# Slipstream Processors:
# Improving both Performance and Fault Tolerance

Karthik Sundaramoorthy Zach Purser Eric Rotenberg

North Carolina State University

Department of Electrical and Computer Engineering

Engineering Graduate Research Center, Campus Box 7914, Raleigh, NC 27695

{ksundar,zrpurser,ericro}@ece.ncsu.edu, www.tinker.ncsu.edu/ericro/slipstream

## ABSTRACT

Processors execute the full dynamic instruction stream to arrive at the final output of a program, yet there exist shorter instruction streams that produce the same overall effect. We propose creating a shorter but otherwise equivalent version of the original program by removing ineffectual computation and computation related to highly-predictable control flow. The shortened program is run concurrently with the full program on a chip multiprocessor or simultaneous multithreaded processor, with two key advantages:

1) *Improved single-program performance*. The shorter program speculatively runs ahead of the full program and supplies the full program with control and data flow outcomes. The full program executes efficiently due to the communicated outcomes, at the same time validating the speculative, shorter program. The two programs combined run faster than the original program alone. Detailed simulations of an example implementation show an average improvement of 7% for the SPEC95 integer benchmarks.

2) *Fault tolerance*. The shorter program is a subset of the full program and this partial-redundancy is transparently leveraged for detecting and recovering from transient hardware faults.

## 1. INTRODUCTION

A conventional processor executes the full dynamic instruction stream to arrive at the final output of the program. The *slipstream paradigm* proposes that only a subset of the original dynamic instruction stream is needed to make full, correct, forward progress [25]. Unfortunately, we cannot know for certain what dynamic instructions can be validly skipped. Creating a shorter, equivalent program is speculative — ultimately, it must be checked against the full program to verify it produces the same overall effect.

Therefore, the operating system creates two redundant processes, i.e., the user program is instantiated twice and each instance has its own context. The two redundant programs execute simultaneously on a single-chip multiprocessor (CMP) [20] or on a simultaneous multithreaded processor (SMT) [37]. One of the programs always runs slightly ahead of the other. The leading program is called the *advanced stream*, or A-stream, and the trailing program is called the *redundant stream*, or R-stream. Hardware monitors the trailing R-stream and detects 1) dynamic instructions that repeatedly and predictably have no observable effect (e.g., unreferenced writes, non-modifying writes) and 2) dynamic branches whose outcomes are consistently predicted correctly. Future dynamic instances of the ineffectual instructions, branch instructions, and the computation chains leading up to them are speculatively bypassed in the leading A-stream — but only if there is high confidence correct forward progress can still be made, in spite of bypassing the instructions.

The much-reduced A-stream is sped up because it fetches, executes, and retires fewer instructions than it would otherwise. Also, all values and branch outcomes produced in the leading A-stream are communicated to the trailing R-stream. Although the R-stream is not reduced in terms of retired instructions, it has an accurate picture of the future and fetches/executes more efficiently. In summary, the A-stream is sped up because it is shorter and the R-stream is sped up because it receives accurate predictions from the A-stream. *The two redundant programs combined run faster than either can alone*.

The A-stream's outcomes are used only as *predictions* to speed up the R-stream. But ultimately, the same information is redundantly and independently computed by the R-stream. This is crucial because the A-stream occasionally (but infrequently) bypasses computation that should not have been bypassed, and it no longer makes correct forward progress. The R-stream can detect deviations because its redundantly-computed outcomes differ from the A-stream's outcomes. And the checks are already in place if the existing design implements conventional branch and value prediction [24]. When the A-stream deviates, the architectural state of the R-stream is used to selectively recover the corrupted architectural state of the A-stream.

An analogy to the slipstream paradigm (and the source of its name) is "slipstreaming" in stock-car racing (e.g., NASCAR) [23]. At speeds in excess of 190 m.p.h., high air pressure forms at the front of a race car and a partial vacuum forms behind it. This creates drag and limits the car's top speed. A second car can position itself close behind the first (a process called *slipstreaming* or *drafting*). This fills the vacuum behind the lead car, reducing its drag. And the trailing car now has less wind resistance in front (and by some accounts, the vacuum behind the lead car actually helps pull the trailing car). As a result, both cars speed up by several m.p.h.: the two combined go faster than either can alone.

Similarly, the A-stream and R-stream mutually improve one another's performance. The A-stream could not be accurately reduced without the trailing R-stream. And the R-stream is helped along in the slipstream (control and data flow outcomes) of the A-stream. The user perceives an overall speedup because both programs finish earlier (the R-stream finishes just after the A-stream, so the R-stream determines when the user's program is done). The amount of performance improvement depends on the nature and amount of reduction in the A-stream. Slipstreaming also relies on proper resource allocation between the two programs (e.g., dedicated processors in a CMP or careful fetch/issue scheduling in an SMT processor).

In addition to potential performance improvements, slipstreaming provides fault-tolerant capabilities. The trends of very high clock speeds and very small transistors may make the entire chip prone to transient faults [29], and there is renewed interest in fault-tolerant architectures for *commodity, high-performance* microprocessors [24,2,22].

Slipstream processors provide substantial but incomplete fault coverage, specifically, faults that affect redundantly-executed instructions are detectable and recoverable. Not all instructions are redundantly-executed because the A-stream is a subset of the R-stream, and this opens up opportunities for dynamically and flexibly trading performance and fault coverage. A transient fault, whether it affects the A-stream, the R-stream, or both streams, is transparently detected as a "misprediction" by the R-stream because the communicated control and data flow outcomes from the A-stream will differ from the corresponding outcomes in the R-stream. Fault detection/recovery is transparent because transient faults are indistinguishable from prediction-induced deviations.

In summary, this paper makes the following contributions.

- We suggest speculatively creating a shorter but otherwise equivalent version of the program, exploiting 1) computation that repeatedly and predictably has no effect on the final program output and 2) computation that influences highly-predictable branches.

- The shortened program is run in parallel with the full program on a single-chip multiprocessor or simultaneous multithreaded processor and, by communicating information from the shortened program to the full program, single-program execution time is potentially improved and substantial transient fault coverage is achieved.

- This work is part of a larger effort using multiple on-chip, architectural contexts in new ways. CMP/SMT processors are strategic because they effectively utilize billion-transistor chips with *relative* ease, integrating parallelism that already exists at the system-level onto a single die. Our goal is threefold: 1) provide more functionality in the same CMP/SMT processor — not just throughput-oriented parallelism, but also fault tolerance and improved single-program performance (for example), 2) provide the new functions in a non-intrusive way, by placing hardware "around" the existing components and leveraging, as much as possible, the existing design, and 3) enable the user/operating system to flexibly and dynamically choose from among several modes of operation, e.g., throughput mode, single-program-speedup mode, or reliable mode.

## 2. SLIPSTREAM MICROARCHITECTURE

A slipstream processor requires two architectural contexts, one for each of the A-stream and R-stream, and new hardware for directing instruction-removal in the A-stream and communicating state between the threads. A high-level block diagram of a slipstream processor implemented on top of a two-way chip multiprocessor is shown in Figure 1 (an SMT processor could be used instead). The shaded boxes show the original processors comprising the multiprocessor. Each is a conventional superscalar/VLIW processor with a branch predictor, instruction and data caches, and an execution engine — including the register file and either an in-order pipeline or out-of-order pipeline with reorder buffer (we assume the latter in the rest of the paper).

There are four new components to support slipstream processing.

1. The *instruction-removal predictor,* or IR-predictor, is a modified branch predictor. It generates the program counter (PC) of the next block of instructions to be fetched in the A-stream. Unlike a conventional branch predictor, however, *the predicted next PC may reflect skipping past any number of dynamic instructions* that a conventional processor would otherwise fetch and execute. In this paper, the IR-predictor is built on top of a conventional trace predictor [13] but other designs are possible (e.g., using conventional single-branch predictors).

