, depending on (1) whether the fault is detected by an assert ("Assert") or not ("Undet") and (2) whether the fault corrupts architectural state ("SDC") or not ("Masked"). Thus, the four possible outcomes of a fault are Assert+SDC, Assert+Masked, Undet+SDC, and Undet+Masked.

The combination of an assertion check and a SDC (Assert+SDC) occurring in the same observation window is interesting, because it indicates that the assertion check was able to detect a potential silent data corruption. We are not concerned with the order of occurrence of the two events (Assert before SDC vs. SDC before Assert), as our focus is on fault detection. For RNA, either order may occur. In contrast, TAC always fires an assertion check before the detected fault can cause a SDC, permitting safe recovery from the architectural state.

An assertion check may also detect a fault that is ultimately masked (Assert+Masked). For example, if a faulty register mapping in the rename map table is overwritten before being consumed by any instruction, the fault is detected by RNA but is masked. Similarly, TAC will detect a prematurely issued instruction, even if the instruction still produces the correct outcome (for example, a branch may produce the correct taken/not-taken direction despite consuming wrong values).

As a final note about methodology, we run perfect branch prediction in order to minimize masking due to speculative state. This way, the chance that a fault surfaces as an error is increased, thus testing the fault detection capabilities of TAC and RNA better.

B. TAC Evaluation

To test TAC, we randomly inject faults that cause instructions to issue prematurely. In particular, (1) ready bits of physical registers are prematurely set and (2) speculatively issued dependents of cache-missing loads are not reissued. The TAC-LAT check discussed in Section IV.A.1 is used to detect faults. Results are shown in Figure 5.

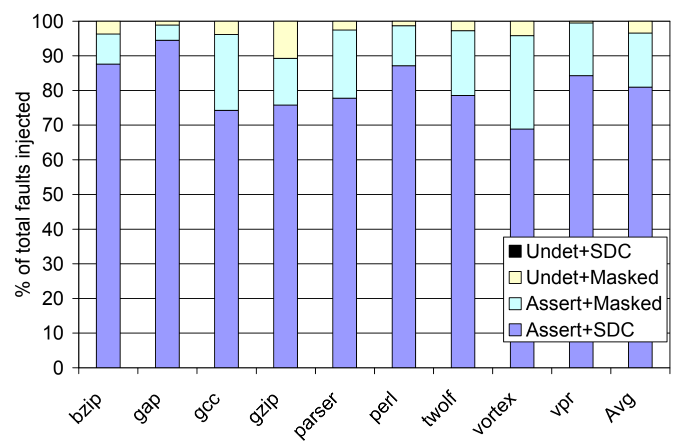

Figure 5. Breakdown of outcomes of TAC fault injection campaign.

As seen in Figure 5, on average, 80% of the faults are detected by TAC, that would otherwise cause a SDC (Assert+SDC). Another 17% of the faults are detected by TAC, that are masked (a prematurely issued instruction produces the correct outcome even with wrong operands). The remaining 3% of the faults are undetected and masked, because the faults did not cause instructions to issue prematurely (e.g., select logic may stall issuing the instructions long enough). Notice that none of the faults were allowed to cause a SDC (no Undet+SDC). This is expected because TAC detects violations before wrongly issued instructions commit to the architectural state.

C. RNA Evaluation

To test RNA, we inject faults that cause renaming anomalies similar to those discussed in Section IV.B. In particular, four types of faults are injected, 1) bits of a random entry in the architectural map table are flipped (arch_map), 2) bits of a random entry in the rename map table are flipped (rename_map), 3) bits of a random entry in the physical register freelist are flipped (freelist) and 4) bits of the destination physical register tag of an instruction are flipped at dispatch, to emulate a destination renaming logic error (dest).

For the dest fault, only the instruction's tag is affected by the fault, i.e., the rename map table and future consumers get the correct tag. This creates a disconnect between the faulty producer and its consumers, potentially causing deadlock. To detect deadlocks that phantom producers can cause, we include a watchdog timer check (wdog) aside from the two primary RNA checks discussed in Section IV.B, i.e., the previous mapping check (prevmapping) and the writeback state check (writeback). We discuss faults in relation to the checks that detect them (prevmapping, writeback, wdog), later in this section.

A breakdown of fault outcomes is shown in Figure 6. Corresponding to the watchdog timer, two new outcomes appear in Figure 6. Assert+Wdog represents faults that are detected by RNA assertion checks, which would have later resulted in a deadlock had they not been detected. Undet+Wdog represent faults that escape RNA assertion checks and cause a deadlock, and hence will only be detected by a watchdog timer.

