Inherent Time Redundancy (ITR): Using Program Repetition for Low-Overhead Fault Tolerance

Vimal Reddy, Eric Rotenberg
Center for Efficient, Secure and Reliable Computing, ECE, North Carolina State University
{vkreddy, ericro}@ece.ncsu.edu

Abstract

A new approach is proposed that exploits repetition inherent in programs to provide low-overhead transient fault protection in a processor. Programs repeatedly execute the same instructions within close time periods. This can be viewed as a time redundant re-execution of a program, except that inputs to these inherent time redundant (ITR) instructions vary. Nevertheless, certain microarchitectural events in the processor are independent of the input and only depend on the program instructions. Such events can be recorded and confirmed when ITR instructions repeat.

In this paper, we use ITR to detect transient faults in the fetch and decode units of a processor pipeline, avoiding costly approaches like structural duplication or explicit time redundant execution.

1. Introduction

Technology scaling makes transistors more susceptible to transient faults. As a result, it is becoming increasingly important to incorporate transient fault tolerance in future processors.

Traditional transient fault tolerance approaches duplicate in time or space for robust fault tolerance, but are expensive in terms of performance, area, and power, counteracting the very benefits of technology scaling. To make fault tolerance viable for commodity processors, unconventional techniques are needed that provide significant fault protection in an efficient manner. In this spirit, we are pursuing a new approach to fault tolerance based on microarchitecture insights. The idea is to engage a regimen of low-overhead microarchitecture-level fault checks. Each check protects a distinct part of the pipeline, thus, the regimen as a whole provides comprehensive protection of the processor. This paper adds to the suite of microarchitecture checks that we have begun developing. Recently, we proposed microarchitecture assertions to protect the register rename unit and the out-of-order scheduler of a superscalar processor [3]. In this paper, we introduce a new concept called inherent time redundancy (ITR), which provides the basis for developing low-overhead fault checks to protect the fetch and decode units of a superscalar processor. Although ITR only protects the fetch and decode units, it is an essential piece of an overall regimen for achieving comprehensive pipeline coverage.

Programs possess inherent time redundancy (ITR): the same instructions are executed repeatedly at short intervals. This program repetition presents an opportunity to discover low-overhead fault checks in a processor. The key idea is to observe microarchitectural events which depend purely on program instructions, and confirm the occurrence of those events when instructions repeat.

There have been previous studies on instruction repetition in programs [1][2]. The focus has been on reusability of dynamic instruction results to reduce the number of instructions executed for high performance. Our proposal is to exploit repetition of static instructions for low-overhead fault tolerance.

We characterize repetition in SPEC2K programs in Figure 1 (integer benchmarks) and Figure 2 (floating point benchmarks). Instructions are grouped into traces that terminate either on a branching instruction or on reaching a limit of 16 instructions. The graphs plot the number of dynamic instructions contributed by static traces. Static instructions are unique instructions in the program binary, whereas dynamic instructions correspond to the instruction stream that unfolds during execution of the program binary.

A relatively small number of static instructions contribute a large number of dynamic instructions. For instance, in most integer benchmarks, less than five hundred static traces contribute nearly all dynamic instructions (e.g., in bzip, 100 static traces contribute 99% of all dynamic instructions). Gcc and vortex are the only exceptions due to the large number of static traces. Floating point benchmarks are even more repetitive, as seen in Figure 2 (e.g., in wupwise, 50 static traces contribute 99% of all dynamic instructions).

An important aspect of repetition is the distance at which traces repeat. This is characterized in Figure 3

37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)
0-7695-2855-4/07 $20.00 © 2007 IEEE

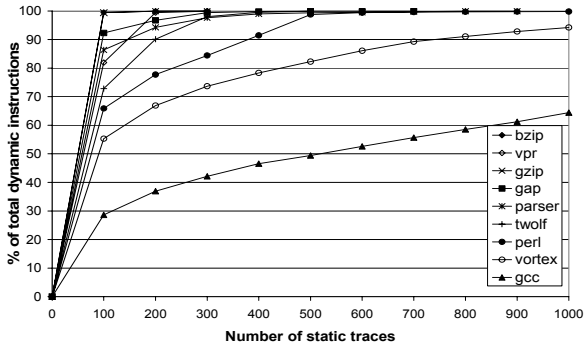

Figure 1. Dynamic instructions per 100 static traces (integer benchmarks).

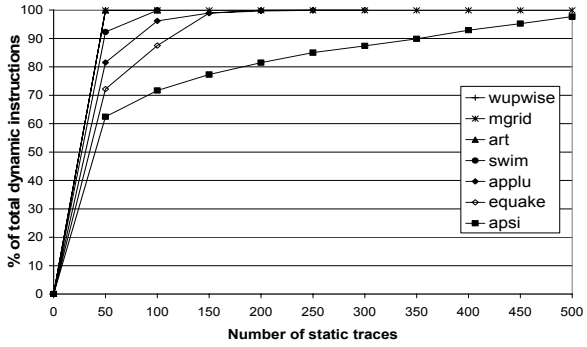

Figure 2. Dynamic instructions per 50 static traces (floating point benchmarks).

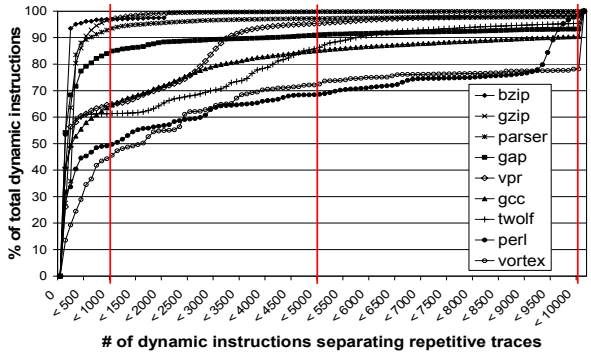

Figure 3. Distance between trace repetitions (integer benchmarks).

