A Study of Slipstream Processors

Zach Purser Karthik Sundaramoorthy Eric Rotenberg

North Carolina State University

Department of Electrical and Computer Engineering

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

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

Abstract

A slipstream processor reduces the length of a running program by dynamically skipping computation non-essential for correct forward progress. The shortened program runs faster as a result, but it is speculative. So a second, unreduced copy of the program is run concurrently with and slightly behind the reduced copy — leveraging a chip multiprocessor (CMP) or simultaneous multithreading (SMT). The short program passes its control and data flow outcomes to the full program for checking. And as it checks the short program, the full program fetches and executes more efficiently due to having an accurate picture of the future. Both programs are sped up: combined, they outperform conventional non-redundant execution.

We study slipstreaming with the following key results.

1. *A 12% average performance improvement is achieved by harnessing an otherwise unused, additional processor in a CMP. Slipstreaming using two small superscalar cores often achieves similar instructions-per-cycle as one large superscalar core, but with a potentially faster clock and a more flexible architecture.*
2. *A majority of the benchmarks show significant reduction in the short program (about 50%). Slipstreaming using an 8-way SMT processor improves their performance from 10% to 20%.*
3. *For some benchmarks, including* gcc, *performance improvement is due to the short program resolving branch mispredictions in advance. Others benefit largely due to value predictions from the short program, and the effect is not always reproducible by conventional value prediction tables.*
4. *As execution bandwidth is increased, slipstreaming provides less of a performance advantage* — unless instructions are removed in the short program before they are fetched. *A simple program sequencing mechanism is developed to bypass instruction fetching.*

1. Introduction

The slipstream paradigm [21,27] proposes only a fraction of the dynamic instruction stream is needed for a program to make full, correct, forward progress. For example, some instruction sequences have no observable effect. They produce results that are not subsequently referenced, or results that do not change the state of the machine. And then there are instruction sequences whose effects are observable, but the effects are invariably predictable. Computation influencing control flow is the most notable example.

Ineffectual and branch-predictable computation can be exploited to reduce the length of a running program, speeding it up. Unfortunately, we cannot know for certain what instructions can be validly skipped. Constructing a shorter program is speculative and, ultimately, it must be checked against the full program to verify it produces the same overall effect.

Therefore, a slipstream processor concurrently runs two copies of the program, leveraging either a single-chip multiprocessor (CMP) [17] or a simultaneous multithreading processor (SMT) [28,31] (the user program is instantiated twice by the operating system and each copy has its own context). One program 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 R-stream and detects 1) instructions that repeatedly and predictably have no observable effect (e.g., unreferenced writes, non-modifying writes) and 2) branches whose outcomes are consistently predicted correctly. Future instances of the ineffectual instructions, branch instructions, and the computation chains leading up to them are speculatively removed in the A-stream — but only if there is high confidence correct forward progress can still be made, in spite of removing the instructions.

The reduced A-stream fetches, executes, and retires fewer instructions than it would otherwise, resulting in a faster program. To verify that the A-stream makes correct forward progress, all control and data flow outcomes of the A-stream are passed to the R-stream. The R-stream checks the outcomes against its own and, if a deviation is detected, the R-stream's architectural state is used to selectively repair the A-stream's corrupted architectural state (an infrequent event).

A key point is the R-stream uses the outcomes it is checking as predictions [20]. This has two advantages.

- First, the R-stream fetches and executes more efficiently due to having near-ideal predictions from the A-stream. Thus, although the unreduced R-stream retires more instructions, it keeps pace with the A-stream and the two programs combined finish sooner than a single copy of the program would. The slipstream processor's approach of speeding up a single program via redundancy is analogous to "slipstreaming" in car racing, where two cars race nose-to-tail to increase the speed of *both* cars [19].
- Second, by using A-stream outcomes as predictions, *the R-stream leverages existing speculation mechanisms for checking the A-stream*. Conventional processors typically have mechanisms in place to check control flow speculation, and future processors may incorporate value prediction and mechanisms to check data flow speculation.

Another benefit of slipstreaming is improved reliability. Transient faults that affect redundantly-executed instructions are transparently detectable and recoverable [20,27]. Fault detection/recovery is transparent because transient faults are indistinguishable from prediction-induced deviations.

1.1. Contributions

This paper is a follow-up study of our recent slipstream proposal [27] and makes four new contributions.

1. *Understanding slipstreaming*.

   Slipstreaming can be explained and understood in several ways. We describe two different interpretations of slipstreaming, qualitatively explain where its performance improvement is derived from, and expose its limitations. Insight into the limitations of slipstreaming allows us to focus efforts on areas that are likely to payoff.

   More comprehensive experimental results provide important insight and confirm the expectations of our qualitative arguments. Multiple CMP configurations are explored — examining multiple CMP configurations is relevant because conclusions change as the processor cores scale.

2. *Slipstreaming using SMT processors*.

   Slipstreaming was not previously implemented on an SMT processor. Insufficient reduction in the A-stream made SMT-based slipstreaming less viable. Artifacts of our previous instruction-removal mechanism have been addressed (see next item below), so SMT-based slipstreaming is now viable and this paper provides results.

3. *More effective instruction-removal*.

