Understanding Prediction-Based Partial Redundant Threading for Low-Overhead, High-Coverage Fault Tolerance

Vimal K. Reddy Sailashri Parthasarathy * Eric Rotenberg

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695
{vkreddy, ericro}@ece.ncsu.edu

* Architecture Modeling Infrastructure Group
Intel Corporation
Hudson, MA 01749
sailashri.parthasarathy@intel.com

Abstract

Redundant threading architectures duplicate all instructions to detect and possibly recover from transient faults. Several lighter weight Partial Redundant Threading (PRT) architectures have been proposed recently. (i) Opportunistic Fault Tolerance duplicates instructions only during periods of poor single-thread performance. (ii) ReStore does not explicitly duplicate instructions and instead exploits mispredictions among highly confident branch predictions as symptoms of faults. (iii) Slipstream creates a reduced alternate thread by replacing many instructions with highly confident predictions. We explore PRT as a possible direction for achieving the fault tolerance of full duplication with the performance of single-thread execution. Opportunistic and ReStore yield partial coverage since they are restricted to using only partial duplication or only confident predictions, respectively. Previous analysis of Slipstream fault tolerance was cursory and concluded that only duplicated instructions are covered. In this paper, we attempt to better understand Slipstream's fault tolerance, conjecturing that the mixture of partial duplication and confident predictions actually closely approximates the coverage of full duplication. A thorough dissection of prediction scenarios confirms that faults in nearly 100% of instructions are detectable. Fewer than 0.1% of faulty instructions are not detectable due to coincident faults and mispredictions. Next we show that the current recovery implementation fails to leverage excellent detection capability, since recovery sometimes initiates belatedly, after already retiring a detected faulty instruction. We propose and evaluate a suite of simple microarchitectural alterations to recovery and checking. Using the best alterations, Slipstream can recover from faults in 99% of instructions, compared to only 78% of instructions without alterations. Both results are much higher than predicted by past research, which claims coverage for only duplicated instructions, or 65% of instructions. On an 8-issue SMT processor, Slipstream performs within 1.3% of single-thread execution whereas full duplication slows performance by 14%.

A key byproduct of this paper is a novel analysis framework in which every dynamic instruction is considered to be hypothetically faulty, thus not requiring explicit fault injection. Fault coverage is measured in terms of the fraction of candidate faulty instructions that are directly or indirectly detectable before retirement. This framework provides a reliable means to compare coverage of different PRT approaches, avoiding pitfalls of incomplete fault injection experiments. Moreover, one simulation does the work of very many fault injection experiments.

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

Categories and Subject Descriptors B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance. C.1.0 [**Processor Architectures**]: General. C.4 [**Performance of Systems**] – *fault tolerance*.

General Terms Design, Performance, Reliability.

Keywords Simultaneous multithreading (SMT), chip multiprocessor (CMP), slipstream processor, transient faults, time redundancy, redundant multithreading, branch prediction, value prediction.

1. Introduction

Redundant multithreading architectures [15][16][19][25] fetch and execute all dynamic instructions in a program twice, via two redundant threads on a simultaneous multithreading (SMT) pipeline, to detect and possibly recover from single transient faults that affect the pipeline. The first and second results of each duplicated instruction are compared. A difference indicates a transient fault occurred in the pipeline. Recovery may be possible, by preventing retirement of the first faulty instruction in one [19] or both threads [25] and restarting both threads from this instruction.

The performance overhead of dual redundant threads on an SMT pipeline can be significant due to resource contention (fetch, issue, and retire bandwidth, physical registers, etc.) and checking bandwidth. Various techniques have been proposed for reducing resource pressure, involving staggering the two threads and exploiting the leading thread's outcomes to reduce the trailing thread's resource utilization [6][16][19]. Checking bandwidth can be relieved by reducing the number of result comparisons [16][25].

While the techniques above improve the performance of redundant threads, they still duplicate all dynamic instructions.

Lighter weight approaches have been proposed [4][23][26], that duplicate only a subset of the dynamic instruction stream. We refer to these approaches as Partial Redundant Threading (PRT) and discuss them in the context of the PRT spectrum shown in Figure 1.

- *Partial duplication*. Opportunistic Fault Tolerance [4] duplicates instructions only during periods of poor single-thread performance (e.g., during L2 cache misses or low instruction-level parallelism).
- *Confident predictions*. ReStore [26] does not explicitly duplicate instructions. Instead, highly confident branch

predictions are used to indirectly detect faults. Since a highly confident branch prediction is very likely correct, a "misprediction" may indicate, not that the prediction is wrong, but rather that a fault afflicted the branch or an instruction in the backward slice of the branch. Whether due to a fault or a misprediction, the processor rolls back to a prior register/memory checkpoint. If there was a fault, rolling back masks the fault if the original faulty instruction is logically after the checkpoint (i.e., not yet retired) and therefore the checkpoint is not corrupted.

- *Partial duplication and confident predictions*. Slipstream [13][23] combines partial duplication and confident predictions. A second reduced thread is created by (i) removing predictable branches and their backward slices, replacing them with highly confident branch predictions, (ii) removing predictable dynamically-dead instructions and their backward slices, and (iii) removing predictable silent writes (they do not change the value in a location) and their backward slices, implicitly replacing them with highly confident value predictions.

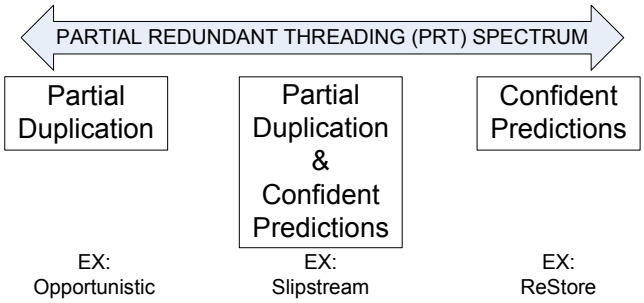


Figure 1. Partial Redundant Threading (PRT) spectrum.

The goal of this paper is to determine whether or not PRT can achieve the coverage of full duplication with the performance of single-thread execution, or determine how close PRT can get to this ideal. From a performance standpoint, PRT approaches are more efficient than full duplication and may provide a path towards achieving single-thread performance. From a fault tolerance standpoint, Opportunistic and ReStore provide only partial coverage. The only previous work on Slipstream fault tolerance is cursory and concluded "the system transparently recovers from transient faults affecting redundantly-executed instructions" [23][13], i.e., only duplicated instructions are covered. In this paper, we attempt to better understand Slipstream's fault tolerance, conjecturing for the first time that the mixture of partial duplication and confident predictions actually closely approximates the coverage of full duplication. In particular, the contributions of this paper are as follows.

- We provide a thorough dissection of the four prediction scenarios in slipstream, illuminating cases in which faulty instructions are detectable vs. undetectable. This analysis shows, for the first time, that slipstream is able to detect faults in nearly 100% of dynamic instructions. Fewer than 0.1% of faulty instructions are undetectable due to the coincidence of a fault and an incorrect confident prediction, two flawed counterparts which reinforce each other.
- We identify weaknesses in slipstream's current recovery implementation that cause it to fall far short of 100% instruction coverage despite excellent detection capability, and propose and evaluate three simple microarchitectural alterations to recovery. The problem is that recovery may initiate too late, after already retiring the faulty instruction which it is supposed to prevent from retiring. With the best alterations, slipstream can recover from faults in 99% of instructions, compared to only 78% of instructions without alterations. Both results are higher than predicted by past work, which claims coverage for only duplicated instructions, or 65% of instructions.
- We identify opportunities for earlier checks of silent writes, further increasing chances that recovery occurs prior to retiring a faulty instruction.
- We develop a novel analysis framework in which every dynamic instruction is considered to be hypothetically faulty, i.e., all instructions are "candidate faulty instructions". This approach is equivalent to separate fault injection experiments for each and every instruction, all in the same experiment, avoiding pitfalls of incomplete random fault injection experiments and doing the work of many experiments with one simulation. Fault tolerance is measured in terms of the fraction of candidate faulty instructions that are directly or indirectly detectable before retirement. This is the key criterion for successful recovery: the first faulty instruction must be detected before it retires, so that the architectural state is a safe checkpoint for recovery. Our analysis framework and instruction coverage metric enable a fair comparison of different PRT approaches.
- We provide in-depth quantitative analysis of slipstream fault tolerance and performance for the suite of new check and recovery alterations mentioned above. The best implementation yields 99% instruction coverage within 1.3% of single-thread performance, compared to 100% instruction coverage and 14% performance loss with full duplication, on average.
- To demonstrate the wider applicability of principles articulated in this paper, we provide in-depth analysis of performance/coverage trade-offs of a ReStore-like architecture that uses only branch predictions for detection.