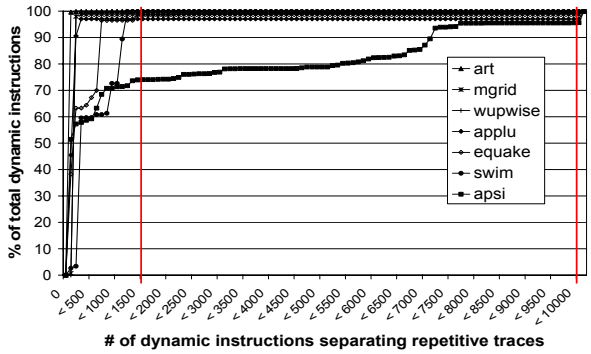

Figure 4. Distance between trace repetitions (floating point benchmarks).

(integer benchmarks) and Figure 4 (floating point benchmarks). Here, instructions are grouped into traces like before, and the number of dynamic instructions between repeating traces is measured. The graphs show the number of dynamic instructions contributed by all static traces that repeat within a particular distance. Distances are shown at increasing intervals of five hundred dynamic instructions.

As seen, there is a high degree of ITR in programs. In all integer benchmarks, except perl and vortex, 85% of all dynamic instructions are contributed by traces repeating within five thousand instructions, four of them reaching that target within one thousand instructions. In all floating point benchmarks, except apsi, nearly all dynamic instructions are contributed by repetitive traces with high proximity (within 1500 instructions).

The main idea of the paper is to record and confirm microarchitecture events that occur while executing highly repetitive instruction traces. The fact that relatively few static traces contribute heavily to the total instruction count, suggests that a small structure is sufficient to record events for most benchmarks. We propose to use a small cache to record microarchitecture events during repetitive traces. The cache is indexed with the program counter (PC) that starts a trace. A miss in the cache indicates the unavailability of a counterpart to check the correctness of the microarchitectural events. However, misses do not always lead to loss of fault detection. A future hit to a trace that previously missed in the cache can detect anomalies during execution of both the missed instance and the newly executed instance of the trace. In a single-event upset model, a reasonable assumption for fault studies, the two instances will differ if there is a fault. However, if a missed instance is evicted from the cache before it is accessed, it constitutes a loss in fault detection, since a fault during the missed instance goes undetected. Based on this, even benchmarks with a large number of static traces and mild proximity (e.g., gcc) can get reasonable fault detection coverage with small event caches.

The recorded microarchitectural events depend purely on instructions being executed. For example, the decode signals generated upon fetching and decoding an instruction are the same across all instances. Recording and confirming them to be the same can detect faults in the fetch and decode units of a processor. Indexes into the rename map table and architectural map table generated for a trace are constant across all its instances. Recording and confirming their correctness will boost the fault

coverage of the rename unit of a processor, especially when used with schemes like Register Name Authentication (RNA) [3]. For instance, RNA cannot detect pure source renaming errors like reading from a wrong index in the rename map table. Further, recording and confirming correct issue ordering among instructions in a trace can detect faults in the out-of-order scheduler of a processor, similar to Timestamp-based Assertion Checking (TAC) [3].

In this paper, we add microarchitecture support to use ITR to extend transient fault protection to the fetch and decode units of a processor. Signals generated by the decode unit for instructions in a trace are combined to generate a signature. The signature is stored in a small cache, called the ITR cache. On the next occurrence of the trace, the signature is re-generated and compared to the signature stored in the ITR cache. A mismatch indicates a transient fault either in the fetch or the decode unit of the processor. On fault detection, safe recovery may be possible by flushing and restarting the processor from the faulting trace, or the program must be aborted through a machine check exception. We provide insight into diagnosing a fault and define criteria to accurately identify fault scenarios where safe recovery is possible, and where aborting the program is the only option.

The main contributions of this paper are as follows:

- A new fault tolerance approach is proposed based on inherent time redundancy (ITR) in programs. The key idea is to record and confirm microarchitectural events that depend purely on program instructions.
- We propose an ITR cache to record microarchitectural events pertaining to a trace of instructions. The key novelty is that misses in the ITR cache do not directly lead to a loss in fault detection. Only evictions of unreferenced, missed instances lead to a loss in fault detection coverage. We develop microarchitectural support to use the ITR cache for protecting the fetch and decode units of a high-performance processor.
- On fault detection, we show it is possible to accurately identify the correct recovery strategy: either a lightweight flush and restart of the processor, or a more expensive program restart.
- We show that the ITR-based approach compares favorably to conventional approaches like structural duplication and time redundant execution, in terms of area and power.

The rest of the paper is organized as follows. Section 2 discusses detailed microarchitectural support to exploit ITR for protecting the fetch and decode units of a superscalar processor. In Section 3, the ITR cache design space is explored to achieve high fault coverage. In section 4, we perform fault injection experiments to further evaluate fault coverage. In Section 5, we compare area and power overheads of the ITR approach to other fault tolerance approaches. Section 6 discusses related work and Section 7 summarizes the paper.

2. ITR components

The architecture of a superscalar processor, augmented with support for exploiting ITR, is shown in Figure 5. The shaded components are newly added to protect the fetch and decode units of the processor using ITR. The new components are described in subsections 2.1 through 2.5.