2. The *instruction-removal detector,* or IR-detector, monitors the R-stream and detects instructions that could have been removed from the program. The IR-detector conveys to the IR-predictor that particular instructions should potentially be skipped by the A-stream when they are next encountered. Repeated indications by the IR-detector build up confidence in the IR-predictor, and the predictor will remove future instances from the A-stream.

3. The *delay buffer* is used to communicate control and data flow outcomes from A-stream to R-stream (the R-stream is "delayed" with respect to the A-stream [24]).

4. The *recovery controller* maintains the addresses of memory locations that are potentially corrupted in the A-stream context. A-stream context is corrupted when the IR-predictor removes instructions that should not have been removed. Unique addresses are added to and removed from the recovery controller as stores are processed by the A-stream, the R-stream, and the IR-detector. The current list of memory locations in the recovery controller is sufficient to recover the A-stream memory context from the R-stream's memory context. The register file is repaired by copying all values from the R-stream's register file.

Note that two kinds of speculation occur in the A-stream. Conventional speculation occurs when branches are predicted and the branch-related computation has not been removed from the A-stream. Mispredictions resulting from conventional speculation are detectable by the A-stream, do not corrupt the A-stream context, and do not involve the recovery controller.

The second type of speculation occurs when the IR-predictor removes instruction sequences from the A-stream. The A-stream has no way of detecting whether or not removing the instructions

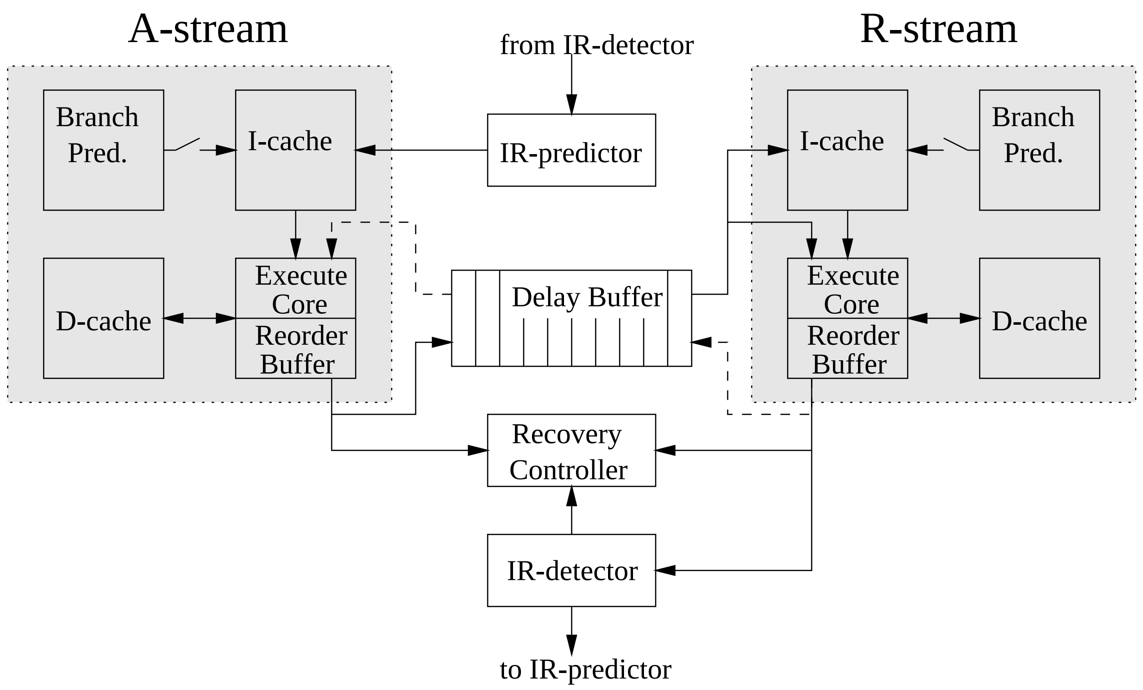

**Figure 1: Slipstream Processor using a two-way chip multiprocessor.**

was correct. Therefore, an incorrect decision by the IR-predictor can result in corrupted A-stream state. In the remainder of the paper, we refer to mispredictions by the IR-predictor as *instruction-removal mispredictions*, or IR-mispredictions, distinguishing this type of misprediction from A-stream-detectable ones.

In Sections 2.1 through 2.3, we describe the above components in more detail and elaborate on Figure 1. Section 2.1 describes how the IR-detector and IR-predictor work to create the shorter program. Section 2.2 describes the delay buffer and its interfaces to the A-stream and R-stream. Section 2.3 explains how IR-mispredictions are detected by either the R-stream or IR-detector, and how the A-stream context is recovered from the R-stream context with relatively low latency.

## 2.1 Creating the Shorter Program

The IR-detector monitors past run-time behavior and detects instructions that could have been removed, and might possibly be removed in the future. This information is conveyed to the IR-predictor, and after sufficient repeated indications by the IR-detector, the IR-predictor removes future instances of the instructions.

### *2.1.1 IR-predictor*

In this paper, the IR-predictor is built on top of a conventional trace predictor [13]. A trace predictor divides the dynamic instruction stream into *traces* — large dynamic instruction sequences (e.g., 16 to 32 instructions) typically containing multiple taken/not-taken branch instructions. The next trace in the dynamic instruction stream is predicted using a path history of past traces.

A conventional trace predictor works as follows [13]. A trace is uniquely identified by a starting PC and branch outcomes indicating the path through the trace, and the combination of start PC plus branch outcomes is called a *trace id*. An index into a correlated prediction table is formed from the sequence of past trace ids, using a hash function that favors bits from more recent trace ids over less recent trace ids. Each entry in the correlated prediction table contains a trace id and a 2-bit counter for replacement. The predictor is augmented with a second table that is indexed with only the most recent trace id. The second table requires a shorter learning time and suffers less from aliasing pressure. Together, the two tables form a hybrid predictor that outputs the predicted trace id of the next trace.

To form an IR-predictor, three pieces of information are added to each table entry.

1. *Instruction-removal bit vector* (*ir-vec*). This bit vector indicates which instructions in the predicted trace to remove from the A-stream. An instruction is removed if its corresponding bit in the *ir-vec* is set.

2. *Intermediate program counter values*. To fetch a trace from a conventional instruction cache, the trace is decomposed into multiple sequential fetch blocks (fetch blocks are separated by taken branches). A trace id only specifies the PC of the first fetch block and a series of embedded branch predictions. PCs of embedded fetch blocks are not available. Conventionally, embedded fetch block PCs are produced using pre-decoded branches in the branch target buffer (BTB) and/or instruction cache. If this approach is used unmodified, the number of dynamic instructions fetched in the A-stream is not reduced. The *ir-vec* itself is only useful for removing instructions after fetch and before decode. To remove A-stream instructions *before they are fetched*, each predictor entry contains intermediate program counter values needed by the instruction fetch unit to skip over chunks of the trace.

3. *Confidence mechanism*. A single resetting confidence counter [12] limits instruction-removal to cases where it is likely to be

correct. The counter is incremented when a newly-generated {*trace-id*, *ir-vec*} pair from the IR-detector matches the old {*trace-id*, *ir-vec*} pair at the entry being updated. If the new pair does not match the old pair, the counter is reset to zero. When the confidence counter has reached a certain threshold, the *ir-vec* and intermediate PCs are used to remove instructions from the A-stream. Otherwise, the instruction-removal information is ignored and all instructions in the trace are fetched/executed.