Corresponding to the first contribution above, Section 2 analyzes the fault detection capabilities of the three PRT approaches. This includes detailed analysis of slipstream's four prediction scenarios. Corresponding to the second and third contributions above, Section 3 discusses problems with slipstream's recovery implementation and proposes recovery alterations and early check optimizations to better capitalize on slipstream's fault detection capability. In Section 4, we explain our novel analysis framework for measuring fault tolerance in terms of instruction coverage. Simulation environment is described in Section 5. Results are presented in Section 6. Related work is discussed in Section 7. Finally, we summarize the paper in Section 8.

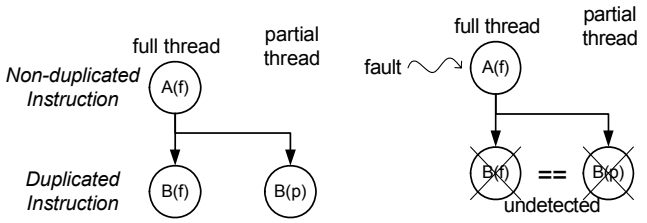
2. Fault Detection

In order to recover from faults, they must first be detected. In this section we analyze the fault detection capabilities of each of the three PRT approaches.

2.1 Opportunistic: Partial Duplication

Partial duplication produces a full thread and a partial thread. For the example in Figure 2, instruction A is not duplicated and

instruction B is duplicated. Thus, the full thread consists of instructions A(f) and B(f) and the partial thread consists of only B(p). Throughout this paper, "(f)" denotes an instruction in the full thread and "(p)" denotes an instruction in the partial thread.



(a) Dependence between full and partial threads. **(b) Fault in non-duplicated instruction is undetectable.**

Figure 2. PRT using only partial duplication.

The partial thread requires some results from the full thread, to compensate for its missing instructions, as shown in Figure 2(a). Thus, there are dependences between the full thread and the partial thread. As explained below, these dependences are the reasons for loss in coverage.

A fault that affects a duplicated instruction is detectable. For example, a fault that affects either B(f) or B(p) is detectable because the two instances will differ.

However, a fault that affects a non-duplicated instruction is not detectable because (1) there is no direct counterpart for comparison and (2) the fault cannot be detected indirectly by duplicated consumers. For example, a fault that affects A(f) is not detectable, because its faulty result is consumed by both B(f) and B(p), as shown in Figure 2(b). Thus, both instances of the consumer instruction B will be flawed in the same way, and their comparison will not detect a problem.

2.2 ReStore: Confident Predictions

As explained in Section 1, ReStore executes only one thread and some faults are detected as "mispredictions" among confident branch predictions. Figure 3 shows an example, where the thread consists of a branch instruction (br) and its producer instruction. The corresponding branch prediction is confident. If the confident prediction is correct (very likely), it can detect a fault in either the branch or its producer. This is because the faulty branch outcome will differ from the correct prediction, as shown in Figure 3(a). On the other hand, if the confident prediction is incorrect, it will not detect a fault in either instruction since the faulty branch outcome and the incorrect prediction match, as shown in Figure 3(b). Generalizing, ReStore can detect faults among branches and their backward slices, that are confidently and correctly predicted.

In ReStore, the confidence threshold controls coverage and performance. Coverage is maximized when the confidence threshold is zero, i.e., when all branch predictions are deemed confident. In this case, faults are detectable among all correctly predicted branches and their backward slices. This is the upper bound on coverage for ReStore using only branch symptoms, i.e., not counting exceptions and other symptoms [26]. Unfortunately, performance is significantly degraded in this case, because all mispredictions (not just mispredictions among confident predictions) cause rollbacks to a checkpoint.

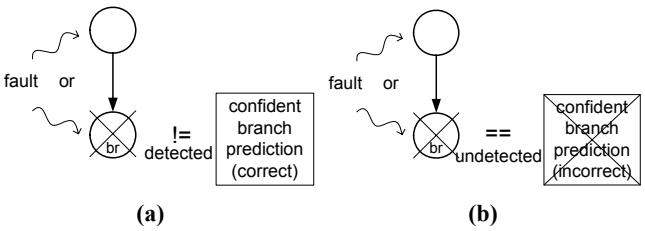


Figure 3. PRT using only confident predictions.

2.3 Slipstream: Partial Duplication & Confident Predictions

In Section 2.1, we explained that partial duplication loses coverage due to dependences between the full thread and partial thread.

To achieve full coverage with partial duplication, the partial thread must be independent of the full thread. The key idea is to only exclude those dynamic instructions from the partial thread, whose effects can be confidently emulated by predictions. In general, this implies replacing predictable instructions and their backward slices with confident branch and value predictions. Confident proxy predictions make it possible to fetch and execute the partial thread self-sufficiently, without relying on the full thread for unchecked results. The partial thread is speculative, yet the combination of highly confident prediction and execution yields control-flow and data-flow outcomes that are almost always equal to corresponding outcomes of the full thread, in the fault-free case. Because the partial thread is accurate and independent, it is nearly as good a redundant counterpart as a second full thread.

This prediction-based PRT paradigm is depicted in Figure 4. The key difference from Section 2.1 and Figure 2 (partial duplication only) is that the partial thread is no longer influenced by unchecked outcomes from the full thread. Assuming the speculative partial thread is correct in the fault-free case, this separation makes it possible to detect a single transient fault in any one of the three instructions shown, even the singly executed producer instruction A(f) in the full thread. As shown, a fault in A(f) is inherited by the full thread's instance of the redundantly executed consumer, B(f), but not by the partial thread's instance, B(p). Thus, the two instances B(f) and B(p) will differ, indirectly detecting the original fault.

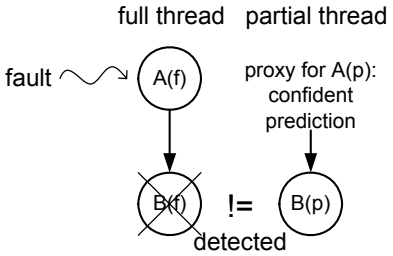


Figure 4. PRT using partial duplication and confident predictions.

So far we have discussed prediction-based PRT generically. The slipstream implementation in this paper specifically exploits four types of prediction to form the partial thread.

1. *Branch prediction*. Highly predictable branches are removed.

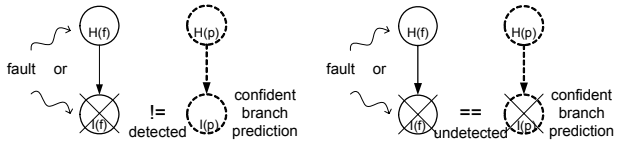
2. *Silent write prediction*. This is a special case of value prediction. Instructions that predictably write the same value into a logical register as the previous write to the logical register are removed. The implied value prediction is the value in the logical register prior to writing it.
3. *Dead write prediction*. Predictably dead register-writing instructions are removed.
4. *Silent store prediction*. Store instructions that predictably store the same value into a memory location as the previous store to the location are removed.