2.1. ITR signature generation

As seen in Figure 5, signals from the decode unit are redirected for signature generation. The signals are continuously combined until the end of each trace. The end of a trace is signaled upon encountering a branching instruction or the last of 16 instructions. On a trace ending instruction, the current signature is dispatched into the ITR ROB. The signature is then reset and a new start PC is latched in preparation for the next trace.

Signature generation could be done in many ways. We chose to simply bitwise XOR the signals of a new instruction with corresponding signals of previous instructions in the trace. For a given trace, if a fault on an instruction in the fetch unit or the decode unit causes a wrong signal to be produced by the decode unit, then the signature of the trace would differ from that of a fault-free signature. Even multiple faulty signals in a trace would lead to a difference in signature, unless an even number of instructions in the trace produce a fault in the same signal. Using XOR to produce the signature loses information about the exact instruction that caused a fault. But this precision is not required as long as recovery is cognizant that a fault could be anywhere in the trace and rollback is prior to the trace. For a single-event upset model, we believe this overall approach is sufficient for detecting faults on an instruction of a trace in the fetch and decode units.

2.2. ITR ROB and ITR cache

Trace signatures are dispatched into the ITR ROB, when trace termination is signaled. The ITR ROB is sized to match the number of branches that could exist in the processor, since every branch causes a new trace. Since a trace is terminated on a branch, its ITR

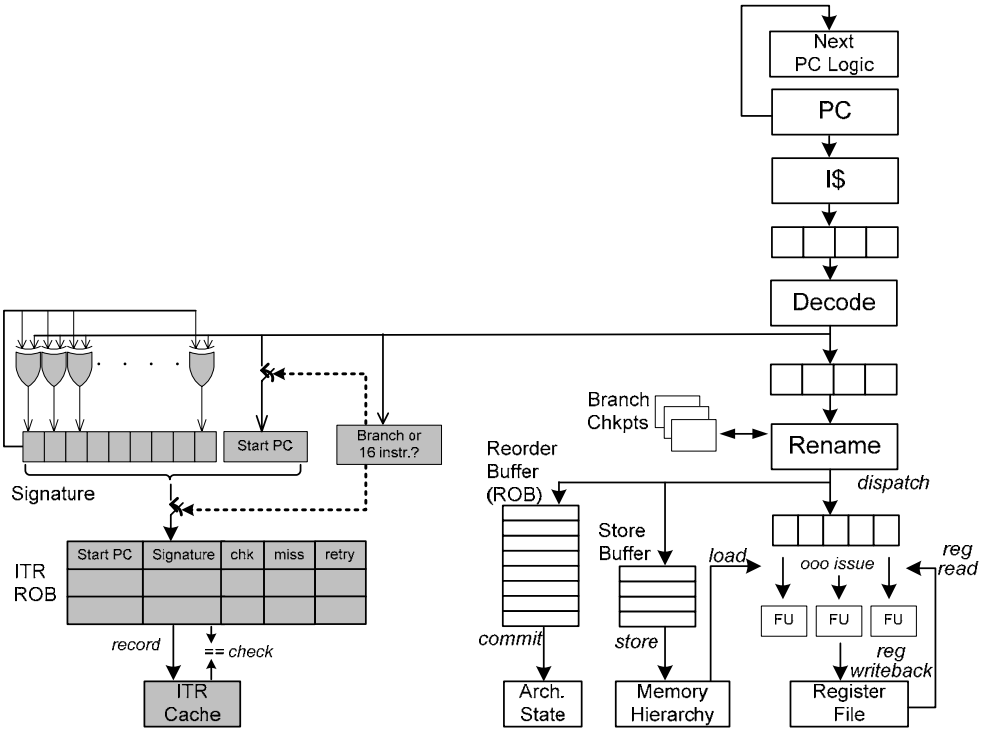

Figure 5. Superscalar processor augmented with ITR support.

ROB entry is noted in the branch's checkpoint to facilitate rollback to the correct ITR ROB entry on branch mispredictions.

Each ITR ROB entry stores the start PC and the signature of a trace. An ITR ROB entry also contains control bits (chk, miss, retry), which indicate the status of checking the trace with the copy in the ITR cache.

The ITR cache stores signatures of previously encountered traces and is indexed with the start PC of a trace. Each trace in the ITR ROB accesses the ITR cache at dispatch. This ensures that reading the ITR cache is complete before the instructions in the trace are ready to commit. If the trace hits, the signature is read from the ITR cache and checked with the signature of the trace. Regardless of the outcome, the chk (for checked) bit is set in the corresponding ITR ROB entry. If it's a mismatch, the retry bit of the ITR ROB entry is set. If the trace misses, the miss bit of the ITR ROB entry is set.

The ITR ROB enables the commit logic of the processor to determine whether the trace of the currently committing instruction has been formed, whether it is has been checked, whether it is faulty, etc. The only extra work for the commit logic is to poll the head entry of the ITR ROB when an instruction is ready to commit. It polls to see if the miss bit or the chk bit of the ITR ROB head entry is set. If neither is set, commit is stalled until one of the bits is set. If the miss bit is set, then a write to the ITR cache is initiated and commit from the main ROB progresses normally. If the chk bit is set, and additionally the retry bit is not set, then instructions are committed from the main ROB normally. If the retry bit is set, it indicates a transient fault occurred in either the new trace or the previous trace that stored its signature in the ITR cache. To confirm which trace instance is faulty, the processor is flushed and restarted from the start PC of the new trace. If the signatures mismatch again, then it is clear the previous trace executed with a fault. Since this means the processor's architectural state could be corrupted, a machine check exception is raised and the program is aborted. However, if the signatures match after the retry, it means the new trace was faulty, and recovery through flushing and restarting the processor was successful. In all cases, when a trace-terminating instruction is committed from the main ROB, the ITR ROB head entry is freed.