Figure 2 shows an example of A-stream instruction fetching. A single length-32 dynamic trace is shown. The trace is decomposed into five fetch blocks separated by taken (T) branches. Not-taken (NT) branches are embedded within fetch blocks. The fetch blocks are labeled with PCs *A* through *E*. Each box is an individual instruction, and shaded boxes indicate predicted-removable instructions. In the example, the instruction cache can supply up to 16 sequential instructions per cycle.

- Conventional fetching: The trace predictor stores only the trace id {*A*, NT-T-T-NT-T-T}. PCs *B*, *C*, *D*, and *E* are produced using the BTB/instruction cache and multiple branch predictions. *Five fetch cycles* are expended and a total of 32 instructions are fetched, decoded, and executed.

- Fetching using the IR-predictor: The trace predictor additionally stores an *ir-vec* {0011...} and intermediate PCs *X* and *Y*. Pre-decoded targets from the BTB/instruction cache are ignored. Only *three fetch cycles* are expended: 1 cycle for each of the blocks starting at PC *A*, PC *X*, and PC *Y*. 18 instructions are fetched (block *A*: 8, block *X*: 7, block *Y*: 3). Among these, the *ir-vec* is used to collapse the number of instructions decoded/executed down to 12 instructions.

  Note that a compressed version of the *ir-vec* is stored in place of the full *ir-vec*, in order to properly line up removal bits with corresponding instructions in the new fetch blocks {*A*, *X*, *Y*}.

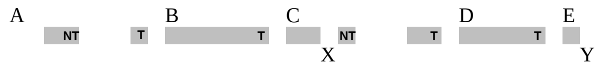

**Figure 2: A-stream instruction fetching example.**

### 2.1.2 IR-detector

There are potentially many ways of speculatively creating a shorter program. Here, we consider two cases of ineffectual computation — writes that are never referenced (dynamic dead code) and writes that do not modify the state of a location [14,15,18,19,38] — as well as branch-predictable computation.

- Some instructions write a value to a register or memory location and the value is overwritten before ever being used. Such instructions, and the computation chains leading up to these instructions, have no effect on final program state.

- Some instructions write the same value into a register or memory location as already exists at that location. Such instructions, and the computation chains leading up to them, have no effect on final program state because their writes were not truly modifications.

- Certain control flow in the program may be so predictable that it *appears* deterministic. With a high level of confidence, we may choose to remove the branches involved, along with the computation chains feeding the branches.

To detect candidate instructions for removal, the R-stream is monitored as it retires instructions. Retired instructions and values are used to 1) construct a small reverse dataflow graph (R-DFG) and 2) detect any of the three triggering conditions for instruction removal, i.e., unreferenced writes, non-modifying writes, and branch instructions. When a triggering condition is observed, the corresponding instruction is selected for removal. Then, the circuits forming the R-DFG back-propagate the selection status to predecessor instructions. A predecessor instruction is also selected for removal if all of its dependent instructions are known and they have been selected for removal. All dependent instructions are known when the consumed value is killed, i.e., when there is another write to the same register/memory location.

The IR-detector is shown in Figure 3. The size of the R-DFG is a single trace (32 instructions in this paper), resulting in practical back-propagation circuitry. Although the scope of back-propagation is limited to a single trace, the IR-detector tracks multiple traces. Maintaining multiple traces at the same time allows a much larger scope for killing values (observing another write to the same location) without increasing the size/complexity of each individual R-DFG.

The operand rename table in Figure 3 is similar to a register renamer but it can track both memory addresses and registers. It performs data dependence checking for merging new instructions into the R-DFG and also detects unreferenced writes and non-modifying writes. Memory entries are invalidated and reclaimed, and register entries simply invalidated, when the last producer of the location is no longer within the analysis scope (the *producer* field facilitates this).

A single entry of the operand rename table is shown in Figure 3, for demonstration. To merge an instruction into its R-DFG, each source operand is checked in the rename table to get the most recent producer of the value (check the *valid bit* and *producer* field). The consumer instruction uses this information to establish connections with its producer instructions, i.e., set up the back-propagation logic. If the producer is not in the same trace, no connection is made. The *ref bit* is set for each source operand indicating the values have been used.

When an instruction writes a register/memory location, the corresponding operand rename table entry is checked to detect non-modifying/unreferenced writes and to kill values, as follows.

1. If the *valid bit* is set, and the current instruction produced the same value as indicated in the *value* field, then the current instruction is a non-modifying write. The current instruction is selected for removal as it is merged into the R-DFG.

2. If the *valid bit* is set and the new and old values do not match, the old producer indicated by the *producer* field is killed. Furthermore, if the *ref bit* is not set, then the old producer is an unreferenced write and is selected for removal.

After these checks are performed, all fields are updated to reflect the new producer instruction unless it is a non-modifying write (the old producer remains “live” in this case).

All branch instructions are selected for removal when they are merged into the R-DFG. This means all branches are candidates for removal, and the confidence counter associated with each trace prediction (Section 2.1.1) makes the actual decision to remove branches.

Finally, any other instruction may be selected for removal if it has been killed, all of its consumer instructions are in the same trace, and all consumers are selected for removal. The R-DFG back-propagation circuitry handles this case.

When a trace becomes the oldest trace in the analysis scope, an instruction-removal bit vector (*ir-vec*) is formed based on the selected instructions within the trace. Intermediate PCs for skipping chunks of the trace are also computed. The trace id, *ir-vec*, and intermediate PCs are loaded into the IR-predictor (Section 2.1.1) and the R-DFG circuitry is reclaimed for a new trace.

### *2.1.3 Discussion of Trace-Based Instruction-Removal*

Instruction-removal in this paper is trace-based, meaning 1) confidence is measured for a trace as a whole (single confidence counter), and 2) back-propagation is confined to a trace. Both conditions guarantee dependence chains are removed as a whole, or not at all. Doing otherwise risks removing a producer instruction but not the corresponding consumer instruction. Even if both producer and consumer are always removable, in practice the IR-predictor could remove one and not the other if separate confidence counters are maintained (e.g., table aliasing can displace the consumer's counter). This scenario results in many spurious IR-mispredictions. It is explicitly avoided by maintaining a single confidence counter per trace and confining back-propagation to a trace.

Trace-based instruction-removal has serious drawbacks, however.

1. Often, there are stable and unstable removal patterns within a trace. The stable patterns correspond to dependence chains that are consistently removable. Unrelated, unstable patterns dilute overall confidence and *no* instructions are removed as a result.

2. Traces themselves are unstable if they embed unpredictable branches. When a trace is unstable, its confidence counter is rarely saturated. Thus, removable instructions before and after the difficult branch are not removed in practice. Terminating traces at difficult branches can reduce the number of unstable traces. But accurate trace prediction relies on a consistent (static) trace selection policy.

3. Confining back-propagation to a trace limits the amount of instruction-removal.

We believe *diluted confidence* and *unstable traces* are largely responsible for modest A-stream reduction in some of our benchmarks (Section 5). We are currently developing a more effective instruction-removal mechanism, *not* available in this paper: 1) it measures confidence for instructions individually, so unrelated instructions do not dilute confidence; 2) traces are not used, so trace stability is not an issue; 3) chains are not confined within a small region, except to reduce R-DFG complexity if needed; 4) dependence chains tend to be removed together even though per-instruction confidence counters are used.

## 2.2 Delay Buffer

The delay buffer is a simple FIFO queue that allows the A-stream to communicate control flow and data flow outcomes to the R-stream. During normal operation, the A-stream pushes both a *complete* history of branch outcomes and a *partial* history of operand values onto the delay buffer. This is shown in Figure 1 with a solid arrow from the reorder buffer of the A-stream (left-most processor) to the delay buffer. Value history is partial because only a subset of the program is executed by the A-stream. Complete control history is available, however, because the instruction-removal process involves predicting all control flow first and then modifying it so that the A-stream may skip instructions (the job of the combined IR-predictor/trace predictor, described in Section 2.1.1).