   Previously, removal-confidence was measured for a group of instructions as a whole, i.e., for a trace [27]. A trace-based approach ensures producer instructions are not removed from the A-stream unless corresponding consumer instructions are also removed. Not enforcing this constraint leads to spurious instruction-removal mispredictions.

   Trace-based removal has severe limitations, however [27]. Frequently-varying removal patterns within a trace cause the overall confidence to be low, despite stable patterns among certain dependence chains. As a result, no instructions in the trace are removed even if many are removable. And although traces ensure dependence chains are removed together, chains are confined to the same trace.

   Our new approach measures confidence for instructions individually, so unrelated instructions do not dilute confidence. Yet dependence chains still tend to be removed together and chains are not confined within a small region.

4. *Bypassing instruction fetching*.

   The A-stream is most effective when both the number of instructions fetched and executed are reduced. Reducing the number of fetched instructions requires a different sequencing model than conventional branch predictors currently provide. A conventional branch predictor is modified in a novel and simple way to bypass fetching of large, dynamic instruction sequences.

1.2. Paper outline

The paper is organized as follows. Section 2 develops models for understanding slipstreaming and examines its fundamental limits. Section 3 reviews the slipstream microarchitecture and introduces the new instruction-removal mechanisms. In sections 4 and 5, the simulation environment and results are presented, respectively. Related work is discussed in Section 6 and conclusions in Section 7.

2. Understanding slipstreaming

We present two different interpretations of slipstreaming to better understand the paradigm. In subsection 2.1, the A-stream is interpreted as the "main" thread and the R-stream "assists" the A-stream. In subsection 2.2, roles are reversed: the R-stream is the "main" thread and the A-stream "assists" the R-stream. Actually, the two programs in a slipstream processor are functionally equivalent and mutually beneficial, so either interpretation is valid.

We next examine limits of the paradigm to motivate removing instructions from the A-stream before they are fetched. Finally, we consider other ways of reducing the A-stream to highlight the conceptual simplicity of our chosen approach.

2.1. R-stream: a fast checker

The A-stream does not explicitly derive any performance benefit from the R-stream. Rather, the R-stream checks (and occasionally redirects) the A-stream without slowing it down. This is possible because *checking is inherently parallel* [13,20]. As depicted in Figure 1, *the R-stream is a fast checking assist to the A-stream* [20,21,2].

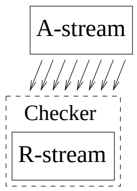

FIGURE 1. A fast checking assist to the A-stream.

2.2. A-stream: a program-based predictor

Alternatively, *the A-stream is a program-based predictor for the R-stream* [7,23,33,5]. For example, the A-stream assists the performance of the R-stream by improving its branch prediction accuracy. Dynamic branch predictions are classified into two groups, *confident* and *unconfident* [10], as shown in Figure 2. Confident branch predictions are more likely to be correct and the corresponding branches and computation feeding the branches are removed from the A-stream. Confident predictions represent the most accurate predictions, therefore, removing the computation needed to verify them is sound, and it allows the A-stream to focus instead on verifying unconfident branch predictions. As a result, *many branch mispredictions are resolved by the A-stream in advance of when the R-stream reaches the same point*.

The A-stream also serves as an accurate value predictor [13] for the R-stream. Although only the results of A-stream-executed instructions are available, the predictions are potentially more accurate than those provided by conventional value predictors: A-stream "predictions" are produced by program computation as opposed to being history-based. Perhaps there is some overlap in what the A-stream provides and what a conventional value predictor could provide. Initial investigations in Section 5.3 indicate some benchmarks (e.g., *gcc*) benefit primarily from the short program resolving branch mispredictions in advance; others benefit largely due to value predictions from the A-stream, and the effect is not always reproducible by conventional value prediction tables. However, comprehensive comparisons are left for future work.

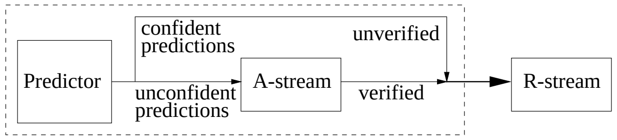

FIGURE 2. A combined predictor/program for improving R-stream branch prediction accuracy.

2.3. Importance of bypassing instruction fetch

Prior research has shown that in the absence of any resource constraints, performance is generally dictated by mispredicted branches [30,11]. That is, in an ideal processor with unconstrained fetch and execution bandwidth, mispredicted branches and their dependence chains tend to dominate the critical path of the program. *The A-stream cannot reduce this critical path* because the dependence chains of mispredicted branches are not safely removable from the A-stream — only correctly predicted branches are safely removable. The A-stream, like a full version of the program, encounters the same mispredictions and resolves them in program order. Therefore, slipstreaming is not likely to provide performance advantages if fetch and execution bandwidth are unconstrained.

Understanding slipstreaming's limitations enables us to focus research efforts on areas that are likely to pay off. For example, we can reason about the relative importance of bypassing instruction fetch and execution in the A-stream. Consider a slipstream processor that reduces the number of instructions *executed* in the A-stream, but not the number of instructions *fetched*. The A-stream runs on one core of a CMP and the R-stream on a second core (for example). As raw execution bandwidth of both cores is increased, the A-stream starts to lose its edge with respect to the R-stream. Instruction fetching becomes the bottleneck and, from a practical standpoint, the A-stream is not truly reduced if the number of fetched instructions is not reduced.