2.3. Fault detection and recovery coverage

Writing to the ITR cache involves replacing an existing, least recently used (LRU) trace signature. Evicting an existing trace signature has implications on the fault detection coverage, i.e., the number of instructions in which a fault can be detected. If a trace's signature is not referenced before being evicted, it amounts to a loss in fault detection coverage. To prevent this, a bit could be added to each cache line to

indicate that it is checked and the replacement policy could be modified to evict the LRU trace that has been checked. We do not study this optimization and instead report the loss in fault detection coverage for different cache configurations. Moreover, this policy is not applicable to direct mapped caches and breaks down when no ways of a set are checked yet.

ITR cache misses decrease the fault recovery coverage, i.e., the number of instructions in which a fault can be detected and successfully recovered by flushing and restarting the processor. This is because on a miss, an unchecked trace signature is entered into the cache. If the unchecked trace is faulty, the fault is only detected in the future by the next instance of the trace. However, since the faulty trace has already corrupted the architectural state, the program has to be aborted. In Section 3, we measure the fault coverage for different ITR cache configurations.

Recovery coverage can be enhanced through a coarse-grained checkpointing scheme (e.g., [6][7]). The key idea is to take a coarse-grain checkpoint when there are no unchecked lines in the ITR cache. The number of unchecked lines could be tracked. Once it reaches zero, a coarse-grain checkpoint could be taken. Then in cases where the lightweight processor flush and restart is not possible, recovery can be done by rolling back to the previously taken coarse-grain checkpoint instead of aborting the program.

2.4. Faults on ITR components

The new ITR components do not make the processor more vulnerable to faults, assuming a single-event upset model. A fault on signature generation components will be detected as a signature mismatch. A fault on the latched start PC is not a concern. If its signature matches the faulty start PC's signature, the fault gets masked. If it mismatches, the fault is detected. If it misses in the ITR cache, the next instance of the faulty PC will either detect it or mask it. The control bits chk, miss and retry can be protected using one-hot encoding. The possible states are: {none set – 0001, chk and retry set – 0010, chk set and retry not set—0100, miss set – 1000}. Faults on the ITR cache will cause false machine check exceptions when they are detected, i.e., a retry will indicate a fault on the trace signature in the ITR cache and a machine check exception will be raised, as described in Section 2.2. This can be avoided by parity-protecting each line in the ITR cache. On a signature mismatch, retry is attempted. If the signature mismatches again, then parity is checked on the trace signature in the cache. A parity error indicates an error in the ITR cache and not the previous instance of the trace. Successful recovery involves invalidating the erroneous line in the cache, or updating it with the signature of the new trace.

2.5. Faults on the program counter (PC)

A fault on the PC or the next-PC logic causes incorrect instructions to be fetched from the I-cache.

If the disruption is in the middle of a trace, then its signature will be a combination of signals from correct and incorrect instructions, and will differ from the trace's fault-free signature. In this case, a PC fault is detected by the ITR cache.

If the disruption is at a natural trace boundary, then a wrong trace is fetched from the I-cache. Since the signature of the wrong trace itself is unaffected by the fault, it will agree with the ITR cache. Hence, the PC that starts a trace at a natural trace boundary represents a vulnerability of the ITR cache, and needs other means of protection. For natural trace boundaries caused by branches, substantial protection of the PC already exists, because the execution unit checks branch targets predicted by the fetch unit. For natural trace boundaries caused by the maximum trace length, protection of the PC is possible by adding a simple commit PC and asserting that a committing instruction's PC matches the commit PC. The commit PC is updated as follows. Sequential committing instructions add their length (which can be recorded at decode for variable-length ISAs) to the commit PC and branches update the commit PC with their calculated PC. Comparing a committing instruction's PC with the commit PC will detect a discontinuity between two otherwise sequential traces. As part of future work, we plan to comprehensively study PC related fault scenarios to identify other potential vulnerabilities and devise robust solutions.

3. The ITR cache design space

As noted in Section 2.3, evictions of unreferenced lines from the ITR cache cause a loss in fault detection coverage, and misses in the ITR cache cause a loss in fault recovery coverage. In this section, we try different ITR cache configurations and measure the loss in fault detection coverage and fault recovery coverage for each design point. Loss in coverage is measured by noting the number of instructions in vulnerable traces.

For experiments, we ran SPEC2K integer and floating point benchmarks compiled with the Simplescalar gcc compiler for the PISA ISA [14]. The compiler optimization level is –O3. Reference inputs are used. In our runs, we skip 900 million instructions and simulate 200 million instructions.

Two ITR cache parameters are varied, (1) Associativity: direct mapped (dm), 2-way, 4-way, 8-way, 16-way and fully associative (fa), and (2) Cache size: 256, 512 and 1024 signatures. Figure 6 shows the loss in fault detection coverage and Figure 7 shows the loss in fault recovery coverage for the various cache configurations. For a given associativity, a smaller cache increases the number of evictions of unreferenced ITR signatures and the number of ITR cache misses. The corresponding increase in coverage loss is shown stacked for the various cache sizes.

Bzip, gzip, art, mgrid and wupwise have negligible coverage loss for all ITR cache configurations. For clarity, they are not included in the graphs. Their excellent ITR cache behavior can be explained by referring back to Figure 3 and Figure 4, which characterize ITR in benchmarks. In these benchmarks, traces repeat in close proximity and such traces contribute to nearly all the dynamic instructions.