Instructions in the backward slices of confidently predicted branches, silent writes, dead writes, and silent stores are also removed. However, note that a backward-slice instruction is only removed if all of its consumers are removed [20][23][13][5].

In the following subsections, we analyze each of the four prediction scenarios in depth to understand when faults are detectable vs. undetectable. The key idea is to consider both correct prediction and incorrect prediction. The analysis reveals that faulty instructions are undetectable only when faults coincide with mispredictions. Fortunately, mispredictions among confident predictions are very rare.

2.3.1 Confident Branch Prediction

Figure 5 shows a confidently predicted branch I and its producer H. Annotations (f) and (p) denote which thread an instruction belongs to, full or partial, respectively. The full thread has instances H(f) and I(f). Counterparts H(p) and I(p) are removed from the partial thread, as indicated by dashed circles and arcs, and replaced with a confident branch prediction. This confident branch prediction becomes the redundant counterpart of I(f). I(f) produces the wrong branch outcome due to a transient fault in either H(f) or I(f), depicted by an X over I(f). If the confident branch prediction is correct, then it will differ from the incorrect branch outcome of I(f) as shown in Figure 5(a), thus detecting a faulty H(f) or I(f). However, if the confident branch prediction is incorrect, depicted by an X over I(p), then it will match the incorrect branch outcome of I(f) as shown in Figure 5(b), failing to detect a faulty H(f) or I(f).

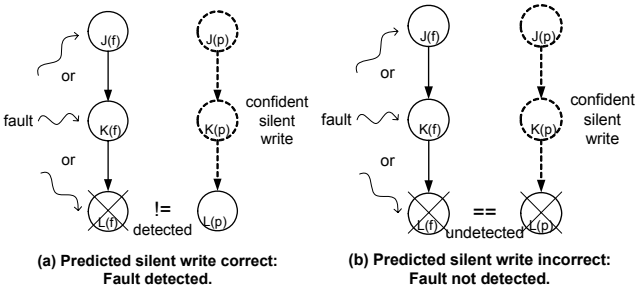


(a) Correct prediction for removed branch: Fault detected. **(b) Incorrect prediction for removed branch: Fault not detected.**

Figure 5. Confidently predicted branch removed from partial thread.

2.3.2 Confident Silent Write Prediction

Figure 6 shows a confidently predicted silent write K, and its producer J and consumer L. The full thread has instances J(f), K(f), and L(f). The predicted silent write K(p) and its producer J(p) are removed from the partial thread, as indicated by their dashed circles and arcs. They are implicitly replaced with a confident value prediction that is the value produced by the last writer of K's logical destination register. That is, the consumer of the predicted silent write, L(p), remains in the partial thread and it is predictively renamed to the previous writer of K's register (not shown) instead of K(p) itself. As such, L(p) becomes a predictive redundant counterpart of L(f). L(f) produces a wrong result due to a transient fault in either J(f), K(f), or L(f), depicted by an X over L(f). If K(p) is truly a silent write, then L(p) produces the correct result, thus detecting a faulty J(f), K(f), or L(f), as shown in Figure 6(a). However, if K(p) is not actually a silent write, then L(p) may produce an incorrect result since it uses a stale value instead of the present value from K(p). If L(p) and L(f) are incorrect in exactly the same way, a faulty J(f), K(f), or L(f) is not detected, as shown in Figure 6(b).

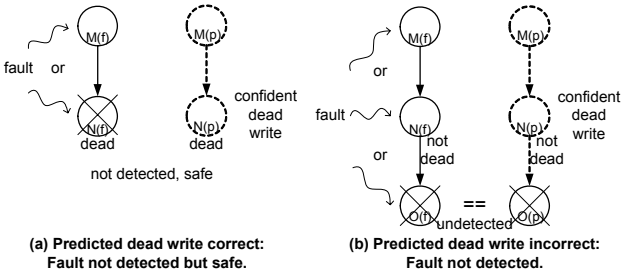


(a) Predicted silent write correct: Fault detected. **(b) Predicted silent write incorrect: Fault not detected.**

Figure 6. Confidently predicted silent write removed from partial thread.

2.3.3 Confident Dead Write Prediction

Figure 7 shows a confidently predicted dead write N and its producer M. The full thread has instances M(f) and N(f). The predicted dead write N(p) and its producer M(p) are removed from the partial thread, as indicated by their dashed circles and arcs. N(f) produces a wrong result due to a transient fault in either M(f) or N(f), depicted by an X over N(f). If N is truly dead as in Figure 7(a), then a faulty M(f) or N(f) will not be detected, because there is no redundant counterpart for N(f) to compare against nor is there a redundantly executed consumer of N(f) to do an indirect comparison. However, an undetected faulty M(f) or N(f) is safe for this very reason – it is not referenced in the future. If N is actually live as in Figure 7(b), then an unforeseen consumer O is brought into the picture. O(f) produces a wrong result due to a transient fault in either M(f), N(f), or O(f), depicted by an X over O(f). The partial thread's counterpart, O(p), may also produce a wrong result because it consumes a stale value, instead of the value that was supposed to be produced by the removed, presumed dead write N(p). If O(f) and O(p) produce the same wrong result, then a faulty M(f), N(f), or O(f) is not detected, as shown in Figure 7(b).



(a) Predicted dead write correct: Fault not detected but safe. **(b) Predicted dead write incorrect: Fault not detected.**

Figure 7. Confidently predicted dead write removed from partial thread.

2.3.4 Confident Silent Store Prediction

The analysis for silent store predictions mirrors that of silent write predictions in Section 2.3.2 and Figure 6, where instruction K is a store and instruction L is a dependent load instruction. However, the store K(f) may be faulty in two ways: faulty value or faulty address. Figure 6 only shows the faulty value case. Nonetheless, a faulty store address merely causes the store K(f) to store to a different address than the load L(f), leading to the same situation of a faulty load L(f).

2.3.5 Result: % Undetectable Faulty Instructions

Since instruction-removal mispredictions are rare in slipstream (due to conservative confidence), we expect very few mispredictions and correspondingly low loss in coverage. In all benchmarks studied, we measure losses in coverage below 0.1%. Therefore, slipstream can detect faults in 99.9% of instructions.

Fault recovery is a separate issue. A good recovery implementation will capitalize as much as possible on the fault detection capability. We discuss recovery in the next section.

3. Fault Recovery

We now discuss slipstream's fault recovery capability, given its excellent fault detection capability. Section 3.1 reviews the previous recovery implementation and discusses its weaknesses. Section 3.2 proposes recovery improvements. Section 3.3 proposes early (direct) checks of silent writes/stores to improve the chances for successful (i.e., timely) recovery.

3.1 Previous Recovery Implementation

The slipstream recovery implementation described in previous papers [13][23] works as follows. When a duplicated instruction detects a fault, it posts a fault exception and the slipstream processor waits for the exception to reach the head of the reorder buffer (ROB). Then, the full and partial threads are re-synchronized and restarted from the faulty duplicated instruction. This recovery implementation attributes the fault to the duplicated instruction. However, it may be that its non-duplicated producer is faulty. Suppose this is the case. The above delayed recovery model permits the faulty producer to retire, corrupting the full thread's architectural state which is supposed to be a safe checkpoint for recovery. When recovery is initiated, the partial thread's architectural state inherits the flawed full thread's architectural state. Now, both threads are architecturally corrupted and the system is potentially unrecoverable, depending on whether or not the corrupt state is architecturally masked in the future.

3.2 Recovery Alterations

3.2.1 ROB-head Recovery

The problem with the previous recovery implementation is not so much delaying recovery until the detected fault reaches the head of the ROB, as restarting the threads from the faulty consumer instead of from the faulty producer. In other words, previous slipstream implements "consumer recovery" instead of "producer recovery".

Producer recovery is more conservative since it always attributes a fault to the producer. However, the overhead of extra squashed instructions is a small price to pay for recovery from faults in non-duplicated instructions, the primary objective of this paper. In practice the overhead is negligible, since slipstream mispredictions are rare thus recovery is rarely initiated.