Fortunately, it is possible to bypass even instruction fetching in the A-stream. The A-stream has a distinct advantage in this regard because raw instruction fetch bandwidth cannot be as easily extended as raw execution bandwidth, e.g., due to taken branches and branch predictor bandwidth.

2.4. Other ways of reducing the A-stream

One method for reducing the A-stream is removing branch-predictable computation. Another possibility is removing value-predictable computation. As was described in Figure 2 in the context of branch prediction, an overall better value predictor may be possible by combining a conventional value predictor with the A-stream: the value predictor identifies and removes highly value-predictable computation, and the A-stream focuses instead on hard-to-predict values. The R-stream observes a stream of accurate values comprised of both unverified confident values and computed values.

This approach complicates the mechanism for reducing the A-stream, however. For the A-stream to make correct forward progress, the effects of removed, value-predictable computation must be emulated by updating the state of the A-stream with values directly, similar to block/trace/computation reuse [9,8,6] but without the reuse test. This is why we focused initially on the special cases of ineffectual and branch-predictable computation: this computation can be literally removed (i.e., replaced with nothing), and only the program counter needs to be updated to skip instructions.

3. Microarchitecture description

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 3, although an SMT processor might also be used. 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.

Slipstreaming requires four new components.

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. Also, the IR-predictor indicates which instructions *within* a fetched block can be removed after the instruction fetch stage and before the decode/dispatch stage.
2. The *instruction-removal detector*, or IR-detector, monitors the R-stream and detects instructions that could have been removed from the program, and might possibly be removed in the future. 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 [20].
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.

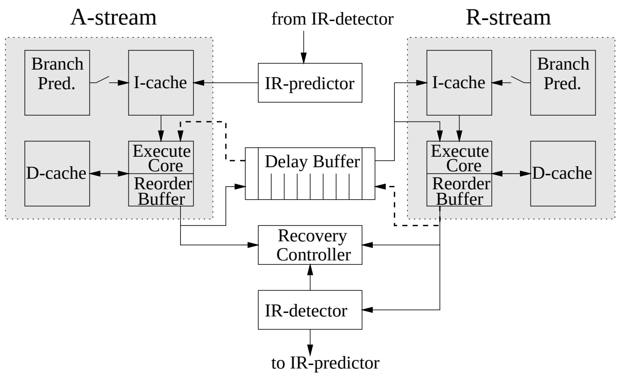

FIGURE 3. Slipstream processor using a two-way chip multiprocessor [27].

The diagram in Figure 3 shows the A-stream on the leftmost core and the R-stream on the rightmost core. This is arbitrary and does not reflect specializing the two cores. A real design would have one core that flexibly supports either the A-stream or R-stream. In any case, there is a clear symmetry that makes designing a single core natural. In both cores, there is an interface to the fetch unit that overrides the conventional branch predictor, indicated symbolically with an open switch and a second interface to the fetch unit. Likewise, both cores show symmetric interfaces to and from the execution pipeline.

3.1. Creating the shorter program

3.1.1. Base IR-predictor. The IR-predictor resembles a conventional branch predictor. In this paper, the IR-predictor is indexed identically to a *gshare* predictor [15], i.e., an index is formed by XORing the PC and the global branch history bits. Each table entry contains information for a single dynamic basic block.

- *Tag*: This is the start PC of the basic block and is used to determine whether or not the entry contains information for the desired block.
- *2-bit counter*: If the block ends in a conditional branch, the 2-bit counter predicts its direction.
- *Confidence counters*. There is a resetting confidence counter [10] for each instruction in the block. The counters are updated by the IR-detector: a counter is incremented if the corresponding instruction is detected as removable, otherwise the counter is reset to zero. If a counter is saturated, then the corresponding instruction will be removed from the A-stream when it is next encountered.

Every fetch cycle, the IR-predictor supplies a branch prediction and an *instruction-removal bit vector* to the A-stream fetch unit. The branch prediction is used to select a PC for the next fetch cycle; potential target PCs are stored within existing structures of the processor, e.g., pre-decoded targets in the instruction cache or branch target buffer.

The instruction-removal bit vector reflects the state of the confidence counters for the basic block being fetched. A bit is set in the vector if the corresponding confidence counter is saturated, and this directs the fetch unit to remove the corresponding instruction from the A-stream. Thus, although all instructions in the basic block are fetched, potentially many instructions are removed before the decode stage of the pipeline.

In Figure 3, the IR-predictor is shown as a new component outside the processor core that overrides the conventional branch predictor. Alternatively, since the IR-predictor is built on top of a conventional branch predictor, the core's predictor and the IR-predictor may be integrated.

3.1.2. Improved IR-predictor: bypassing instruction fetch. With the base IR-predictor described in Section 3.1.1, the A-stream is not reduced in terms of the number of instructions *fetched*. Only the number of instructions *executed* is reduced. If execution bandwidth is relatively unconstrained, then the A-stream will not be effectively reduced.