The R-stream pops control and data flow information from the delay buffer. This is shown in Figure 1 with solid arrows from delay buffer to the instruction cache and execution core of the R-stream (right-most processor). Branch outcomes from the delay buffer are routed to the instruction cache to direct instruction fetching. Source operand values and load/store addresses from the delay buffer are merged with their respective instructions after the instructions have been fetched/renamed and before they enter the execution engine. To know which values/addresses go with which instructions, the delay buffer also includes information about which instructions were skipped by the A-stream (for which there is no data flow information available).

Notice that neither the A-stream nor the R-stream use the conventional branch predictors in their respective processors. This is indicated with an open-switch symbol between branch predictors and instruction caches in Figure 1. As already mentioned, the IR-predictor/trace predictor provides all branch predictions to the A-stream. For branch-related computation that is executed in the A-stream, the corresponding branch predictions are validated — although validation itself may be speculative due to removal of

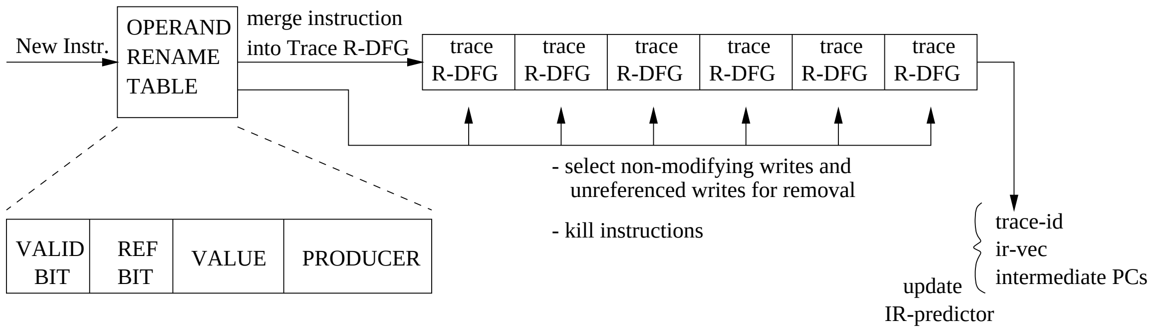

**Figure 3: IR-detector.**

other, presumed-ineffectual computation. For branch-related computation that is bypassed, branch predictions are presumed correct. Via the delay buffer, the R-stream sees a complete branch history as determined by the A-stream — whether it is correct or not — and the conventional branch predictor is not used.

The delay buffer contents can be summarized as follows.

- *Control flow*: Control flow outcomes are encoded as the sequence of trace ids and corresponding *ir-vecs*. The *ir-vec*, which encodes instructions not executed in the A-stream, is used by the R-stream to match data flow outcomes with instructions.

- *Data flow*: There is an entry in the data flow buffer for each instruction executed in the A-stream. An entry contains operand register names and values and load/store addresses.

## 2.3 IR-misprediction Detection and Recovery

An *instruction-removal misprediction*, or IR-misprediction, occurs when A-stream instructions were removed that should not have been. The A-stream has no way of detecting the IR-misprediction, therefore, it continues instruction retirement and corrupts its architectural state. Two things are required to recover from an IR-misprediction. First, the IR-misprediction must be detected and, second, the corrupted state must be pinpointed for efficient recovery actions.

We break down IR-mispredictions into two types, the first type is removal of mispredicted branches and the second type is removal of effectual writes.

1. *Removal of mispredicted branches*. The trace predictor may mispredict a branch and the IR-predictor removes the branch from the A-stream. The R-stream will detect this IR-misprediction because the branch outcome from the delay buffer will differ from the R-stream's computed branch outcome. I.e., it appears as a branch misprediction in the R-stream.

2. *Removal of effectual writes*. The IR-predictor predicts a store instruction is an unreferenced write but the store is actually referenced in the future (or the IR-predictor predicts the store is a non-modifying write but it actually produces a new value at the location). Removing the store instruction causes dependent loads to load an incorrect value, uses of the load value will produce more incorrect values, etc. The first incorrect source operand value popped from the delay buffer will be detected by the R-stream as a value misprediction — in our implementation, source operand value prediction is used [17].

   This kind of IR-misprediction may be detected by the R-stream well after the store was initially removed. The IR-detector can detect these IR-mispredictions much sooner by comparing its computed *ir-vecs* against the corresponding predicted *ir-vecs* — if they differ, computation was removed that should not have been. Thus, the IR-detector serves the dual-role of updating the IR-predictor *and* checking for IR-mispredictions. Although checking by both the R-stream and IR-detector is redundant, it will become clear why final checking by the IR-detector is *required* when we explain recovery, below.

When an IR-misprediction is detected, the reorder buffer of the R-stream is flushed. The R-stream architectural state now represents a precise point in the program to which all other components in the processor are re-synchronized. The trace predictor/IR-predictor is backed up to the precise program counter, the delay buffer is flushed, and the reorder buffer of the A-stream is flushed and its program counter set to that of the R-stream.

All that remains is restoring the corrupted register and memory state of the A-stream so it is consistent with the R-stream. Because register state is finite, the entire register file of the R-stream is copied to the A-stream register file. The movement of data (both register and memory values) occurs via the delay buffer, in the reverse direction, as shown with dashed arrows in Figure 1.

During normal operation, the *recovery controller* receives control signals and the addresses of store instructions from the A-stream, the R-stream, and the IR-detector, as shown in Figure 1. The control signals indicate when to start or stop tracking a memory address (only unique addresses need to be tracked). After detecting an IR-misprediction, stores may either have to be "undone" or "done" in the A-stream. These two cases are described below.

- The addresses of stores retired by the A-stream but not yet checked/retired by the R-stream will need to be restored after detecting an IR-misprediction. In effect, the A-stream store must be "undone" since the R-stream has not yet performed the store. These stores only need to be tracked between the time they are retired from the A-stream and the companion (redundant) stores are retired from the R-stream, as shown in Figure 4 ("store 1").

- Although *all* IR-mispredictions are *eventually* detectable as a value misprediction in the R-stream, a virtually unbounded number of addresses would need to be tracked by the recovery controller if we did not place a "time limit" on the detection of IR-mispredictions. This is why the IR-detector compares predicted *ir-vecs* against computed *ir-vecs*. The recovery controller tracks addresses of stores retired in the R-stream and skipped in the A-stream, only until the IR-detector verifies that the stores are truly ineffectual, as shown in Figure 4 ("store 2"). When an IR-misprediction is detected, all unverified, predicted-ineffectual stores are "done" in the A-stream by copying data from the redundant locations in the R-stream.

## 3. TRANSIENT FAULT TOLERANCE

A formal analysis of the fault tolerance of slipstream processors is left for future work. For now, we informally analyze three key scenarios, shown in Figure 5, to better understand *potential* fault tolerance. In Figure 5, the horizontal lines represent the dynamic instruction streams of the A-stream and R-stream, with older instructions on the left. For this simple analysis, we assume only a single fault occurs and that the fault is ultimately manifested as an erroneous value. A single fault can affect instructions in both streams simultaneously. This is not a problem because the two redundantly-executed copies of an instruction execute at different times (*time redundancy*) [24], therefore, a single fault that affects both streams will affect different instructions. Since only one copy of an instruction is affected by a fault, we arbitrarily choose the R-stream copy, indicated with X's in Figure 5. An X indicates the first erroneous instruction in program order.