Implementing producer recovery explicitly would require dependence vectors, so that a faulty consumer could mark its direct/indirect producers as potentially faulty. Instead, we propose a hardware-free approach that emulates producer recovery (achieves the same coverage). Namely, the two threads are restarted from the oldest instruction in the ROB (ROB head) at the time the fault is detected. Recovery succeeds if the original faulty instruction is still in the ROB when the fault is detected, since it is prevented from retiring in this case.

3.2.2 ROB Occupancy Threshold

Slipstream staggers its two threads. Staggering enables the trailing thread to use outcomes from the leading thread as mostly perfect branch and value predictions [19][16]. Breaking dependences in the trailing thread causes it to fetch and execute more efficiently. In an SMT implementation, this releases resources back to the leading thread and thereby reduces overall execution time for the dual threads.

In our case, efficiency means low occupancy of the ROB by the full thread (the trailing thread). While this is good for performance, it reduces the effectiveness of ROB-head Recovery. Low ROB occupancy means a faulty non-duplicated instruction is more likely to retire before it is detected by a consumer. ROB-head Recovery is unsuccessful in this case.

One countermeasure for boosting ROB occupancy is to target a ROB occupancy threshold (e.g., 32 or 48 instructions). The target is met and maintained by throttling full thread retirement. We study the performance/coverage trade-off of various ROB occupancy thresholds, in the results section.

3.2.3 History Buffer

One way to increase the success rate of producer recovery without delaying retirement is to use a history buffer [21]. Before an instruction retires from the ROB and commits its value to the architectural register/memory state, the previous committed value is read and stored in the history buffer. When a fault is detected, the ROB is flushed as before (ROB-head Recovery). In addition, the architectural state is restored to an even earlier precise state by stepping through the history buffer in reverse and using the saved values to undo changes to the architectural state. If the original faulty instruction had retired from the ROB but not the history buffer, then the faulty committed value is safely removed.

The history buffer is a potentially simpler alternative than register and memory checkpoints advocated by ReStore but the principle is the same.

3.3 Direct Checks of Silent Writes and Stores

Since silent writes and stores are removed from the partial thread, their counterparts in the full thread have nothing to compare with. Therefore, faults affecting these instructions in the full thread are detected by future duplicated consumers (e.g., instruction L in Figure 6a). An indirect check is not useful if it is too late to prevent retiring the original faulty instruction.

Fortunately, direct checks are possible for silent writes and stores. First, the slipstream components that facilitate instruction removal (IR-detector and IR-predictor) can be augmented to remember the reason for speculatively removing an instruction. Therefore,

predicted silent writes and stores can be explicitly marked in the full thread (e.g., K(f) in Figure 6 is marked for direct checking). Second, predicted silent writes and stores can be directly checked in the full thread by comparing the value being written or stored with the value already in the register or memory location. If the values differ, either the prediction is wrong (not truly silent) or the silent instruction is faulty. Either way recovery is needed (to repair the partial thread or to mask the fault, respectively).

3.3.1 Methods for Directly Checking Silent Writes

Directly checking a predicted silent write in the full thread requires obtaining the value of the previous write to the same logical register. We propose and evaluate five approaches listed below. We advocate the fourth approach (ORT_ret) because it essentially comes for free in the existing slipstream implementation and results show it increases coverage about as well as the other approaches.

All approaches except ORT_ret require knowing the previous physical register mapping of the logical register since it is this physical register that will have the previous value. In some contemporary superscalar processors, the previous mapping is already read from the rename map table before updating it with the new mapping.

- RF: When a predicted silent write issues, it indexes the physical register file using the previous mapping, to obtain the previous value of the logical register. If the previous value has not been produced yet, the direct check of the silent write is not performed. This approach is the most complex since it increases pressure on register read ports and may require changes to the select logic for read port arbitration.
- ORT_dis: Slipstream's Operand Rename Table (ORT) is an existing component that detects dead writes and silent writes, an essential part of learning what to remove from the partial thread in the future [5]. It resembles an architectural register file, namely it is indexed by logical register and values are committed to it as instructions retire. These values facilitate detection of silent writes: a silent write is detected when its value matches the corresponding one in the ORT. The ORT can be leveraged for direct checks of silent writes, before or at retirement. In the case of ORT_dis, the ORT is queried at dispatch. When a predicted silent write dispatches, it reads the corresponding value from the ORT. The immediately previous write may not have retired yet in which case the ORT does not have the value we need to compare with. This can be determined by adding a mapping field to the ORT which indicates the committed mapping. If the previous mapping obtained at dispatch matches the ORT committed mapping, the ORT value is the immediately previous value we want to compare with. Otherwise the direct check of the silent write is not performed.
- ORT_exe: This approach is the same as ORT_dis except the ORT is queried when the predicted silent write executes. Waiting until execution increases the chance that the previous value is in the ORT.
- ORT_ret: The ORT is queried for the previous value when the predicted silent write retires. In fact, this query is already done by the IR-detector in the course of learning about past silent writes, but previously this query was not exploited to perform direct checks of predicted silent writes (and possible recovery) before retirement. ORT_ret essentially comes for free in the baseline slipstream implementation. While this direct check detects and recovers from faults in all correctly predicted silent writes, without a history buffer recovery model, the direct check is too late to recover from faults originating in the backward slices of the silent writes. Fortunately, results in Section 6 show that good coverage is primarily needed for the silent writes themselves.
- ORT_all: This is a combination of ORT_dis, ORT_exe, and ORT_ret: the ORT is queried at all stages until the previous value becomes available in the ORT. The previous value is guaranteed to be available by the time the predicted silent write retires (ORT_ret).

3.3.2 Method for Directly Checking Silent Stores

A predicted silent store is converted to a load in the load/store pipeline to obtain the previous value at the store address [7]. If it is truly silent, there is no net increase in cache bandwidth since the converted load replaces the store.

Assuming the silent store prediction is correct (assumption for coverage, as explained in Section 2), the direct check can detect a fault in either the silent store address or value. If the value is flawed, it will differ from the value in memory and be detected as a silent store misprediction. If the address is flawed, and the wrong location contains a different value than the store value, the flawed address will be detected as a silent store misprediction. If the value in the wrong location is the same as the store value, then the silent store is silent with respect to both the original and wrong locations, therefore, the flawed address is masked.

4. Novel Coverage Analysis

Instead of literally injecting faults in the simulator, we view each and every instruction as potentially faulty. Accordingly, we say that each instruction is a "candidate faulty instruction". Each candidate faulty instruction is scrutinized to determine whether or not it is directly or indirectly checked before it retires. In slipstream, duplicated instructions and some non-duplicated instructions (explained below) are directly checked via pairwise comparisons whereas all other non-duplicated instructions are indirectly checked, as discussed in Section 2.3. If the analysis framework determines that a candidate faulty instruction is checked either directly or indirectly before retirement, the instruction is included in coverage since recovery would succeed in this case (the faulty instruction would be prevented from retiring, or the faulty instruction is architecturally dead [9]).

Analysis is straightforward for duplicated instructions and directly-checked non-duplicated instructions. Directly-checked non-duplicated instructions include (1) confident branches and (2) confident silent writes and stores, only if direct checks of silent writes and stores are employed. We call these instructions "checkers" since they check themselves. Checkers are included in coverage if there are no coincident mispredictions (mispredictions cause losses in coverage, as discussed in Section 2.3.5).

The complexity of the analysis framework stems from non-duplicated instructions that are not directly checked. We call these instructions "non-checkers" since they cannot check themselves. Our analysis technique examines a forward slice of each non-checker instruction in the full thread, sufficient for safely (conservatively) determining whether or not the instruction is

checked by checker descendants before retirement or does not need to be checked (architecturally dead). Note that this is only a measurement technique in the simulator, not a hardware mechanism (although it could be a useful mechanism for explicitly deferring retirement of non-checker instructions for higher coverage).