The A-stream is more effective if *fewer fetch cycles* are expended on it than on the full program. In Figure 4, we show an example of how the number of fetch cycles can potentially be reduced. Four basic blocks, labeled *A* through *D*, are to be predicted and fetched. The corresponding table entries in the IR-predictor are shown; shaded entries indicate that all of the confidence counters are saturated and the entire basic block is predicted for removal. The base IR-predictor predicts each block in sequence, requiring four cycles. During two of these cycles, the instruction cache fetches instructions and then throws them all away (basic blocks *B* and *C*). Clearly, only two fetch cycles are required, but it is not known in advance that instruction fetching of blocks *B* and *C* can be bypassed.

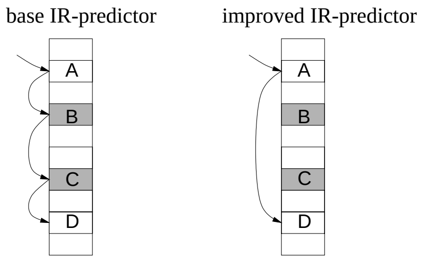

FIGURE 4. Reducing fetch cycles in the A-stream.

Interestingly, the effect we want to produce — bypassing basic blocks — is the same effect produced by taken branches. The improved IR-predictor shown on the right-hand side of Figure 4 exploits the analogy. The improved predictor "converts" the branch terminating block *A* into a taken branch whose target is block *D*. Below, we consider two possible ways to implement this conversion.

- Two additional pieces of information are stored in block *A*'s table entry. First, the predicted directions of any bypassed branches must be stored, in this case, the predicted directions of the branches in blocks *B* and *C*. The reason is all control flow information must be pushed onto the delay buffer to be consumed by the R-stream, in spite of partially bypassing instruction fetching in the A-stream. Second, a target address must be stored, in this case, the start PC of block *D*. The target address overrides the next PC computation performed by the fetch unit. The additional information (bypassed predictions and corresponding target address) is accumulated for block *A*'s entry as the IR-detector sequentially updates the entries of blocks *B*, *C*, and *D*.
- Effectively, the branch terminating block *A* is now a multi-way branch. It has more potential targets than its original taken and fall-through targets because it inherits the targets of skipped blocks. The processor's branch target buffer may be modified to store multiple targets per branch. Now, dynamically-created target addresses do not have to be stored in the IR-predictor.

The bypassed predictions still need to be stored and, conveniently, this path information is sufficient to select the appropriate target address from the branch target buffer.

3.1.3. IR-detector. The IR-detector consumes retired R-stream instructions, addresses, and values. The instructions are buffered and, based on data dependences, circuitry among the buffers is dynamically configured to establish connections from consumer to producer instructions. In other words, a reverse dataflow graph (R-DFG) is constructed. The graph is finite in size, so the oldest instructions exit the graph to make room for newer instructions. Removal information for exiting instructions are used to update the IR-predictor.

As new instructions are merged into the R-DFG, the IR-detector watches for any of three triggering conditions for instruction removal. Triggering conditions are unreferenced writes (a write followed by a write to the same location, with no intervening read), non-modifying writes [12,14,16,29] (writing the same value to a location as already exists at that location), and correctly-predicted 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. Predecessors may also be selected if certain criteria (described later) are met.

The IR-detector is shown in Figure 5. A single R-DFG is shown, however, the buffering could be partitioned into multiple smaller R-DFGs. The latter approach reduces the size/complexity of each individual R-DFG but still allows a large analysis scope for killing values (observing another write to the same location).

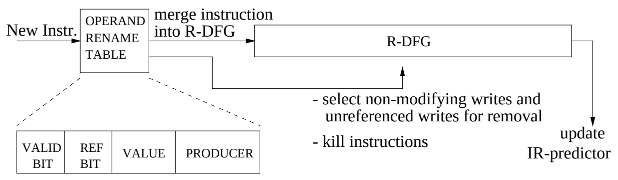

FIGURE 5. IR-detector.

The operand rename table in Figure 5 is similar to a register renamer but it can track both memory addresses and registers. A single entry of the operand rename table is shown in Figure 5. To merge an instruction into the 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 instruction uses this information to establish connections with its producer instructions, i.e., set up the back-propagation logic (if the buffering is partitioned into smaller R-DFGs, connections cannot be made across partition boundaries). The *ref bit* is set for each source operand indicating the values have been used. If the 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. No fields are updated in the rename table entry since the old producer remains "live" in this case.
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. Finally, all fields in the rename table entry are updated to reflect the new producer.

Correctly predicted branch instructions are selected for removal when they are merged into the R-DFG.

Finally, any other instruction *x* may be selected for removal via the R-DFG back-propagation circuitry, if three conditions are met.

1. All of *x*'s dependent instructions must be known, i.e., *x*'s production(s) must be killed by other production(s).
2. All of *x*'s dependent instructions must be selected for removal.
3. *All of x's dependent instructions must have been removed by the IR-predictor this time around.*

When a basic block becomes the oldest basic block in the analysis scope, the appropriate entry for that basic block is updated in the IR-predictor, i.e., confidence counters are incremented for selected instructions and reset for non-selected instructions.

The third (highlighted) condition above is the major innovation with respect to our previous instruction-removal mechanism. Previously, this constraint was not needed because dependence chains were confined to a trace and a single confidence counter was maintained for the entire trace; this ensured producers and consumers were removed together or not at all, but it also resulted in unrelated chains diluting overall confidence. The dilution problem is fixed by maintaining confidence for instructions individually; however, this can lead to partial-chain removal and the specifically bad situation of removing a producer but not the consumer. The third constraint above ensures a producer's counter saturates only after all consumers' counters saturate. The end result: 1) our new approach measures confidence for instructions individually, so unrelated instructions do not dilute confidence, yet 2) dependence chains still tend to be removed as a unit, and chains are not confined within a small region other than to reduce R-DFG complexity.