Scenario #1 in Figure 5 shows the A-stream and R-stream executing redundantly, i.e., all instructions overlap and have the same data flow. The fault is detectable because the operands of the first

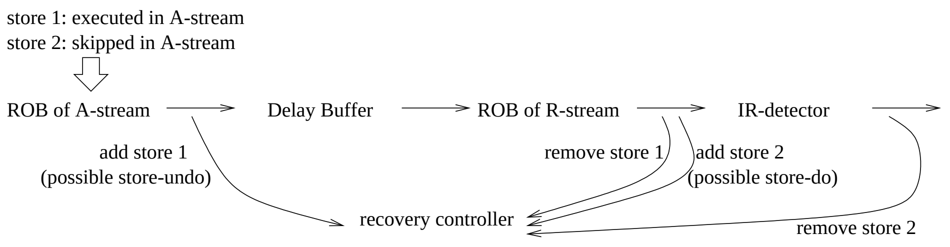

**Figure 4: Tracking memory addresses for potential recovery.**

erroneous instruction differ between A-stream and R-stream. Without more information, however, *the fault is indistinguishable from an IR-misprediction*. Under the circumstances, the processor must assume an IR-misprediction since misspeculation is by far the common case. We point out three successively stronger fault tolerance claims.

1. If we assume a fault cannot flip bits in the R-stream's *architectural* state, then it does not matter that faults and IR-mispredictions are indistinguishable. Recovery succeeds using the R-stream state. Under this model, faults in the pipeline are transparently recoverable. Faults that hit the R-stream register file and data cache are unrecoverable, and worse, undetectable as a fault.

2. If all IR-predictions prior the first erroneous instruction have been verified, then the source of error is known to be a fault. Software is invoked to diagnose the system and perform recovery operations (e.g., restart). But we default back to (1) if there are prior unresolved IR-predictions.

3. ECC can be used to protect the R-stream register file and data cache, in which case all transient faults within scenario #1 are transparently recoverable.

Scenario #2 in Figure 5 shows a region of the program that is not executed redundantly (the A-stream bypassed these instructions). A transient fault in the R-stream is undetectable because there is nothing to compare the erroneous values with. Although an error may be detected in later, redundantly-executed instructions, the R-stream architectural state is already corrupted and the system is unaware of this fact.

Scenario #3 shows the A-stream diverging from the R-stream due to an IR-misprediction, and a transient fault occurs after the divergent point. The IR-misprediction is detected and subsequent erroneous instructions are flushed before the fault can do damage.

In summary, slipstream processors potentially improve the fault tolerance of the chip. The system transparently recovers from transient faults affecting redundantly-executed instructions.

## 4. SIMULATION ENVIRONMENT

We developed a detailed execution-driven simulator of a slipstream processor. The simulator faithfully models the architecture depicted in Figure 1 and outlined in Section 2: the A-stream produces real, possibly incorrect values/addresses and branch outcomes, the R-stream and IR-detector check the A-stream and initiate recovery actions, A-stream state is recovered from the R-stream state, etc. The simulator itself is validated via a functional simulator run independently and in parallel with the detailed timing simulator [33]. The functional simulator checks retired R-stream control flow and data flow outcomes.

The Simplescalar [3] compiler and ISA are used. Binaries are compiled with -O3 level optimization. The Simplescalar compiler is gcc-based and the ISA is MIPS-based; as a result, programs inherit any inefficiencies of the gnu compiler and MIPS ISA. We used the SPEC95 integer benchmarks, shown in Table 1, for evaluation. Benchmarks were run to completion.

**Table 1: Benchmarks.**

| benchmark | input dataset | instr. count |
|---|---|---|
| compress | 40000 e 2231 | 248 million |
| gcc | -O3 genrecog.i -o genrecog.s | 117 million |
| go | 9 9 | 133 million |
| jpeg | vigo.ppm | 166 million |
| li | test.lsp (queens 7) | 202 million |
| m88ksim | -c < ctl.in (dcrand.big) | 121 million |
| perl | scrabble.pl < scrabble.in (dictionary) | 108 million |
| vortex | vortex.in (persons.250, bendian.*) | 101 million |

Microarchitecture parameters are enumerated in Table 2. The CMP is composed of two conventional 4-way superscalar processors, each with private instruction and data caches and a 64-entry ROB (a shared level-two cache always hits). A large IR-predictor/trace predictor is used for accurate instruction removal. For all experiments, the IR-predictor/trace predictor uses length-32 traces and a resetting-counter confidence threshold of 32. The IR-detector has a scope of 8 length-32 traces. The delay buffer length is 256 instructions. The recovery controller tracks any number of store addresses, although we observe not too many outstanding addresses in practice. The recovery latency (*after* the IR-mispre-

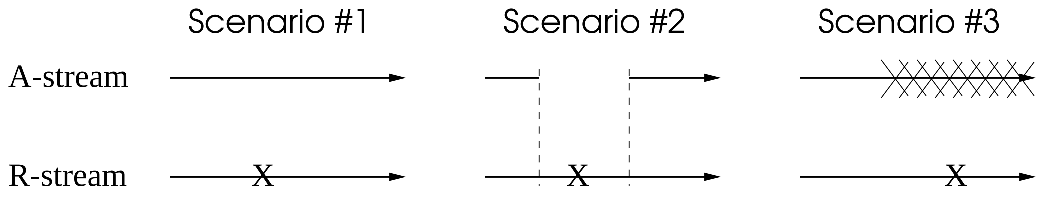

**Figure 5: Transient fault scenarios.**

diction is detected) is 5 cycles to startup the recovery pipeline, followed by 4 register restores per cycle, and lastly 4 memory restores per cycle. As there are 64 general purpose integer and floating point registers, the minimum recovery latency is 21 cycles (5 + 64/4) if no memory locations are restored.

**Table 2: Microarchitecture configuration.**

| | |
|---|---|
| **single processor** | |
| **instruction cache** | fetch bandwidth: • 2-way interleaved to fetch full cache block • fetch past multiple *not-taken* branches in single cycle |
| | size/assoc/repl = 64kB/4-way/LRU |
| | line size = 16 instructions |
| | miss penalty = 12 cycles |
| **data cache** | size/assoc/repl = 64kB/4-way/LRU |
| | line size = 64 bytes |
| | miss penalty = 14 cycles |
| **superscalar core** | reorder buffer (default): 64 entries |
| | dispatch/issue/retire bandwidth (default): 4-way |
| | *n* fully-symmetric function units (*n* = issue bandwidth) |
| | *n* loads/stores per cycle (*n* = issue bandwidth) |
| **execution latencies** | address generation = 1 cycle |
| | memory access = 2 cycles (hit) |
| | integer ALU ops = 1 cycle |
| | complex ops = MIPS R10000 latencies |
| **slipstream components** | |
| **IR-predictor** | trace predictor (hybrid): • $2^{16}$-entry path-based pred.: 8 traces in path history • $2^{16}$-entry simple pred.: 1 trace in path history |
| | resetting-counter confidence threshold = 32 |
| **IR-detector** | trace length (R-DFG size) = 32 instructions |
| | scope = 8 traces/256 instructions |
| **delay buffer** | data flow buffer: 256 instruction entries |
| | control flow buffer: 128 {*trace-id*, *ir-vec*} pairs |
| **recovery controller** | number of outstanding store addresses = unconstrained |
| | recovery latency (*after* IR-misprediction detection): • 5 cycles to start up recovery pipeline • 4 register restores per cycle (64 regs performed first) • 4 memory restores per cycle (mem performed second) • $\therefore$ minimum latency (no memory) = 21 cycles |

## 5. RESULTS

The performance of three models is presented.

- SS(64x4) — A single copy of the program is run on one conventional 4-way superscalar processor with 64 ROB entries.
- SS(128x8) — A single copy of the program is run on one conventional 8-way superscalar processor with 128 ROB entries.
- CMP(2x64x4) — This is a slipstream processor using a CMP composed of two SS(64x4) cores.