In fact, coverage loss for all benchmarks correlates with their characteristics in Figure 3 and Figure 4. In perl and vortex, traces that repeat far apart contribute to a large number of dynamic instructions. Correspondingly, they have the highest loss in fault coverage. Cache capacity has a big impact on mitigating this loss. For example, in vortex, for a direct-mapped cache, increasing the cache capacity to 1024 signatures from 256 signatures decreases the loss in fault detection coverage to 12% from 33%.

Gcc, twolf and apsi also have a notable number of traces that repeat far apart, and experience a loss in fault coverage. They also benefit significantly from increasing the cache capacity. For insight, we refer to Table 1. It shows the total number of static traces for all benchmarks. Notice for vortex and perl, the number of static traces (2,655 and 1,704) is higher than the capacity of all the ITR caches simulated. Their poor trace proximity exposes this capacity problem. Far-apart repeating traces get evicted before they are accessed again, leading to a notable loss in fault coverage. Increasing the cache capacity somewhat makes up for the poor proximity and, hence, has a big impact on reducing coverage loss. Gcc confirms our hypothesis that proximity amongst traces is a strong factor. Even though it has far more traces than vortex and perl (24,017), it has lower coverage loss for a given cache configuration as a result of its better trace proximity. Mgrid is another example. It has negligible coverage loss for all ITR cache configurations even though it has a relatively high number of static traces (798). Again, proximity amongst its traces is excellent. The remaining benchmarks have a small loss in fault coverage which can be overcome with bigger caches or higher associativity.

Table 1. Number of static traces for SPEC.

SPECInt	#static
bzip	283
gap	696
gcc	24017
gzip	291
parser	865
perl	1704
twolf	481
vortex	2655
vpr	292

SPECfp	#static
applu	282
apsi	1274
art	98
equake	336
mgrid	798
swim	73
wupwise	18

Note that the loss in fault coverage should not be interpreted as a conventional cache miss rate, i.e., it does not correspond to signatures that missed on accessing the ITR cache. Firstly, the loss in fault detection coverage (Figure 6) corresponds to signatures that were evicted from the ITR cache before being referenced. Secondly, both the loss in fault detection coverage and the loss in fault recovery coverage are influenced by the number of instructions in signatures, which is not uniform across all signatures. These factors may explain why, in some benchmarks, higher associativity sometimes happens to show slightly higher loss in fault coverage than lower associativity.

An important point is that the loss in fault detection coverage is significantly lesser than the loss in fault recovery coverage for all benchmarks. This is because all ITR cache misses lead to a loss in recovery coverage, but only those missed traces that are then evicted before being referenced lead to a loss in detection coverage.

Across all benchmarks, for a 2-way associative cache with 1024 signatures, the average loss in fault detection coverage is 1.3% with a maximum loss of 8.2% for vortex. The corresponding numbers for loss in fault recovery coverage are 2.5% average and 15% maximum for vortex.

In general, programs with less repetition or greater distance between repeated traces would have a higher loss in fault coverage. One possible solution to mitigate this is to redundantly fetch and decode traces only on a miss in the ITR cache, still achieving the benefits of ITR but falling back on conventional time redundancy when inherent time redundancy fails. After the signature of the re-fetched trace is checked against the ITR cache, instructions in that trace are discarded from the pipeline. Another possible solution is to have a fully duplicated frontend, like in the IBM S/390 G5 processor [4], but use the ITR cache to guide when the space redundancy should be exercised (for significant power savings). The use of ITR as a filter for selectively exercising time redundancy or space redundancy is an interesting direction we want to explore in future research.

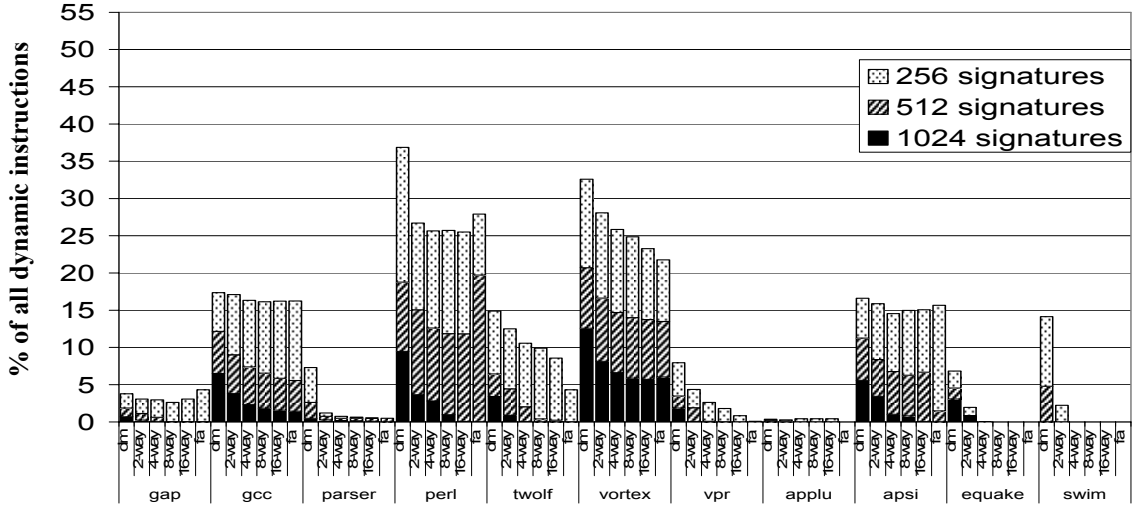

Figure 6. Loss in fault detection coverage.