3.2. 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. 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 3 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 IR-predictor predicts all branches even though the A-stream may not fetch all instructions (Section 3.1.2).

The R-stream pops control and data flow information from the delay buffer. This is shown in Figure 3 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).

3.3. IR-misprediction 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.

IR-mispredictions are detectable by the R-stream because either the control or data flow outcomes from the delay buffer will not match its redundantly computed outcomes. In other words, IR-mispredictions usually surface as branch or value mispredictions in the R-stream.

Some IR-mispredictions take awhile to cause any visible symptoms in the A-stream. For example, a store may be removed incorrectly and the next load to the same location may not occur for a very long time. The IR-detector can detect these IR-mispredictions much sooner by comparing its computed removal information against the corresponding predicted removal information — 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.

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 IR-predictor is backed up to the precise program counter, the delay buffer is flushed, the reorder buffer of the A-stream is flushed, and the A-stream's program counter is 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 3.

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 3. 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.

- The recovery controller tracks addresses of stores retired in the A-stream but not yet retired in the R-stream. After detecting an IR-misprediction, these A-stream stores must be "undone" since the R-stream has not yet performed the companion, redundant store.
- 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. 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.

4. Simulation environment

We developed a detailed execution-driven simulator of a slipstream processor. The simulator faithfully models the architecture depicted in Figure 3 and outlined in Section 3: 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 [26]. The functional simulator checks retired R-stream control flow and data flow outcomes.

The Simplescalar [3] compiler and ISA are used. We use the SPEC95 integer benchmarks (-O3 optimization) run to completion (Table 1).

TABLE 1. Benchmarks.

benchmark	input dataset	instr. count
compress	40000 e 2231	124 million
gcc	cccp.i -o cccp.s	265 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	108 million
vortex	vortex.in (persons.250)	101 million

TABLE 2. Microarchitecture configuration.

single processor core	
instruction cache	size/assoc/repl = 64kB/4-way/LRU
	line size = 16 instructions
	2-way interleaved
	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: 64, 128, or 256 entries
	dispatch/issue/retire bandwidth: 4-/8-/16-way
	n fully-symmetric function units (*n* = issue b/w)
	n loads/stores per cycle (*n* = issue b/w)
execution latencies	address generation = 1 cycle
	memory access = 2 cycles (hit)
	integer ALU ops = 1 cycle
	complex ops = MIPS R10000 latencies
new components for slipstreaming	
IR-predictor	2^{20} entries *gshare*-indexed (16 bits of global branch history)
	block size = 16 16 confidence counters per entry
	confidence threshold = 32
IR-detector	R-DFG = 256 instructions, unpartitioned
delay buffer	data flow buffer: 256 instruction entries
	control flow buffer: 4K branch predictions
recovery controller	# of outstanding store addr. = unconstrained
	recovery latency (*after* IR-misp. detected): • 5 cycles to start up recovery pipeline • 4 reg. restores/cycle (64 regs performed 1st) • 4 mem. restores/cycle (mem performed 2nd) • ∴ min. latency (no memory) = 21 cycles

Microarchitecture parameters are listed in Table 2. The top half of the table lists parameters for individual processors within a CMP or, alternatively, a single SMT processor. The bottom half describes the four slipstream components. A large IR-predictor is used for accurate instruction removal. The removal confidence threshold is 32. The IR-detector has a scope of 256 instructions and the R-DFG is unpartitioned. The delay buffer stores 256 instructions (data flow buffer) and 4K branch predictions (control flow buffer). 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-misprediction is detected) is 5 cycles to startup the recovery pipeline, followed by 4 register restores per cycle, and lastly 4 memory restores per cycle.

5. Results

5.1. Slipstream performance results

In this section, we compare the performance of eight models. Three are superscalar configurations (SS). Four are chip-multiprocessor configurations (CMP) with slipstreaming. One is a simultaneous multithreading configuration (SMT) with slipstreaming.

- **SS(64x4)**: A single 4-way superscalar processor with 64 ROB entries.
- **SS(128x8)**: A single 8-way superscalar processor with 128 ROB entries.
- **SS(256x16)**: A single 16-way superscalar processor with 256 ROB entries.
- **CMP(2x64x4)**: Slipstreaming on a CMP composed of two SS(64x4) cores.
- **CMP(2x64x4)/byp**: Same as previous, but A-stream can bypass instruction fetching.
- **CMP(2x128x8)**: Slipstreaming on a CMP composed of two SS(128x8) cores.
- **CMP(2x128x8)/byp**: Same as previous, but A-stream can bypass instruction fetching.
- **SMT(128x8)/byp**: Slipstreaming on SMT, where the SMT is built on top of SS(128x8).

For consistent comparisons, the same (*gshare*-based) IR-predictor provides branch predictions in all of the processor models, and the base superscalar processor models ignore the instruction-removal information. Performance is measured in retired instructions-per-cycle (IPC). For slipstream models, IPC is computed as the number of retired R-stream instructions (i.e., the full program, counted only once) divided by the number of cycles required for both the A-stream and R-stream to complete (total execution time).