For fair and direct comparisons, *the same trace predictor is used for accurate and high-bandwidth control flow prediction in all three processor models*. Of course, only CMP(2x64x4) uses an IR-predictor on top of the trace predictor. Performance is measured in retired instructions per cycle (IPC). IPC for the slipstream processor is computed as the number of retired R-stream instructions (i.e., the full program, counted only once) divided by the number of cycles for both the A-stream and R-stream to complete (total execution time).

The graph in Figure 6 shows the IPC improvement of CMP(2x64x4) with respect to SS(64x4). (For a point of reference, the IPC of SS(64x4), our base model, is given in Table 3.) CMP(2x64x4) improves performance by 7% on average. The IPC of half of the benchmarks improve by more than 7% — *li*, *vortex*, *perl*, and *m88ksim* improve by 7%, 7%, 16%, and 20%, respectively — while *gcc* improves by 4% and the other three show little or no improvement. The significance of this result is that a second, *otherwise unused* processor on the chip can be exploited for improving single-program performance.

The performance improvement due to doubling the window size and issue bandwidth of the superscalar processor is shown in Figure 7. On average, SS(128x8) improves performance by 28%. We feel the slipstream paradigm has competitive potential.

1. With the *initial* and *relatively unexplored* slipstream implementation, we achieve one-fourth the IPC-performance gains of the larger superscalar processor. And if superscalar complexity is considered, then a CMP composed of two small superscalar processors will potentially have a faster cycle time than one large superscalar processor.
2. A CMP with slipstreaming provides more functionality and flexibility than a single superscalar processor. For example, depending on the load of the machine, the extra processor may be used to run another job or cooperate with the other processor on a single job.
3. The peak bandwidth of CMP(2x64x4) is only 4 IPC, hence there is less room for improvement than with SS(128x8). This suggests implementing a slipstream processor using an 8-wide SMT processor, which we leave for future work.

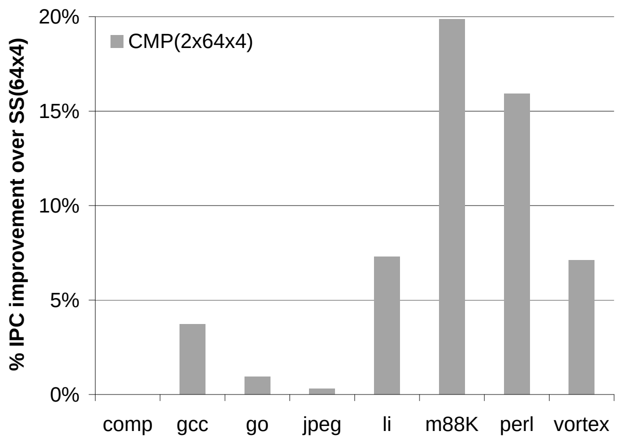

**Figure 6: Performance of CMP(2x64x4) (slipstream processor) with respect to SS(64x4).**

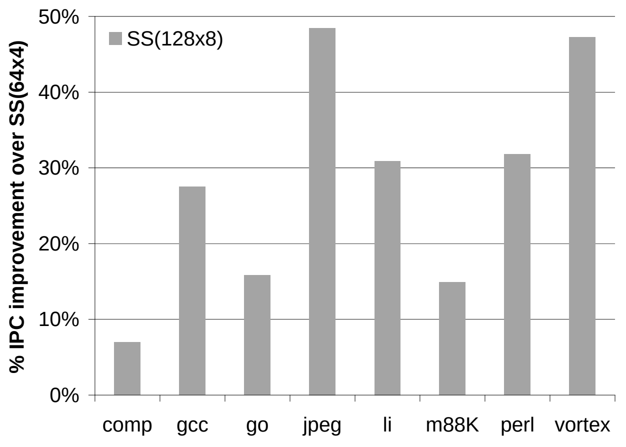

**Figure 7: Performance of SS(128x8) with respect to SS(64x4).**

The uppermost graph in Figure 8 shows the fraction of dynamic instructions removed from the A-stream for each of the benchmarks. Each bar is broken down into the sources of instruction-removal.

- BR: branch instruction.
- WW: A write followed by a write to the same location, with no intervening reference.
- SV: Writing the same value to a location.
- P:{BR | WW | SV}: Instructions that were removed due to back-propagation from other removed instructions. These instructions *inherit* any combination of BR, WW, and SV status.

The number of removed instructions correlates closely with performance improvement. Nearly 50% of *m88ksim*'s instructions were removed and it has the largest performance improvement (20%). Successively fewer instructions were removed in *perl*, *vortex*, *li*, and *gcc* — 20%, 16%, 10%, and 8% of all instructions were removed, respectively — and performance reflects this. On average, the three largest sources of instruction removal are BR (33% *of the removed instructions*), SV (30%), and P:BR (27%). We have observed that WW and SV tend to occur simultaneously and priority is given to SV when accounting.

The lowermost graph in Figure 8 shows what happens when only branch instructions (BR) and their computation chains (P: BR) are candidates for removal, i.e., ineffectual writes are not removed. This is relevant because branch predictability is more likely to be influenced by algorithm than by compiler, whereas the compiler may have more influence on ineffectual writes. Interestingly, the fraction of removed instructions increases substantially for all benchmarks except *m88ksim*, whose fraction drops from half to one-quarter. The results are counterintuitive because there is less opportunity for instruction-removal when ineffectual writes are not considered. *Diluted confidence*, discussed in Section 2.1.3, may explain the results. With fewer candidate instructions for removal, there is also less chance that unrelated instructions dilute the confidence of consistently-removable branches. Overall confidence is higher and more instructions are removed in practice, despite less total opportunity. The average IPC improvement with only branch-removal remains at 7%, but per-benchmark IPCs change: *perl* (16%), *li* (11%), *m88ksim* (11%), *vortex* (7%), and *gcc* (5%).

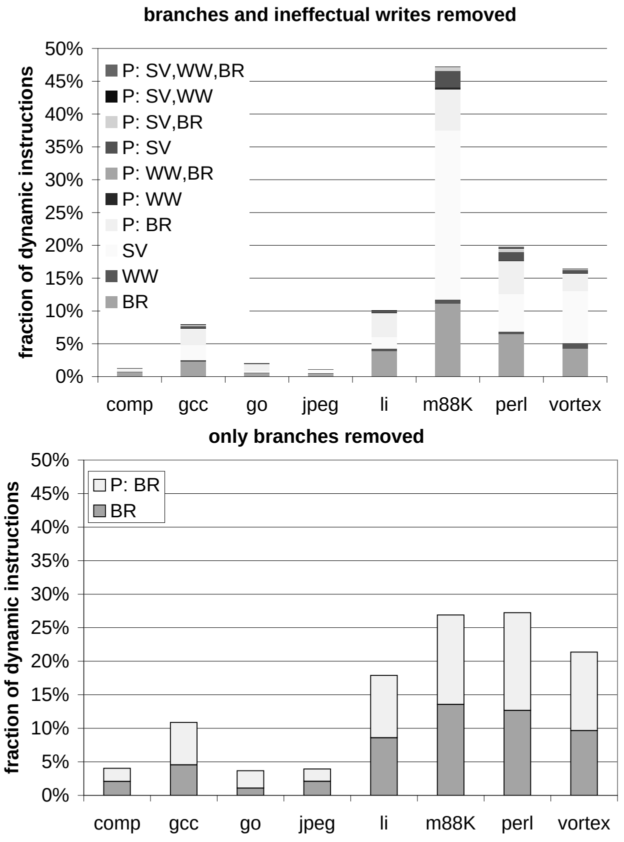

**Figure 8: Breakdown of removed A-stream instructions.**

Branch mispredictions per 1000 instructions for each benchmark is provided in Table 3. A key observation is that instruction-removal is most successful for highly-branch-predictable benchmarks. The *gcc* benchmark is an interesting case. Although its branch misprediction rate is similar to *li*'s, IR-prediction is more successful with *li*. *Unstable traces*, discussed in Section 2.1.3, may explain the discrepancy. We hypothesize *gcc*, more than *li*, has consistently-removable branches and unpredictable branches grouped together in traces. The traces are unstable and the consistently-removable branches are not removed in practice. Using a non-trace-based IR-predictor could fix the problem.