Because of the possibility of masking consumers, using only the first completed checker descendant is not a safe indicator that its non-checker ancestor is checked before retirement. A masking consumer is one which cannot detect a faulty producer because it produces a correct output despite an incorrect input. While it may seem like the faulty producer does not matter, it depends on whether or not there are additional, non-masking consumers. Ideally, the analysis should identify the first completed non-masking checker descendant, since it truly checks its non-checker ancestor. However, determining whether or not an instruction is non-masking is sometimes difficult because it depends on specific values and fault locations interacting with the operation type. Moreover, we prefer a conservative value-agnostic measure of coverage, as a bound.

Therefore, instead of explicitly searching for the first completed non-masking checker descendant, the analysis considers all direct/indirect checker descendants under the assumption that one or more are unknowingly masking. More formally, a finite forward slice is identified, whose leaf instructions consist of only checker instructions: (1) duplicated instructions, (2) confident branches, (3) confident dead writes, and/or (4) confident silent writes or stores, only if direct checks of these are employed.

An example forward slice of a non-checker instruction, A, is shown in Figure 8. The terminal instructions (leaf instructions) of the slice include three duplicated instructions D, F, and H. In addition, there are two other terminal instructions, a confident branch E and confident dead write G. There is a very simple criterion for knowing when a forward slice is fully formed. It is fully formed when there are no "live" non-checker instructions in the slice. If there are no live non-checker instructions, there are no possible places in the slice where additional terminal instructions (checker descendants) can be added. In the example, non-checker instructions A, B, and C have all been killed and hence the forward slice of A is fully formed (note that G must also be killed, confirming its checker status as a confident dead write). In addition, the forward slices of non-checker instructions B and C are fully formed, too.

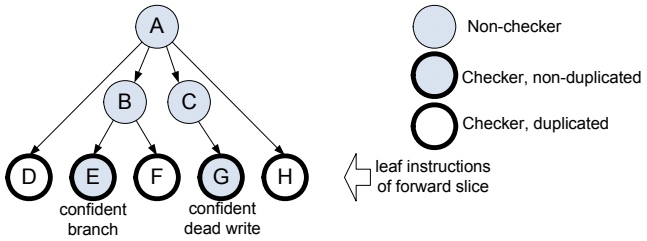


Figure 8. Forward slice example.

If the following three criteria are met before a non-checker instruction is retired, then the non-checker instruction is covered, assuming ROB-flush or History Buffer recovery:

1. Its forward slice is fully formed, i.e., there are no live non-checker instructions remaining in the ongoing slice, since these represent possible places to add other checkers.
2. All predictions in the slice are correct (mispredictions reduce coverage, as explained in Section 2, and this is accounted for in overall coverage).
3. All terminal instructions of the slice have completed execution, not only checking themselves but also indirectly checking all non-checker ancestors in the slice.

If none of the terminal instructions of a fully-formed misprediction-free slice detect a fault, it confirms that either (i) the non-checker ancestors are fault-free or (ii) the non-checker ancestors may be faulty but all of their checker descendants are masking instructions, so they can be safely retired despite being faulty.

5. Simulation Environment

Our detailed cycle-level simulator models three different execution modes on a common simultaneous multithreading (SMT) microarchitecture, shown in Table 1.

1. Single: Only a single thread.
2. Slip: Slipstream execution using a partial leading thread and a full trailing thread.
3. Slip-Full: Slipstream execution using a full leading thread and a full trailing thread (100% coverage). This models an optimized full duplication approach [19][16], since the trailing thread is "accelerated" by outcomes from the leading thread. Full duplication is achieved by turning off instruction removal from the leading thread.

Table 1. SMT microarchitecture for all three modes.

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): 256 load/store queue: 64 issue queue: 64 dispatch/issue/retire bandwidth: 8 cache ports: 4 read/write

Our slipstream implementation mirrors the microarchitecture described in previous work and we also use the same parameters for slipstream components and A-stream/R-stream memory management [5][14].

We run SPEC2K integer benchmarks compiled with the Simplescalar gcc compiler [2] for the PISA ISA. The compiler optimization level is –O3. Reference inputs are used. In our runs, we skip 1 billion instructions and simulate 100 million instructions. Only 10 of 12 integer benchmarks are run because eon and crafty do not compile.

6. Results

6.1 Instruction Breakdown

To understand coverages of different slipstream variants in subsequent sections, it is useful to refer to the breakdown of retired instructions in Figure 9. Three types of checkers are shown

at the bottom of each bar: (1) Dup – duplicated instructions, (2) B – confident branches, (3) D – confident dead writes. The next two types may be checkers or non-checkers, depending on whether or not direct checks are employed: (4) SS – confident silent stores, (5) SW – confident silent writes. The next four types are non-checkers: (6) bs_B – in backward slice of confident branch, (7) bs_SS – in backward slice of confident silent store, (8) bs_SW – in backward slice of confident silent write, (9) Other – this includes smaller components such as bs_D (backward slice of confident dead write) and instructions in backward slices of multiple types. We will refer to this breakdown chart when explaining coverages.

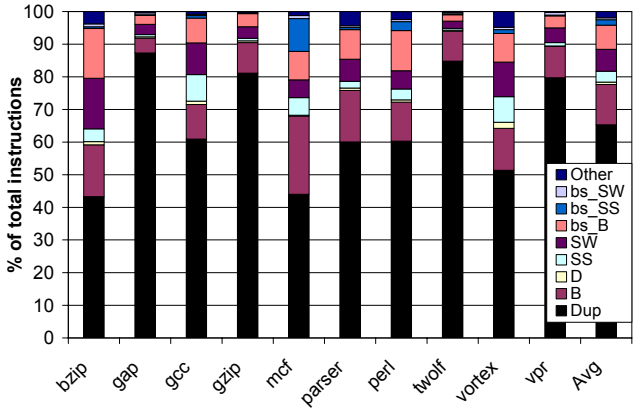


Figure 9. Breakdown of retired instructions.

6.2 Slipstream with Base Recovery

In this section we present coverages for various slipstream implementations that all use the base recovery model. The base recovery model restarts from the checker that detects the fault, whether or not the original faulty instruction is in the backward slice of the checker. Thus, in all cases, coverage is limited to only checker instructions (those that can check themselves): Dup, B, D, and possibly SS, SW.

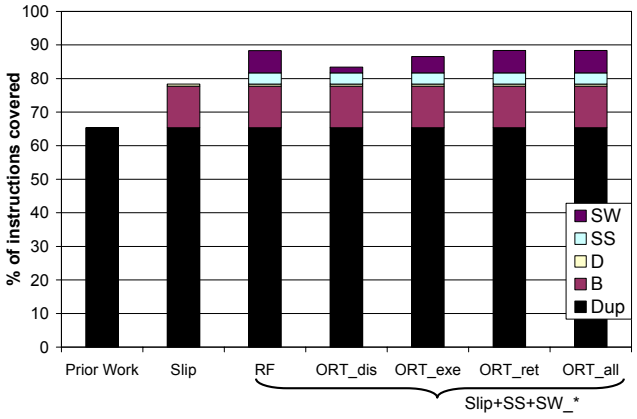


Figure 10. Coverages of slipstream with base recovery.

Coverages are shown in Figure 10. The first bar (PriorWork) shows the coverage of baseline slipstream as predicted by prior work [23][13]. The coverage only includes Dup, which is an underestimate. The second bar (Slip) shows the true coverage of baseline slipstream, corresponding to all checker instructions (Dup, B, D). Coverage of Slip is 78% on average, whereas PriorWork puts coverage at only 65% on average.

The next five bars show slipstream augmented with direct checks of both silent stores and silent writes (Slip+SS+SW_*). There are five bars corresponding to the five methods for direct checking of silent writes, using notation from Section 3.3.1.