IPC performance of the eight models is shown in Figure 6. The first conclusion is a slipstream processor can exploit a second, otherwise unused processor to dramatically improve single-program performance. From Figure 7, CMP(2x64x4) performs on average 12% better than using only a single SS(64x4) processor. And CMP(2x128x8) performs on average 7% better than using only a single SS(128x8) processor. Slipstreaming degrades performance in *jpeg*, by 1% and 5% for CMP(2x64x4) and CMP(2x128x8), respectively. *Jpeg*'s A-stream is not reduced much and *jpeg* is already quite parallel; IR-mispredictions cause an overall degradation.

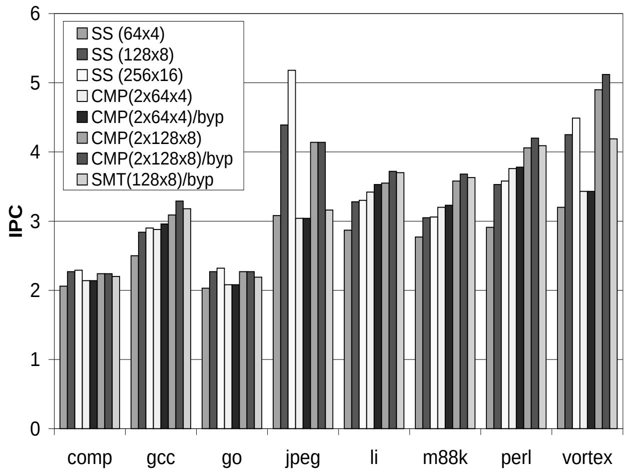

FIGURE 6. IPC results.

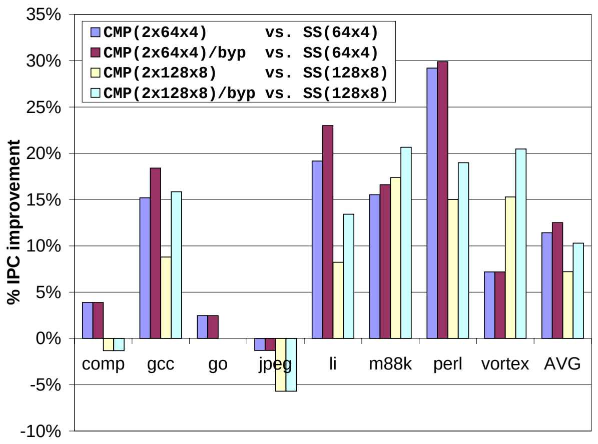

FIGURE 7. Performance improvement using a second processor for slipstreaming.

The second conclusion is the benefit of slipstreaming decreases as more execution bandwidth is made available. This is evident from the first and third bars of Figure 7. For all except *m88ksim* and *vortex*, the performance improvement of CMP(2x128x8) over SS(128x8) is less than the improvement of CMP(2x64x4) over SS(64x4). For example, *perl* drops from a 30% improvement down to a 15% improvement as the window size and issue bandwidth of the processor core is doubled. This is evidence for the arguments made in Section 2.3.

The above result motivates reducing the number of instructions *fetched* in the A-stream, using the improved IR-predictor (Section 3.1.2). From Figure 7, CMP(2x64x4)/byp on average performs 13% better than SS(64x4), a modest change from CMP(2x64x4). As expected, it is more important to bypass instruction fetching for larger processor cores. CMP(2x128x8)/byp on average performs 10% better than SS(128x8), whereas CMP(2x128x8) performs 7% better. With the improved IR-predictor, slipstream performance improvement increases from 8% to 16% for *gcc*, from 8% to 14% for *li*, from 17% to 21% for *m88ksim*, from 15% to 19% for *perl*, and from 15% to 20% for *vortex*.

In Figure 8, we compare the performance of slipstreaming on two small processors to the performance of a larger processor. The larger processor has the same total number of ROB entries and issue bandwidth as the two smaller processors combined. For half of the benchmarks (*perl*, *gcc*, *li*, *m88ksim*), CMP(2x64x4)/byp actually performs from 4% to 8% better than SS(128x8). Overall, CMP(2x64x4)/byp performs comparably to the more complex, less flexible SS(128x8) processor — within 5% on average. The results are more pronounced for CMP(2x128x8)/byp, which on average performs 7% better than SS(256x16).

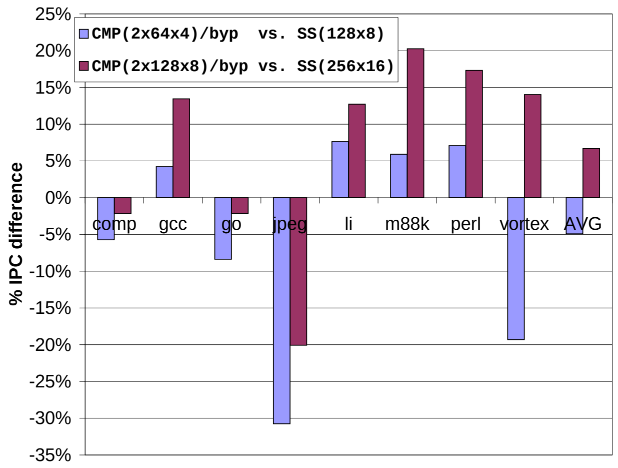

FIGURE 8. Perf. of slipstreaming on two small processors vs. perf. of a single large processor.