The trace predictor's update latency (which is accurately modeled) increases with slipstreaming. Comparing the second and third rows of Table 3, the effect on branch misprediction rate is not too severe, in fact, delayed updates reduce the rate slightly for *li*, *m88ksim*, and *perl*. The confidence threshold of 32 results in fewer than 0.05 IR-mispredictions per 1000 instructions. And the average IR-misprediction penalty is at most 26 cycles, close to the minimum of 21 cycles, which implies only a handful of memory locations need to be restored after an IR-misprediction.

**Table 3: Misprediction measurements.**

| | | comp | gcc | go | jpeg | li | m88k | perl | vortex |
|---|---|---|---|---|---|---|---|---|---|
| SS(64x4) | IPC | 1.72 | 2.69 | 2.15 | 3.24 | 2.88 | 2.82 | 3.08 | 3.24 |
| | branch misp./1000 instr. | 16 | 6.4 | 11 | 4.1 | 6.5 | 1.9 | 2.0 | 1.1 |
| CMP(2x64x4) | branch misp./1000 instr. | 16 | 6.6 | 11 | 4.2 | 6.2 | 1.8 | 1.9 | 1.1 |
| | IR-mispredictions/1000 instr. | 0.03 | 0.03 | 0.02 | 0.01 | 0.02 | 0.03 | 0.02 | 0.05 |
| | avg. IR-misprediction penalty | 22 | 23 | 22 | 22 | 23 | 24 | 24 | 26 |

## 6. RELATED WORK

*Advanced-stream/Redundant-stream Simultaneous Multithreading* (AR-SMT) [24] is based on the realization that microarchitecture performance trends and fault tolerance are related. Time redundancy — running a program twice to detect transient faults — is cheaper than hardware redundancy but it doubles execution time. AR-SMT runs the two programs simultaneously [37] but delayed (via the delay buffer), reducing the performance overhead of time redundancy. Results are compared by communicating all retired A-stream results to the R-stream, and the R-stream performs the checks. Here, the R-stream leverages speculation concepts [17] — the A-stream results can be used as ideal predictions. The R-stream fetches/executes with maximum efficiency, further reducing the performance overhead of time redundancy. And the method for comparing the A-stream and the R-stream is conveniently in place, in the form of misprediction-detection hardware. In summary, AR-SMT leverages the underlying microarchitecture to achieve broad coverage of transient faults with low overhead, both in terms of performance and changes to the existing design.

DIVA [2] and SRT [22] are two other examples of fault-tolerant architectures designed for commodity high-performance microprocessors. DIVA detects a variety of faults, *including design faults*, by using a verified checker to validate computation of the complex processor core. DIVA leverages an AR-SMT technique — the simple checker is able to keep pace with the core by using the values it is checking as predictions. SRT improves on AR-SMT in a variety of ways, including a formal and systematic treatment of SMT applied to fault tolerance (e.g., *spheres of replication*).

Researchers have demonstrated a significant amount of redundancy, repetition, and predictability in general purpose programs [6,9,10,17,18,19,30,32]. This prior research forms a basis for creating the shorter program in slipstream processors. A technical report [25] showed 1) it is possible to ideally construct significantly reduced programs that produce correct final output, and 2) AR-SMT is a convenient execution model to exploit this property.

Tullsen et. al. [36,37] and Yamamoto and Nemirovsky [39] proposed simultaneous multithreading for flexibly exploiting thread-level and instruction-level parallelism. Olukotun et. al. [20] motivate using single-chip multiprocessors.

Farcy et. al. [8] proposed resolving branch mispredictions early by extracting the computation leading to branches. Zilles and Sohi [41] similarly studied the computation chains leading to mispredicted branches and loads that miss in the level-two cache. They suggest identifying a difficult subset of the program for *pre-execution* [27,28], potentially prefetching branch predictions and cache lines that would otherwise be mispredictions and cache misses. Pre-execution typically involves pruning a small kernel from a larger program region and running it as a prefetch engine [26]. Roth and Sohi [28] developed a new paradigm called *Data-Driven Speculative Multithreading* that implements pre-execution. Rather than spawn many specialized kernels on-the-fly, our approach uses a single, *bona fide* program (A-stream). That is, the A-stream's context is persistent and redundant with the R-stream, with several key advantages. First, we avoid the conceptual and possibly real complexity of forking private contexts, within which the specialized kernels must run. Second, Zilles points out there may be difficulty in binding prefetched predictions to fetched branches [41], whereas the one-to-one correspondence between redundant instructions in the A-stream and R-stream avoids this problem entirely. Third, redundant programs can be exploited for transient fault tolerance.

Speculative multithreading architectures [e.g.,1,7,21,33,34,35] speed up a single program by dividing it into speculatively-parallel threads. The speculation model uses *one architectural context* and future threads are spawned within temporary, private contexts, each inherited from the preceding thread's context. Future thread contexts are merged into the architectural context as threads complete. Our speculation model uses redundant architectural contexts, so no forking or merging is needed. And strictly speaking, there are no dependences between the architecturally-independent threads, rather, outcomes are communicated as predictions via a simple FIFO queue. Register and memory mechanisms of the underlying processor are relatively unchanged by slipstreaming (particularly if there is an existing interface for consuming value predictions at the rename stage). In contrast, speculative multithreading often requires elaborate inter-thread register/memory dependence mechanisms. Besides performance, using redundant contexts adds other value to the chip, i.e., fault tolerance. We are not advocating one kind of multithreading model over another, rather, we are proposing another alternative and pointing out its novel implications.

Running background threads to perform some function on behalf of the primary program is increasing in popularity. SSMT [5] is a generic approach in which a subordinate thread monitors events in the primary thread (e.g., mispredictions and cache misses) and adjusts hardware components to compensate and optimize performance. Subordinate threads also allow exception handling to proceed in parallel with code after the excepting instruction [40].

The DataScalar paradigm [4] runs redundant programs on multiple processor-and-memory cores to eliminate memory read requests. DataScalar trades relatively inexpensive computing power for reduced memory traffic.

## 7. CONCLUSIONS AND FUTURE WORK

Making effective use of a billion transistors is a major challenge. Simultaneous multithreading and chip multiprocessing payoff substantially in this regard, because existing parallelism can be migrated from the system level to the chip level *relatively* easily. Even larger payoffs are possible if the same transistors are reused for single-program performance and functions normally reserved for niche computers. The slipstream paradigm allows the operating system to flexibly choose among multiple operating modes based on system and user requirements. The requirements may include: high job throughput and parallel-program performance (conventional SMT/CMP), improved single-program performance and reliability (slipstreaming), or fully-reliable operation with little or no impact on single-program performance (AR-SMT / SRT).

A slipstream processor simultaneously runs two copies of the program. One of the programs (A-stream) always runs slightly ahead of the other (R-stream). The R-stream is monitored for ineffectual and branch-predictable computation, and the information learned is used to speculatively but accurately reduce the A-stream. Outcomes from the A-stream are communicated to the R-stream. The R-stream uses the outcomes to execute more efficiently and, at the same time, validate the speculative A-stream. The two programs combined finish sooner than either would alone. A detailed but relatively-unexplored implementation demonstrates substantial performance improvements are possible, 7% on average.