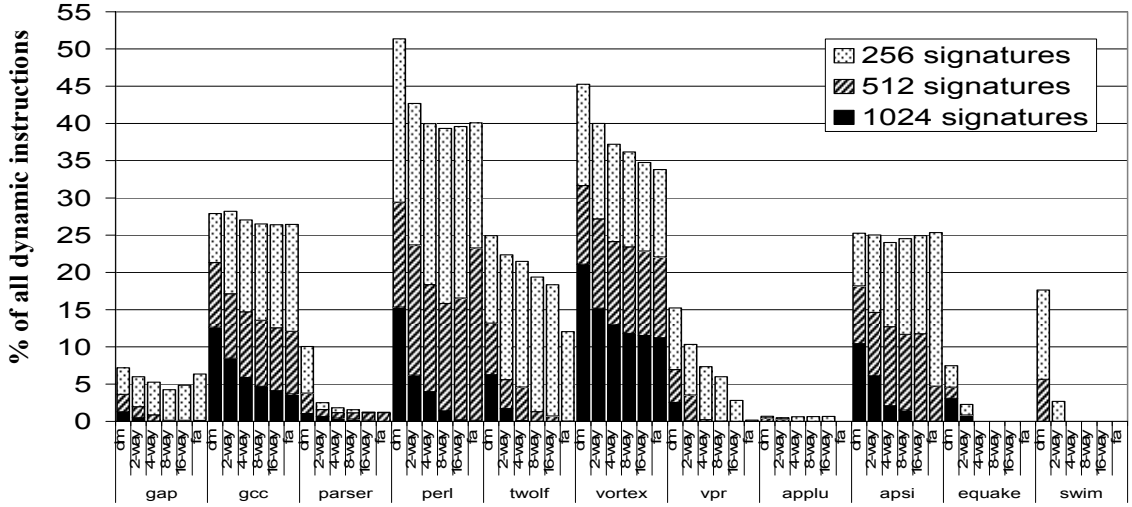

Figure 7. Loss in fault recovery coverage.

4. Fault injection experiments

We perform fault injection on a detailed cycle-level simulator that models a microarchitecture similar to the MIPS R10K processor [5].

For each benchmark, one thousand faults are randomly injected on the decode signals from Table 2. Injecting a fault involves flipping a randomly selected bit. A separate "golden" (fault-free) simulator is run in parallel 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.

An injected fault may lead to one of six possible outcomes, depending on (1) whether the fault is detected by an ITR check ("ITR") or undetected within the scope of the observation window ("MayITR")[1] or undetected for sure ("Undet"), and (2) whether the fault corrupts architectural state ("SDC") or not ("Mask"). Based on this, the six possible outcomes are ITR+SDC, ITR+Mask, MayITR+SDC, MayITR+Mask, Undet+SDC, and Undet+Mask.

[1] A fault may not get detected within the scope of the observation window, but its corresponding faulty signature may still be in the ITR cache. In this case, it is possible that the fault will be detected by ITR in the future, but we would have to extend the observation window to confirm this.

Table 2. List of decode signals.

Field	Description	Width
opcode	instruction opcode	8
flags	decoded control flags (is_int, is_fp, is_signed/unsigned, is_branch, is_uncond, is_ld, is_st, mem_left/right, is_RR, is_disp, is_direct, is_trap)	12
shamt	shift amount	5
rsrc1	source register operand	5
rsrc2	source register operand	5
rdst	destination register operand	5
lat	execution latency	2
imm	immediate	16
num_rsrc	number of source operands	2
num_rdst	number of destination operands	1
mem_size	size of memory word	3
	Total width	64

We further qualify ITR+SDC outcomes with the possibility of recovery (ITR+SDC+R) or only detection (ITR+SDC+D). On detecting a fault through ITR, if the signature accessing the ITR cache is faulty as opposed to the signature within the cache, then, the fault is recoverable by flushing the ROB (discussed in Section 2.3).

We add two more fault checks to support our experiments. A watchdog timer check (wdog) is added to detect deadlocks caused by some faults (e.g., faulty source registers). A sequential-PC check (spc) is added at retirement (discussed in Section 2.5) to detect faults pertaining to control flow.

In the following experiments, we use a two-way set-associative ITR cache holding 1024 signatures. The breakdown of fault injection outcomes is shown in Figure 8. We show fault injection results for the same set of SPEC benchmarks whose coverage results are reported in Section 3. As seen, a large percentage of injected faults are detected through the ITR cache (95.4% on average). On average, 32% of the injected faults are detected and recovered by ITR that would have otherwise led to a SDC (ITR+SDC+R). Only a small percentage (1% on average) of SDC faults detected through ITR is not recoverable (ITR+SDC+D). A large percentage of faults that are detected by ITR happen to get masked (59.4% on average). When a fault is injected on a decode signal that is not relevant to the instruction being decoded or does not lead to an error (e.g., increasing *lat*, the execution latency, only delays wakeup of dependent instructions), then the fault gets masked, but the signature is faulty and gets detected by the ITR cache. A noticeable fraction of faults (3% on average) are detected and recovered by ITR that would have otherwise led to a deadlock (ITR+wdog+R), highlighting another important benefit.

The fraction of faults undetected by ITR within the observation window (MayITR+*) is negligible. This indicates that a one million cycle observation window is sufficient.