With Slip+SS+SW_*, coverage includes a varying percentage of SW instructions depending on when the direct checks of SW instructions are performed. As expected, querying the ORT at dispatch (ORT_dis) yields the least coverage of SW instructions since the ORT often does not yet contain the previous value for comparison.

Querying the ORT at retirement (ORT_ret) yields total coverage of SW instructions since the previous value is always available for comparison at that time. The downside of ORT_ret is that, unless history buffer recovery is used, no bs_SW instructions are covered despite total coverage of SW instructions. This downside is not apparent in this section due to the inadequate base recovery model (no backward slice instructions are covered at all). Referring to Figure 9, notice that bs_SW constitutes less than 1% of instructions whereas SW itself constitutes 7% of instructions, on average. We conclude that any increase in bs_SW coverage that may be afforded by earlier direct checks of SW (e.g., ORT_dis or ORT_exe) is not worth the greater loss in SW coverage. While RF achieves the same coverage of SW instructions as ORT_ret, RF is more complex to implement. Summing up, ORT_ret is the best choice in that it yields the highest incremental coverage and it is very cheap to implement in an existing slipstream implementation.

Slip+SS+SW_ORT_ret yields the highest coverage with base recovery: 88%.

6.3 Slipstream with New Recovery

In this section, we explore the coverage benefits of new slipstream recovery approaches. All coverages in Figure 11 are for Slip+SS+SW_ORT_ret, varying only the recovery model. The first bar reiterates coverage of base recovery (base). The next three bars show coverages with ROB-head (RH) recovery, i.e., when a checker instruction detects a fault, the processor restarts both threads from the instruction at the head of the ROB. The number after RH indicates whether or not our second technique, ROB occupancy management, is used and what the occupancy threshold is. RH0 means there is no threshold hence no occupancy management. RH32 and RH48 correspond to ROB-head recovery with ROB occupancy targets of 32 and 48 instructions by the full thread, respectively.

RH0 increases coverage to 95%, up from 88% with base recovery. This is due to partial coverage of non-checkers, for the first time. RH32 and RH48 increase coverage of non-checkers even further. By deferring retirement of non-checkers, RH32/RH48 increase the chances that non-checkers are indirectly checked before retirement. Coverages reach 97% and 98% for RH32 and RH48, respectively. However, a potential drawback of ROB occupancy management is performance degradation, which we explore further in Section 6.5.

The history buffer (HB) approach provides a performance-friendly alternative to ROB occupancy management. The final two bars in Figure 11 show history buffers of 16 instructions

(HB16) and 32 instructions (HB32). Coverage is 98% and 99% for HB16 and HB32, respectively.

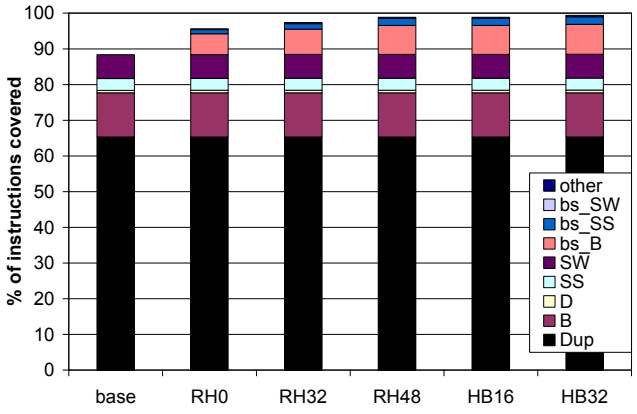


Figure 11. Coverage of Slip+SS+SW_ORT_ret for various recovery approaches.

6.4 No Direct Checks of Silent Writes

We introduced direct checks of silent writes and silent stores to guarantee coverage of these instructions. The rationale is that indirect checks of silent writes and silent stores, via their forward slices, are not guaranteed to be timely. However, this rationale is pessimistic for silent writes, whose complete forward slices are likely to be in the window, and effective. Therefore, in this section, we consider eliminating direct checks of silent writes in favor of indirect checks.

While base recovery cannot cover silent writes without direct checks, the new recovery schemes should be able to cover many silent writes indirectly.

Figure 12 shows coverages for slipstream with direct checks of silent stores but no direct checks of silent writes (Slip+SS). With base recovery, Slip+SS achieves 81% coverage, down from 88% with Slip+SS+SW. With the new recovery schemes, there is a gradual increase in coverage from 90% for RH0 to 98% for HB32. We conclude that most silent writes can be covered by indirect checks, with good recovery.

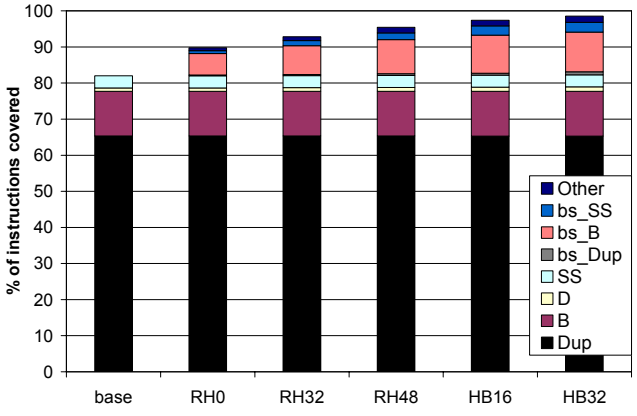


Figure 12. Coverage of Slip+SS for various recovery approaches.

Notice that a new category, bs_Dup, appears in Figure 12. In the slipstream paradigm, of all the types of predicted-ineffectual (i.e., non-duplicated) instructions – branches, dead writes, silent writes, silent stores, and their backward slices – only silent writes and silent stores can possibly lead to predicted-effectual (i.e., duplicated) instructions. Since silent writes no longer check themselves, the bs_Dup category signifies that some silent writes are now indirectly checked by duplicated instructions.

6.5 Performance

Figure 13 shows the performance of (1) single-thread execution (Single), (2) Slipstream with full duplication (Slip-Full), (3) Slipstream with partial duplication and base recovery (Slip:base), and (4) Slipstream with partial duplication, using Slip+SS+SW_ORT_ret and various recovery models (Slip+:*). Single and Slip-Full show the two extremes between full performance and full fault tolerance. Slip-Full achieves 100% fault tolerance, but at the price of 14% average slowdown. Among benchmarks with IPCs above 2.5 (bzip, gap, gcc, perl, and vortex), where the processor is utilized moderately well by a single thread, the average slowdown of Slip-Full is 18.5%.

We use the first three bars (Single, Slip-Full, Slip:base) to categorize benchmarks into three groups. Mcf, twolf, and vpr show little difference in performance among different threading scenarios. These benchmarks utilize the processor poorly with a single thread, hence adding another thread has little impact. Gap is in its own category. Gap utilizes the processor quite well with a single thread and both Slip-Full and Slip:base cause a significant slowdown. Slip:base underperforms because gap has little instruction removal as can be seen in Figure 9: 87% of instructions are duplicated. Finally, bzip, gcc, gzip, parser, perl, and vortex show mild to significant slowdown with Slip-Full, yet Slip:base performs comparably to Single. In some cases Slip:base outperforms Single, which is not unexpected since slipstream is known to enhance performance on both CMP and SMT substrates [13].

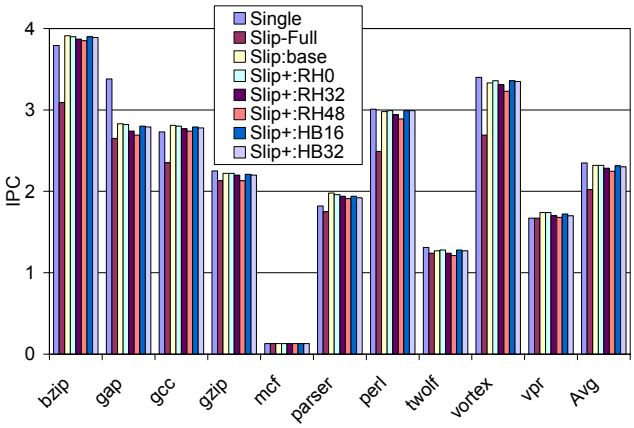


Figure 13. Performance comparisons.