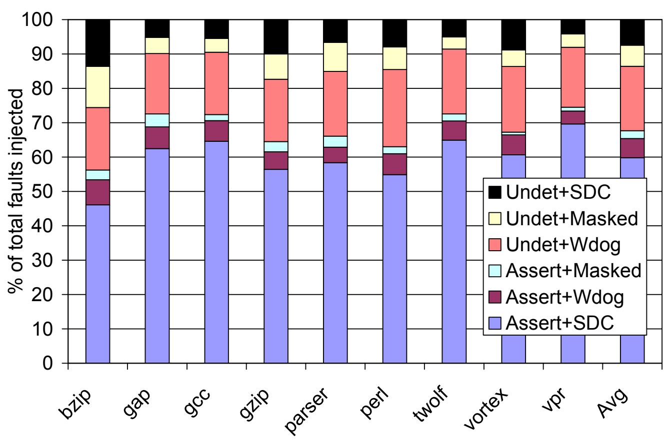

Figure 6. Breakdown of outcomes of RNA fault injection campaign.

As shown in Figure 6, on average, 60% of the faults are detected by RNA assertion checks alone, that would otherwise cause a SDC (Assert+SDC). Another 6% of the faults are detected by RNA assertion checks, that would have caused deadlocks had there not been the earlier RNA checks (Assert+Wdog). Finally, another 2% of the faults are detected by RNA assertion checks, that are also masked (Assert+Masked).

About 18% of the faults escape RNA checks and cause a deadlock, requiring the watchdog timer to detect the faults (Undet+Wdog). About 8% of the faults lead to SDC and are not detected by the two RNA checks (Undet+SDC).

Hence, we find that the two RNA checks proposed in this paper provide reasonable fault detection – 67% of the injected faults are detected on average. Considering the low overheads of RNA, this is a very promising result.

To further understand the fault detection capability of RNA, and possibly enhance it, we investigate the relationship between different fault types and the corresponding outcomes they produce. Figure 7 shows the four fault types used in the RNA fault injection campaign on the x-axis (arch_map, rename_map, freelist, and dest) and the distribution of all possible outcomes due to a fault on the y-axis.

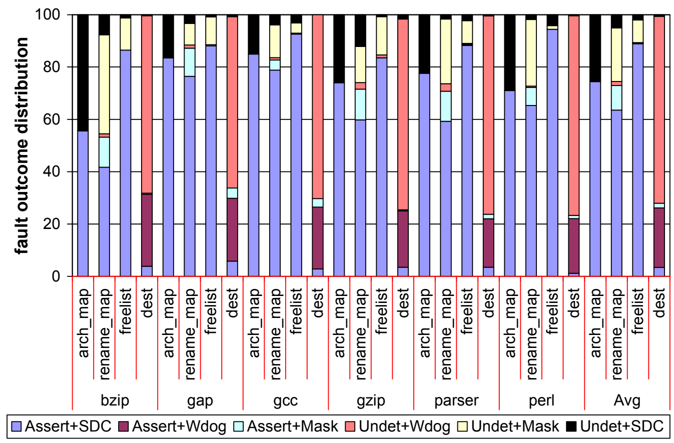

Figure 7. Relation of faults to outcomes.

We notice that some faults get very good detection coverage. For instance, faults on freelist entries are detectable 90% of the time, on average. On the other hand, faults on destination renaming logic get poor coverage from RNA checks, but good coverage from the watchdog timer. A good result is that faults on the architectural map get detected more than 70% of the time, on average. At the same time, a majority of SDC also occurs due to faults on the architectural map, motivating further schemes to protect the committed state.

We next investigate the relation between fault types and different RNA assertion checks that detect them. This may enable understanding which schemes are more suitable for particular types of faults. Figure 8 shows the four fault types on the x-axis and their coverage by various assertion checks on the y-axis.

As expected, the prevmapping check is most suitable for faults on map table entries. For faults on the freelist, the writeback check provides most of the fault coverage. The prevmapping check fails since faults in the freelist are consistent in all tables. Writeback checks provide limited coverage of destination renaming logic faults, which are mainly covered by watchdog timer checks.

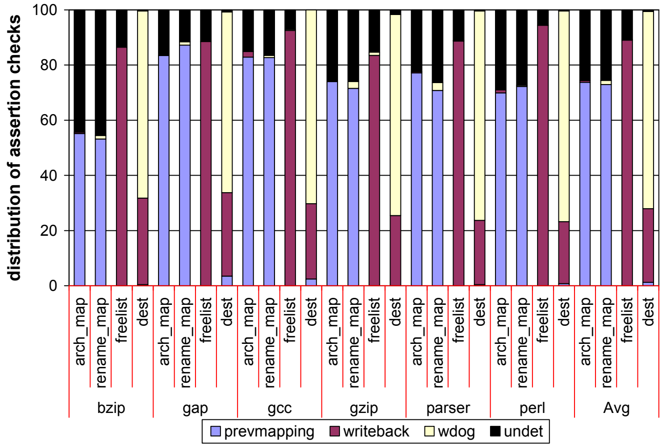

Figure 8. Relation of faults to assertion checks.