Finally, we examine the performance of slipstreaming on an SMT processor. The performance improvement of SMT(128x8)/byp over SS(128x8) is shown in Figure 9. For half of the benchmarks, performance improves by more than 10%. *Gcc*, *li*, *perl*, and *m88ksim* improve by 12%, 13%, 16%, and 19%, respectively. Performance is degraded between 1% and 4% for *compress*, *go*, and *vortex*, and over 25% for *jpeg*. *Compress* showed a small loss even for the CMP(2x128x8) model, so one would expect the same for SMT(128x8)/byp. The reason is the A-stream is less effective for *compress* and IR-mispredictions degrade performance. *Go* was also borderline in the

CMP(2x128x8) case. *Vortex* and *jpeg* utilize the SS(128x8) processor well — in fact, they exceed half of the peak IPC — and the A-stream steals useful processor bandwidth from the R-stream. The effect is more pronounced for *jpeg* than for *vortex* because *jpeg* exhibits little reduction in its A-stream (Figure 10).

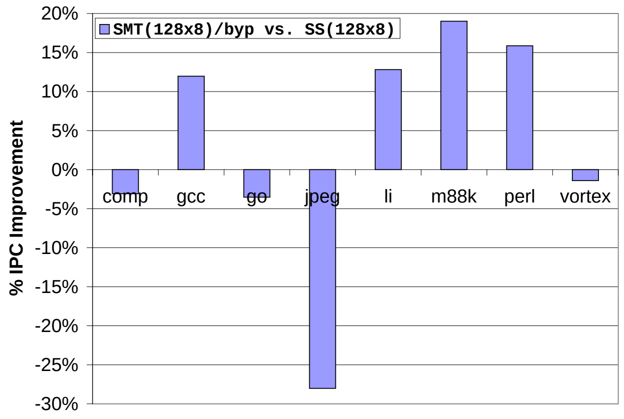

FIGURE 9. Performance improvement of SMT(128x8)/byp over SS(128x8).

5.2. Instruction removal

Figure 10 shows the fraction of original dynamic instructions removed from the A-stream. Nearly half of the program is removed for *gcc*, *li*, *perl*, and *vortex*, and about two-thirds of *m88ksim* is removed. About 20% of *compress* is removed, and only 10% for *go* and *jpeg*.

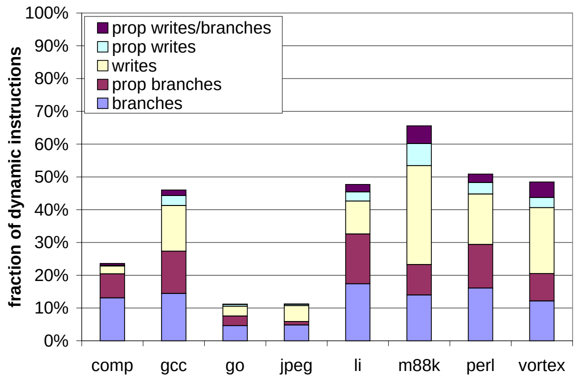

FIGURE 10. Breakdown of instruction removal.

Removing only 10% of the program simply does not buffer the R-stream from many branch mispredictions. But 20% removal in *compress* is significant, and it is surprising slipstream performance improvements are not higher. The problem with *compress* is three-fold: there are frequent branch mispredictions, their dependence chains are quite long, and the chains have long-latency arithmetic operations. Removing 20% of *compress* can perhaps buffer the R-stream against any one of these three, but not two or three combined.

Figure 10 also breaks down the reasons for instruction removal. On average, branches are the primary source, at just over a third of the removed instructions (“branches”). Ineffectual writes are about a third of removed instructions (“writes”). Among instructions removed due to back-propagation (“prop —”), most are in dependence chains of removed branches (“prop branches”).

5.3. Prediction

In Figure 11, we show the performance improvement of three models with respect to SS(64x4). The first is SS(64x4) with conventional value prediction added. A large context-based value predictor (CVP) [24] is used (2^{18} and 2^{20} entries in the first and second levels, respectively). The second is CMP(2x64x4)/byp, but the R-stream does not use A-stream values speculatively (“no value prediction”). The third is CMP(2x64x4)/byp.

We only consider benchmarks that show reasonably large improvements with any of the models (eliminating *compress*, *go*, *jpeg*). For *gcc* and *li*, better branch prediction is the largest benefit due to slipstreaming, not value prediction (we can tell because the second and third bars are close). Also, CVP provides only minor improvements for these benchmarks. For *m88ksim*, value prediction is the dominant factor and CVP is superior. For *perl* and *vortex*, value prediction is the larger benefit due to slipstreaming, however, CVP does not provide the same benefit. Perhaps in *perl*, better branch prediction is needed to better exploit value predictions.