On average, Slip:base is only 1.3% slower than Single, whereas Slip-Full is 14% slower. Among 2.5+ IPC benchmarks, Slip:base is only 2.7% slower, whereas Slip-Full is 18.5% slower.

Next we look at Slip+ with various recovery models. All benchmarks show the expected behavior. First, there is a gradual decrease in performance from base through RH0, RH32, RH48. RH0 flushes more instructions than base, and RH32/RH48 defer

retirement. Second, the history buffer approaches HB16 and HB32 climb back to the performance of base because retirement is not deferred. Perl, twolf, and vortex show slightly more performance with Slip+:* compared to Slip:base. This is due to slight timing perturbations caused by converting silent stores to loads in Slip+.

6.6 Using Only Branch Predictions

In this section, we apply our forward-slice analysis framework to estimate the dynamic instruction coverage of a ReStore-like architecture. In particular, there is only a single thread and confident branch predictions are used to check corresponding branches and their backward slices, as discussed in Section 2.2.

Figure 14 shows the coverage and breakdown of covered instructions. All correctly predicted branches are covered because a correct prediction will differ from a wrong outcome. Thus, both confident (B) and unconfident (b) correctly predicted branches are included in coverage. However, only mispredictions among confidently predicted branches cause a rollback to an earlier point in the program. Therefore, our forward-slice analysis only considers confident correctly predicted branches (B) as checkers, i.e., terminals of a forward slice. Accordingly, an arbitary instruction is not considered checked until its forward slice has only confident correctly predicted branches (B) and/or dead writes (D) for leaves. In Figure 14, arbitrary instructions that are successfully checked in this way constitute the bs_B category (in backward slice of B). The number of bs_B instructions increases with larger rollback distances, as shown for various recovery models (RH0, HB32, HB64, HB128). On average, Single+B (single thread with B as checkers) yields 33% coverage with HB128 recovery (D, B, b, and bs_B are covered).

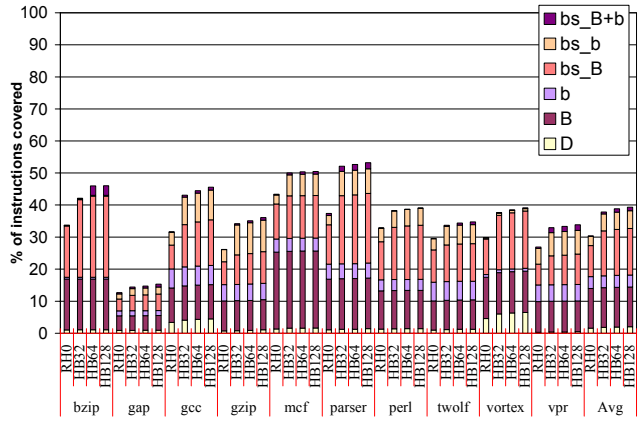


Figure 14. Coverage for Single+B(+b).

Coverage can be maximized by rolling back for all branch mispredictions. This means both confident (B) and unconfident (b) correctly predicted branches are checkers (that is, all correctly predicted branches). This covers additional instructions, bs_b (in backward slice of b) and bs_B+b (in backward slice of B and b) in Figure 14. On average, coverage for Single+B+b is 40% with HB128 recovery.

A more optimistic analysis assumes no fault masking by B (or b) instructions, meaning that an arbitrary instruction is considered checked by the first B (or b) instruction encountered in its forward slice. Corresponding optimistic coverages are shown in Figure 15. (There is no bs_B+b category since the first B or b instruction is used to check.) On average, optimistic coverages for Single+B and Single+B+b are 50% and 60%, respectively, with HB128 recovery.

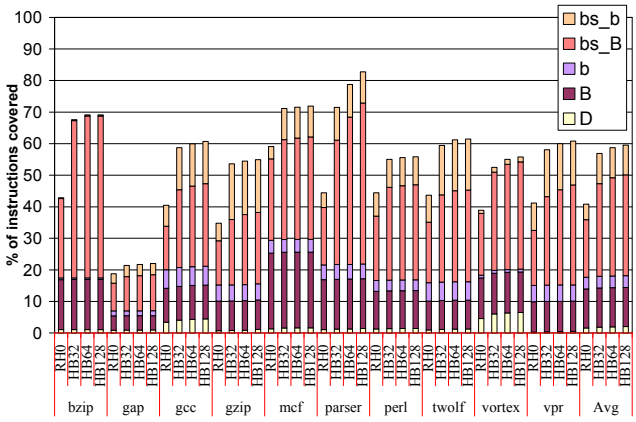


Figure 15. Optimistic coverage for Single+B(+b).

Performance of Single+B and Single+B+b is shown in Figure 16. The performance of Single+B+b is not shown with a dedicated bar, rather, it is shown with a negative error bar with respect to the Single+B bar. Peformance of Single (no coverage) and Slip-Full (full coverage) are shown for comparison.

The performance degradation of Single+B, with respect to Single, is mild. Rollbacks are rare because only mispredictions among confidently predicted branches cause rollbacks. As expected, the performance degradation increases slightly with more distant rollbacks (from RH0 to HB128).

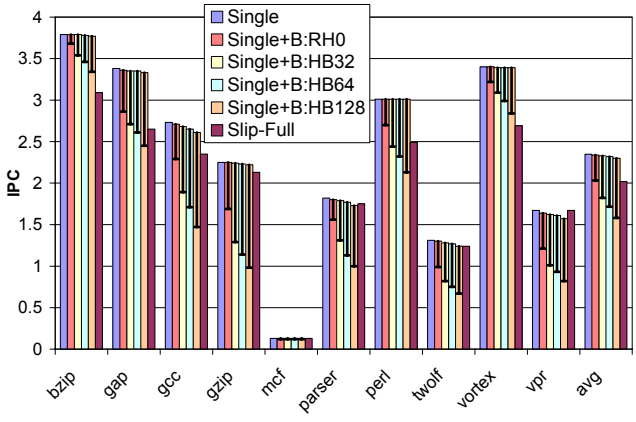


Figure 16. Performance of Single+B(+b).

The performance degradation of Single+B+b (negative error bar) is severe, because all mispredictions cause rollbacks. For the same reason, performance is very sensitive to the rollback distance. Nonetheless, for some benchmarks and rollback distances, Single+B+b outperforms Slip-Full (albeit with less coverage).

7. Related Work

There has been significant interest in redundant multithreading architectures in recent years. These architectures exploit simultaneous multithreading [16][19][23][25], chip multiprocessors [3][8][23], or modified superscalar hardware [1][15][22]. A universal goal has been maximizing both coverage

and performance. Thus, we focus discussion of related work on optimizations towards this goal, including (1) reducing resource pressure, (2) reducing checking bandwidth, and (3) reducing instruction count.

Several techniques have been developed to reduce resource contention. First, resource pressure can be reduced by staggering the two threads, such that one thread runs slightly ahead of the other [19][16][22][23]. The leading thread passes its outcomes to the trailing thread for checking. The leader/follower arrangement enables a key performance optimization: the leading thread's outcomes can be leveraged as likely-correct (they are correct in the fault-free case) branch and value predictions in the trailing thread [19]. The trailing thread executes more efficiently because all of its control and data dependences are eliminated (no wrong-path instructions and perfect value prediction in fault-free case). The effect is that the trailing thread requires fewer resources than the leading thread for the same performance, thus releasing resources back to the leading thread, reducing overall execution time for dual-redundant execution.

Recent work proposes other per-structure optimizations for reducing resource pressure [6], such as packing dual instances of a dynamic instruction into the same physical register for short-width values (exploiting advance knowledge from the leading thread) or intelligently reallocating some leading thread's physical registers to the trailing thread to yield a net reduction in physical register pressure.