VI. Related Work

Recent studies [1, 2, 3, 4] have shown hardware reliability will be a serious concern in the coming generation of high-performance microprocessors. Our work targets transient errors in a very low cost manner and is very relevant in this context.

There have been many proposals for hardware reliability solutions at various levels. In N-Modular Redundancy (e.g., [20]), N processors run in lock-step and compare results at an external chip interface. It has very high cost due to processor redundancy. Redundant Multithreading solutions (e.g., [7, 8, 9, 10, 11, 13, 14]) provide a cheaper alternative where SMT threads are used for redundancy. But RMT has considerable slowdown and high power consumption. Compared to redundant execution, our targeted solution has relatively low performance and power overheads. All redundant execution based solutions transcend checking to an architecture level by comparing values and hence, perform architectural correctness checks. This allows detection of any transient error in the pipeline. Our solution only does microarchitecture correctness checks.

DIVA [6] uses a simple core at retirement to check results of an out-of-order core. In contrast, our solution applies low cost assertion checks inside the microarchitecture. Several other proposals modify an existing superscalar core by adding redundant pipelines or instruction replication to provide coverage of transient faults [5, 12, 15]. Assertion checks do not require such radical modifications to an existing core. More localized checking solutions have also been proposed (e.g., [17, 18]). Our solution provides much broader coverage of the pipeline.

VII. Conclusion

This paper introduces the notion of microarchitecture assertion checks for detecting errors in a high-performance processor pipeline. Assertion checking is a compressed means of detecting several microarchitectural anomalies collectively with a few key “truth” checks, and hence, it can provide broad fault coverage of the microarchitecture at a very low cost. We presented two examples of low-overhead assertion-based checks, TAC and RNA, and showed that they provide good fault detection coverage for the issue unit and the rename unit, respectively.

We believe assertion-based microarchitecture design is an attractive solution for increasing reliability of a processor at a very low cost. In the grand scheme of targeted fault tolerance solutions, checking microarchitecture machinery separately from actual program state might open up new directions for low cost, efficient reliability solutions.

Acknowledgments

This research was supported by NSF CAREER grant No. CCR-0092832, and generous funding and equipment donations from Intel. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] G. Saggese et al. An Experimental Study of Soft Errors in Microprocessors. IEEE Micro, Nov 2005.
[2] P. Shivakumar et al. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. ICCD 2002.
[3] S. Borkar et al. Parameter variations and impact on circuits and microarchitecture. DAC 2003.
[4] N. Wang et al. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. DSN 2004.
[5] S. Kim et al. SSD: An Affordable Fault Tolerant Architecture for Superscalar Processors. PRDC 2001.
[6] T. M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. MICRO 1999.
[7] M. Gomaa et al. Transient-fault recovery for chip multiprocessors. ISCA 2003.
[8] M. Gomaa et al. Opportunistic transient-fault detection. ISCA 2005.
[9] S. S. Mukherjee et al. Detailed design and evaluation of redundant multithreading alternatives. ISCA 2002.
[10] S. S. Mukherjee et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. MICRO 2003.
[11] Z. Purser et al. A study of slipstream processors. MICRO 2000.
[12] J. Ray et al. Dual use of superscalar datapath for transient-fault detection and recovery. MICRO 2001.
[13] S. K. Reinhardt et al. Transient fault detection via simultaneous multithreading. ISCA 2000.
[14] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. FTCS 1999.
[15] J. C. Smolens et al. Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures. MICRO 2004.
[16] N. J. Wang et al. ReStore: Symptom based soft error detection in microprocessors. DSN 2005.
[17] M. Nicolaidis. Efficient Implementations of Self-Checking Adders and ALUs. FTCS 1993.
[18] J.H. Patel et al. Concurrent Error Detection in ALUs by Recomputing with Shifted Operands. IEEE Transactions on Computers. July 1982.
[19] D. Burger et al. The Simplescalar Toolset, Version 2. Tech. Report CS-TR-1997-1342, CS Department, University of Wisconsin-Madison, July 1997.
[20] D. McEvoy. The Architecture of Tandem’s nonstop system. ACM/CSC-ER 1981.
[21] K. C. Yeager. The MIPS R10000 Superscalar Processor. IEEE Micro, April 1996.