Interestingly, the sequential PC check detected a small fraction of faults (0.1% on average) that ITR alone could not detect (spc+SDC). The sequential-PC check mainly detected faults on the *is_branch* control flag, which indicates whether or not an instruction is a conditional branch. Consider the following fault scenario. Suppose that the fetch unit predicts an instruction to be a conditional branch (BTB hit signals a conditional branch and gshare predicts taken). Suppose the instruction is truly a conditional branch (BTB correct) and is actually not taken (gshare incorrect). Then suppose that a fault causes *is_branch* to be false instead of true. First, this fault causes a SDC because the branch misprediction will not be repaired. Second, because *is_branch* is false, the retirement PC is updated in a sequential way. The spc check will fire in this case, because the next retiring instruction is not sequential. Note that if the prediction was correct (actually taken), the spc check still fires, but this is a masked rather than SDC fault.

On average, 4.5% of injected faults go undetected by ITR. Only about 2.6% of the faults lead to SDC and are not detected by ITR (Undet+SDC). A very small fraction of faults (0.1% on average) lead to a deadlock that is not detected by ITR but is caught by the watchdog timer. The remaining undetected faults are masked (on average, 1.8% of all faults).

5. Area and power comparisons

Structural duplication can be used to protect the fetch and decode units of the processor. In the IBM S/390 G5 processor [4], the I-unit, comprised of the fetch and decode units, is duplicated and signals from the two units are compared to detect transient faults. However, this direct approach has significant area and power overheads. We attempt to compare the area and power overhead of the ITR cache with that of the I-unit, to see whether or not the ITR-based approach is attractive compared to straightforward duplication. The die photo of the IBM S/390 G5 provides the area of the I-unit [4]. To estimate the area of the ITR cache, a structure is selected from the die photo that is similar in configuration to the ITR cache. The branch target buffer (BTB) of the G5 has a configuration similar to the ITR cache: 2048 entries, 2-way associative, 35 bits per entry [15]. Based on the decode signals in Table 2, the size of the ITR signature is 64 bits. Though each ITR entry is almost twice as wide as the G5's BTB entry, only half as many entries as the BTB (1024 entries) are needed for good coverage, from results in Section 3 and Section 4.

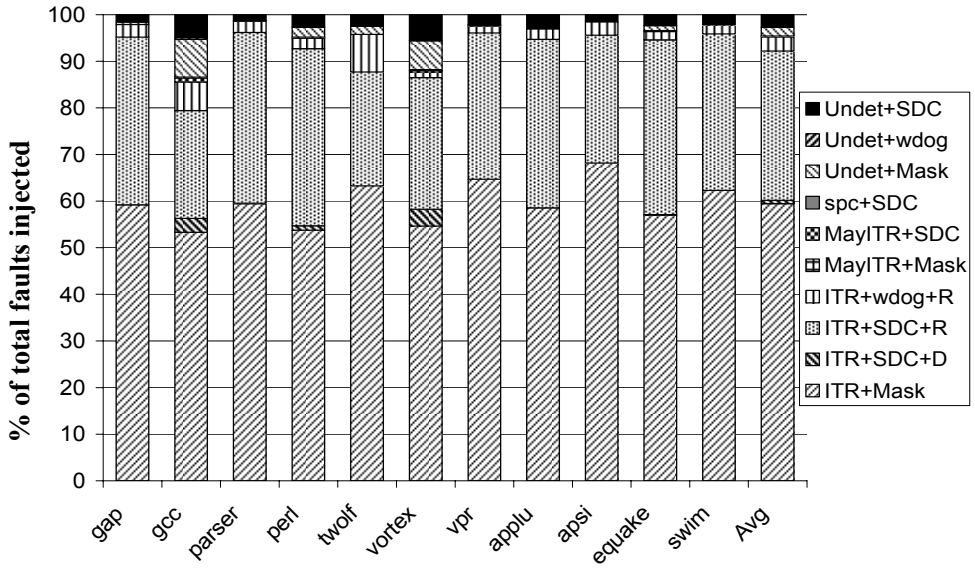

Figure 8. Fault injection results.

The area of the I-unit from the die photo is 1.5 cm x 1.4 cm, i.e., 2.1 cm^2. The area of the ITR-cache like BTB structure from the die photo is 1.5 cm x 0.2 cm, i.e., 0.3 cm^2. The ITR cache is about one seventh the area of the I-unit. Hence, the ITR-based approach to protect the frontend is more area-effective than structural duplication of the entire I-unit.

We next try to find the power-effectiveness of the ITR approach. A major power overhead of structural duplication and conventional time redundancy is that of fetching an instruction twice from the instruction cache. We model power consumption by measuring the number of accesses to the ITR cache and the instruction cache of the processor. Both cache models are fed into CACTI [17] to obtain the energy consumption per access. Multiplying the number of accesses with the energy consumed per access gives us the energy consumption.

Due to lack of information on the instruction cache configuration of the IBM S/390 G5, we chose the instruction cache of the IBM Power4 [16]. The configuration of the Power4 I-cache is: 64KB, direct-mapped, 128 byte line and one read/write port. The configuration of the ITR cache is: 8KB (1024 entries), 2-way associative, 8 byte line, and one read/write port (or one read and one write port). We chose the 0.18 micron technology used in the IBM Power4.

The CACTI numbers were: 0.87 nJ per access for the I-cache, 0.58 nJ per access (or 0.84 nJ for separate read and write ports) for the ITR cache. Overall energy consumption is shown in Figure 9. As seen, the ITR-based approach is far more energy efficient than fetching twice from the instruction cache. Note that the energy savings will be even greater if also considering the redundant decoding of instructions in the frontend in the case of structural duplication or traditional time redundancy.