The shorter program is a subset of the full program and this partial-redundancy is transparently leveraged for detecting and recovering from transient hardware faults. The importance of providing reliability with low overhead — both in terms of design changes and performance — cannot be overstated. For example, a recent conference panel [11] debated the problem of using *commercial off-the-shelf* components (COTS) in reliable applications. Commodity components are inexpensive and high-performance, in part because they lack fault tolerance — therein lies the quandary. While software is currently the larger problem, future chips are susceptible due to technology and microarchitecture trends.

There are many future research topics for slipstream processors. Below, we discuss some of the more pressing topics.

- We need a better understanding of slipstream performance to identify bottlenecks and ultimately produce more effective A-streams. A high priority is determining the amount and quality of instruction-removal needed to improve performance, and then developing effective IR-predictors based on the results.
- Basic microarchitecture research is needed to develop mechanisms for each of the new slipstream components, explore the design space, and optimize the components for both practical implementation and good performance. We also want to demonstrate the new components interface to a conventional pipeline without fundamentally reorganizing it.
- Slipstreaming needs to be implemented on an SMT core and, in general, we should evaluate multiple CMP/SMT configurations. SMT introduces new problems, such as competition for resources. Adaptively turning on/off slipstreaming may be needed, so performance is not degraded when the A-stream is only slightly reduced. Adaptivity is also useful in a CMP, to determine whether or not the second PE should instead be used for an independent program.
- For reliability, we need to formally analyze fault coverage and also improve coverage under partial-redundancy constraints.
- The current slipstream model, due to process replication, has many system-level issues that need to be addressed (coherence and consistency, O/S support, interrupts, I/O, etc.). For example, we are looking at ways of reducing memory overhead while retaining the simplicity of software memory renaming.

## ACKNOWLEDGMENTS

The authors would like to thank Tom Conte, Alvy Lebeck, and the NC State architecture group for their helpful feedback. We are grateful to Jim Smith for suggesting the name “slipstream” and pointing out the useful car racing analogy.

## REFERENCES

[1] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. *31st Int’l Symp. on Microarchitecture*, Dec. 1998.

[2] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. *32nd Int’l Symp. on Microarchitecture*, Nov. 1999.

[3] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The Simplescalar Toolset. Technical Report CS-TR-96-1308, Computer Sciences Department, University of Wisconsin - Madison, July 1996.

[4] D. Burger, S. Kaxiras, and J. Goodman. DataScalar Architectures. *24th Int’l Symp. on Computer Architecture*, June 1997.

[5] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Microthreading (SSMT). *26th Int’l Symp. on Computer Architecture*, May 1999.

[6] D. Connors and W.-M. Hwu. Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results. *32nd Int’l Symp. on Microarchitecture*, Nov. 1999.

[7] P. Dubey, K. O’Brien, K. M. O’Brien, and C. Barton. Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading. *Parallel Architectures and Compiler Techniques*, June 1995

[8] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and its Application to Early Resolution of Branch Outcomes. *31st Int’l Symp. on Microarchitecture*, Dec. 1998.

[9] A. González, J. Tubella, and C. Molina. Trace-Level Reuse. *Int’l Conf. on Parallel Processing*, Sep. 1999.

[10] J. Huang and D. Lilja. Exploiting Basic Block Value Locality with Block Reuse. *5th Int’l Symp. on High-Performance Computer Architecture*, Jan. 1999.

[11] R. Iyer, A. Avizienis, D. Barron, D. Powell, H. Levendel, and J. Samson. Panel: Using COTS to Design Dependable Networked Systems. *29th Int’l Symp. on Fault-Tolerant Computing*, June 1999.

[12] E. Jacobsen, E. Rotenberg, and J. Smith. Assigning Confidence to Conditional Branch Predictions. *29th Int’l Symp. on Microarchitecture*, Dec. 1996.

[13] Q. Jacobson, E. Rotenberg, and J. Smith. Path-Based Next Trace Prediction. *30th Int'l Symp. on Microarchitecture*, Dec. 1997.

[14] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification. *31st Int'l Symp. on Microarchitecture*, Nov. 1998.

[15] K. Lepak and M. Lipasti. On the Value Locality of Store Instructions. *27th Int'l Symp. on Computer Architecture*, June 2000.

[16] M. Lipasti, C. Wilkerson, and J. Shen. Value Locality and Load Value Prediction. *7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[17] M. Lipasti. Value Locality and Speculative Execution. Ph.D. Thesis, Carnegie Mellon University, April 1997.

[18] M. Martin, A. Roth, and C. Fischer. Exploiting Dead Value Information. *30th Int'l Symp. on Microarchitecture*, Dec. 1997.

[19] C. Molina, A. Gonzalez, and J. Tubella. Reducing Memory Traffic via Redundant Store Instructions. *HPCN Europe*, 1999.

[20] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. *7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[21] J. Oplinger, D. Heine, S.-W. Liao, B. Nayfeh, M. Lam, and K. Olukotun. Software and Hardware for Exploiting Speculative Parallelism in Multiprocessors. CSL-TR-97-715, Stanford University, Feb. 1997.

[22] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. *27th Int'l Symp. on Computer Architecture*, June 2000.

[23] D. Ronfeldt. Social Science at 190 MPH on NASCAR's Biggest Superspeedways. *First Monday Journal* (on-line), Vol. 5 No. 2, Feb. 7, 2000.

[24] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. *29th Int'l Symp. on Fault-Tolerant Computing*, June 1999.

[25] E. Rotenberg. Exploiting Large Ineffectual Instruction Sequences. Technical Report, Department of Electrical and Computer Engineering, North Carolina State University, Nov. 1999.

[26] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. *8th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[27] A. Roth and G. Sohi. Speculative Data Driven Sequencing for Imperative Programs. Technical Report CS-TR-2000-1411, Computer Sciences Department, University of Wisconsin - Madison, Feb. 2000.

[28] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. Technical Report CS-TR-2000-1414, Computer Sciences Department, University of Wisconsin - Madison, April 2000.

[29] P. Rubinfeld. Virtual Roundtable on the Challenges and Trends in Processor Design: Managing Problems at High Speeds. *Computer*, 31(1):47-48, Jan. 1998.

[30] Y. Sazeides and J. E. Smith. Modeling Program Predictability. *25th Int'l Symp. on Computer Architecture*, June 1998.

[31] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. *24th Int'l Symp. on Computer Architecture*, June 1997.

[32] A. Sodani and G. S. Sohi. An Empirical Analysis of Instruction Repetition. *8th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[33] G. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. *22nd Int'l Symp. on Computer Architecture*, June 1995.

[34] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. *4th Int'l Symp. on High-Performance Computer Architecture*, Feb. 1998.

[35] J.-Y. Tsai and P.-C. Yew. The Superthreaded Architecture: Thread Pipelining with Run-time Data Dependence Checking and Control Speculation. *Parallel Architectures and Compiler Techniques*, 1996.

[36] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *22nd Int'l Symp. on Computer Architecture*, June 1995.

[37] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. *23rd Int'l Symp. on Computer Architecture*, May 1996.

[38] D. Tullsen and J. Seng. Storageless Value Prediction Using Prior Register Values. *26th Int'l Symp. on Computer Architecture*, May 1999.

[39] W. Yamamoto and M. Nemirovsky. Increasing Superscalar Performance through Multistreaming. *Parallel Architectures and Compilation Techniques*, June 1995.

[40] C. Zilles, J. Emer, and G. Sohi. The Use of Multithreading for Exception Handling. *32nd Int'l Symp. on Microarchitecture*, Nov. 1999.

[41] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. *27th Int'l Symp. on Computer Architecture*, June 2000.