In another direction, several techniques have been proposed for reducing the number of checks (comparisons). All instructions are still executed twice, so that whichever two instructions are compared, they are still based on separate computation. One approach can detect all single transient faults by checking only store instructions [16]. Dependence Based Checking Elision [25][3] reduces the number of checks based on the idea that a fault propagates through dependent instructions, so checking an instruction in a chain implicitly checks instructions leading to it. Prediction-based PRT approaches exploit the same principle to recover from faults on singly executed instructions. In this case there is no choice but to check only the consumer whereas in DBCE all instructions are redundantly executed with the option of not checking all of them. The key difference is that prediction-based PRT fully capitalizes on the notion of consumer-based checking, by not only eliminating the check of the producer, but the producer instruction itself. This reduces pressure not only on the checking machinery, but the processor as a whole.

A number of varied approaches reduce the number of redundantly executed instructions. Two of these, Slipstream [5][12][13][14][20][23] and ReStore [26], employ forms of predictive checking. As mentioned earlier, past characterization of slipstream fault tolerance is not extensive and yields a coverage bound limited to only redundantly executed instructions. This paper contributes new analysis that reveals theoretical coverage of singly executed instructions, and recovery techniques to achieve the coverage. ReStore [26] is closely related in two respects. First, fault detection is achieved purely by symptoms such as exceptions, cache misses, TLB misses, and branch mispredictions, a form of predictive checking since no redundant execution is used at all (although frequent and distant rollbacks are in some sense redundant execution after the fact). Second, when such symptoms are detected, the processor rolls back to a prior distant checkpoint, in the hope that the faulty instruction has not retired. This is similar in spirit to recovery of singly executed instructions in this paper. ReStore does not exploit redundant execution *a priori* and thus is one extreme of the predictive checking spectrum. The "partial thread" has no computation at all and this has performance implications (frequent rollbacks if any misprediction is a symptom) and/or coverage drawbacks (using only confident branch mispredictions as symptoms limits coverage to their slices only). We believe the predictive checking spectrum has more design points worth exploring.

Opportunistic fault tolerance [4] also reduces the number of redundantly executed instructions by initiating redundancy only during phases of otherwise poor performance. Singly executed instructions are not covered. Hybrid compiler/hardware approaches [17][18] provide a level of control over performance and coverage not feasible in purely hardware threading approaches, such as complex analytical frameworks for identifying code regions where performance and coverage are not conflicting. Finally, instruction reuse can be used to reduce the number of redundant executions [10].

A related MS thesis laid the initial groundwork for this paper [11]. It describes a hardware implementation of the forward slice checking analysis. The hardware mechanism explicitly classifies instructions as checked or not checked at retirement, and stalls commit, accordingly. This paper shows that check status does not need to be literally tracked in hardware, and the proposed recovery models achieve similar high coverage without explicit tracking. This paper also contributes an in-depth study of prediction-based partial redundant threading.

8. Summary

Prediction-based checking is a promising new direction in efficient fault tolerance. In this paper, we showed for the first time that the combination of confident predictions and partial duplication can approximate the fault tolerance of full duplication. Slipstream is a convenient substrate for testing this hypothesis, as it embodies the notion of comparing a full thread with an independent, reduced, predictive thread. We performed a thorough dissection of four prediction scenarios, revealing near-100% fault detection capability despite duplicating as few as 43% of instructions. We then revamped slipstream's recovery and checking implementation with a suite of strategies that now make it possible to nearly fully capitalize on the excellent fault detection capability. Our initial foray into prediction-based partial redundant threading, yielded instruction coverages of 99% with performance close to a single thread.

We plan to extend the analysis framework and develop corresponding microarchitecture techniques to encompass more sophisticated error models. The analyzed scenarios assume faults surface directly as erroneous instruction outcomes and the "structure" of the dynamic instruction stream is unaffected. However, faults may cause inter-instruction dependences to change, potentially introducing scenarios outside the scope of our analysis framework. An extended analysis framework may guide the development of more robust microarchitecture techniques.

Longer term, we would like to understand how far we can push the prediction side of prediction-based PRT to achieve extremely low-overhead and high-coverage fault-tolerant architectures. The

principles and analysis framework developed in this paper may reveal additional steps towards this grand challenge.

9. 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.

10. References

[1] T. M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. *32nd International Symposium on Microarchitecture*, pp. 196-207, Nov. 1999.

[2] D. Burger, T. M. Austin, and S. Bennett. The Simplescalar Toolset, Version 2. Tech. Report CS-TR-1997-1342, CS Department, University of Wisconsin-Madison, July 1997.

[3] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. *30th International Symposium on Computer architecture*, pp. 98-109, June 2003.

[4] M. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. *32nd International Symposium on Computer Architecture*, pp. 172-183, June 2005.

[5] J. J. Koppanalil and E. Rotenberg. A simple mechanism for detecting ineffectual instructions in slipstream processors. *IEEE Trans. on Computers*, 53(4):399-413, April 2004.

[6] S. Kumar and A. Aggarwal. Reducing resource redundancy for concurrent error detection techniques in high performance microprocessors. *12th International Symposium on High-Performance Computer Architecture*, pp. 212-221, Feb. 2006.

[7] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. *27th International Symposium on Computer Architecture*, pp. 182-191, June 2000.

[8] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. *29th International Symposium on Computer Architecture*, pp. 99-110, May 2002.

[9] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. *36th International Symposium on Microarchitecture*, pp. 29-40, Dec. 2003.

[10] A. Parashar, S. Gurumurthi and A. Sivasubramaniam. A complexity-effective approach to ALU bandwidth enhancement for instruction-level temporal redundancy. *31st International Symposium on Computer Architecture*, pp. 376-386, June 2004.

[11] S. Parthasarathy. Improving transient fault tolerance of slipstream processors. M.S. Thesis, ECE Department, North Carolina State University, Dec. 2005.

[12] Z. R. Purser. Slipstream processors. Ph.D. Thesis, ECE Department, North Carolina State University, July 2003.

[13] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. *33rd International Symposium on Microarchitecture*, pp. 269-280, Dec. 2000.

[14] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. Slipstream memory hierarchies. Tech. Report CESR-TR-02-3, ECE Department, North Carolina State University, Feb. 2002.

[15] J. Ray, J. C. Hoe and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. *34th International Symposium on Microarchitecture*, pp. 214-224, Dec. 2001.

[16] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. *27th International Symposium on Computer architecture*, pp. 25-36, June 2000.

[17] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan and D. I. August. SWIFT: Software implemented fault tolerance. *3rd International Symposium on Code Generation and Optimization*, pp. 243-254, March 2005.

[18] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Design and Evaluation of Hybrid Fault-Detection Systems. *32nd International Symposium on Computer Architecture*, pp. 148-159, June 2005.

[19] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. *29th International Symposium on Fault-Tolerant Computing*, pp. 84-91, June 1999.

[20] E. Rotenberg. Exploiting large ineffectual instruction sequences. Technical Report, North Carolina State University, Nov. 1999.

[21] J. E. Smith and A. R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. *12th International Symposium on Computer Architecture*, pp. 36-44, June 1985.

[22] J. C. Smolens, J. Kim, J. C. Hoe and B. Falsafi. Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures. *37th International Symposium on Microarchitecture*, pp. 257-268, Dec. 2004.

[23] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: improving both performance and fault tolerance. *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 257-268, Nov. 2000.

[24] D. Tullsen, S. J. Eggers and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *22nd International Symposium on Computer Architecture*, pp. 392-403, June 1995.

[25] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. *29th International Symposium on Computer Architecture*, pp. 87-98, May 2002.

[26] N. J. Wang and S. J. Patel. ReStore: Symptom based soft error detection in microprocessors. *International Conference on Dependable Systems and Networks*, pp. 30-39, June 2005.