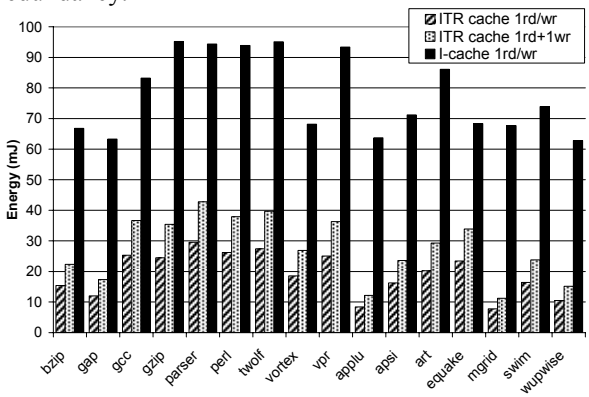

Figure 9. Energy of ITR cache vs. I-cache.

We see that the ITR cache is more cost-effective than straightforward space redundancy in the IBM mainframe processor [4]. However, it should be noted that complete structural duplication provides more robust fault tolerance than the ITR cache. They are two different design points in the cost/coverage spectrum.

6. Related work

Prior research on exploiting program repetition has focused on reusing previous instruction results through a reuse buffer to reduce the total number of instructions executed [1][2]. Instruction reuse has also been used to reduce the number of redundant instructions executed in a time-redundant execution

0-7695-2855-4/07 $20.00

model [8]. In the latter work, the goal was to reduce function unit pressure. Instead of executing two copies of an instruction using two function units, in some cases it is possible to execute one copy using a function unit and the other copy using a reuse buffer. ITR reduces pressure in the fetch and decode units, whereas their approach requires fetching and decoding all instructions twice. In other words, their approach only addresses the execution stage and is an orthogonal technique that could be used in an overall fault tolerance regimen.

Amongst the several proposals to reduce overheads of full-redundant execution, using ITR to protect the fetch and decode units could improve approaches that either do not offer protection to the frontend [9][12], or trade performance for protection by using traditional time-redundancy in the frontend [10][11]. In general, frontend bandwidth is pricier than execution bandwidth. By using ITR to protect the frontend, traditional time-redundancy can be focused on exploiting idle execution bandwidth [10][11][12][13].

ITR-based fault checks augment the suite of fault checks available to processor designers. Developing such a regimen of fault checks to protect the processor (e.g., [3]) will lead to low-overhead fault tolerance solutions compared to more expensive space redundancy or time redundancy approaches.

7. Summary

We introduced a new approach to develop low-overhead fault checks for a processor, based on inherent time redundancy (ITR) in programs. We proposed the ITR cache to store microarchitectural events that depend only upon program instructions. We demonstrated its effectiveness by developing microarchitectural support to protect the fetch and decode units of the processor. We gave insights on diagnosing a fault to determine the correct recovery procedure. We quantified fault detection coverage and fault recovery coverage obtained for a given ITR cache configuration. Finally, we showed that using the ITR-based approach is more favorable than costly structural duplication and traditional time redundancy.

8. Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments in improving this paper. We thank Muawya Al-Otoom and Hashem Hashemi for their help with area and power experiments. 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.

9. References

[1] A. Sodani and G. S. Sohi. Dynamic instruction reuse. ISCA 1997.

[2] A. Sodani and G. S. Sohi. An empirical analysis of instruction repetition. ASPLOS 1998.

[3] V. K. Reddy, A. S. Al-Zawawi and E. Rotenberg. Assertion-based microarchitecture design for improved fault tolerance. ICCD 2006.

[4] T. J. Slegel et al. IBM's S/390 G5 microprocessor design. IEEE Micro, March 1999.

[5] K. C. Yeager. The MIPS R10000 superscalar processor. IEEE Micro, April 1996.

[6] R. Teodorescu, J. Nakano and J. Torrellas. SWICH: A prototype for efficient cache-level checkpoint and rollback. IEEE Micro, Oct 2006.

[7] D. Sorin, M. M. K. Martin and M. D. Hill. Fast checkpoint/recovery to support kilo-instruction speculation and hardware fault tolerance. Tech. Report: CS-TR-2000-1420, Univ. of Wisconsin, Madison. Oct 2000.

[8] A. Parashar, S. Gurumurthi and A. Sivasubramaniam. A complexity effective approach to ALU bandwidth enhancement for instruction-level temporal redundancy. ISCA 2004.

[9] T. M. Austin. Diva: A reliable substrate for deep submicron microarchitecture design. MICRO 1999.

[10] J. Ray, J. C. Hoe and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. MICRO 2001.

[11] J. C. Smolens, J. Kim, J. C. Hoe and B. Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. MICRO 2004.

[12] A. Mendelson and N. Suri. Designing high-performance and reliable superscalar architectures – The out of order reliable superscalar (O3RS) approach. DSN 2000.

[13] M. Franklin, G. S. Sohi and K. K. Saluja. A study of time-redundant techniques for high-performance pipelined computers. FTCS 1989.

[14] D. Burger, T. Austin and S. Bennett. The simplescalar toolset, version 2. Tech Report CS-TR-1997-1342, Univ. of Wisconsin, Madison. July 1997.

[15] M. A. Check and T. J. Slegel. Custom S/390 G5 and G6 microprocessors. IBM Journal of R&D, vol 43, #5/6. 1999.

[16] J. M. Tendler et al. Power4 system microarchitecture. IBM Journal of R&D, vol 46, #1, 2002.

[17] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An Integrated Cache Timing, Power and Area Model. Western Research Lab (WRL) Research Report. 2002.