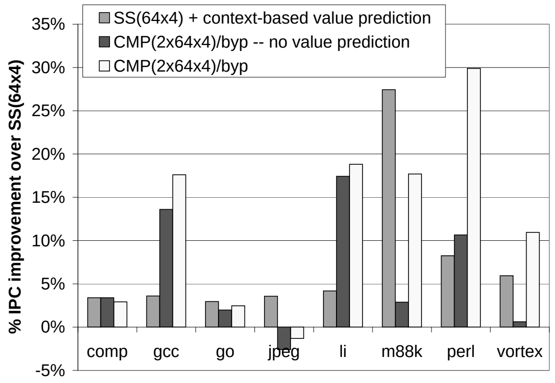

FIGURE 11. Measuring the relative importance of branch and value prediction benefits.

6. Related work

Advanced-stream/Redundant-stream Simultaneous Multithreading (AR-SMT) [20] 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 [28] 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 [13] — 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 [18] 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 [6,8,9,12,13,14,16,24,25,29]. This prior research forms a basis for creating the shorter program in slipstream processors. A technical report [21] 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. [28] and Yamamoto and Nemirovsky [31] proposed simultaneous multithreading for flexibly exploiting thread-level and instruction-level parallelism. Olukotun et. al. [17] motivate using chip multiprocessors.

Farcy et. al. [7] proposed resolving branch mispredictions early by extracting the computation leading to branches. Zilles and Sohi [33] 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* [22,23], 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 [22]. Roth and Sohi [23] developed a new paradigm called *Speculative Data-Driven Multithreading* that implements pre-execution generally. Rather than spawn many specialized kernels on-the-fly, our approach uses a single, functionally complete, and persistent program (A-stream). Slipstreaming avoids the conceptual and possibly real complexity of forking private contexts, within which the specialized kernels must run.

Speculative multithreading architectures [e.g.,1,17,26] 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.

SSMT [5] runs microthreads simultaneously with an application to optimize its performance. Microthreads are small routines designed in conjunction with applications and the processor. For example, microthreads may perform cache prefetching, improve branch prediction accuracy [5], or optimize exception handling [32].

The DataScalar paradigm [4] runs redundant programs on multiple processor-and-memory cores to eliminate memory read requests.

7. Summary and conclusions

Integrating multiple architectural contexts on a single chip is an important trend, and it is difficult to conceive of more effective uses for a billion transistors. The slipstream paradigm extracts more functionality from a CMP or SMT processor, without fundamentally reorganizing it. The operating system may flexibly choose among multiple operating modes based on system and user requirements: high job throughput and parallel-program performance (conventional SMT/CMP), improved single-program performance and reliability (slipstreaming), or fully-reliable operation with low impact on single-program performance (AR-SMT / SRT).

In this paper, we developed a new and more effective instruction-removal mechanism for creating the shorter program. It measures removal-confidence on a per-instruction basis, eliminating many flaws of the prior trace-based approach and leveraging conventional branch predictors. The new approach reduces the A-stream significantly (often by 50%), but also accurately.

We also developed a new and simple sequencing mechanism that enables the A-stream to skip over large dynamic sequences of instructions.

Finally, we reasoned about the sources of slipstream performance, and its limitations. This focused our exploration of the architecture and led us to some key results.

- A 12% average performance improvement is achieved by harnessing an otherwise unused, additional processor in a CMP. Slipstreaming using two small superscalar cores often achieves similar IPC as one large superscalar core, but with a potentially faster clock and a more flexible architecture. For programs with sufficiently reduced A-streams, slipstreaming on an 8-way SMT processor improves performance from 10%-20%.
- For some programs, performance improvement is due to the A-stream resolving branch mispredictions in advance. Others benefit largely from A-stream value predictions, and the effect is not always reproducible using conventional value prediction tables.
- As more execution bandwidth is made available, slipstreaming provides less performance improvement. But if the A-stream is able to bypass instruction fetching, slipstreaming retains its edge — because raw instruction fetch bandwidth is not as easily extended as raw execution bandwidth.

References

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

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

[3] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The Simplescalar Toolset. Tech. Rep. CS-TR-96-1308, CS Dept., Univ. of Wisconsin, July 1996.

[4] D. Burger, S. Kaxiras, and J. Goodman. DataScalar Architectures. *24th Int'l Symp. on Comp. Arch.*, June 1997.

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

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

[7] 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 Microarch.*, Dec. 1998.

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

[9] J. Huang and D. Lilja. Exploiting Basic Block Value Locality with Block Reuse. *5th Int'l Symp. on High-Perf. Comp. Arch.*, Jan. 1999.

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

[11] M. Lam and R. Wilson. Limits of Control Flow on Parallelism. *19th Int'l Symp. on Comp. Arch.*, May 1992.

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

[13] M. Lipasti. Value Locality and Speculative Execution. PhD Thesis, Carnegie Mellon University, April 1997.

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

[15] S. McFarling. Combining Branch Predictors. Tech. Rep. TN-36, WRL, June 1993.

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

[17] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. *ASPLOS-VII*, Oct. 1996.

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

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

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

[21] E. Rotenberg. Exploiting Large Ineffectual Instruction Sequences. Tech. Rep., ECE Dept., NC State, Nov. 1999.

[22] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. *ASPLOS-VIII*, Oct. 1998.

[23] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. Tech. Rep. CS-TR-2000-1414, CS Dept., Univ. of Wisconsin, April 2000.

[24] Y. Sazeides and J. E. Smith. Modeling Program Predictability. *25th Int'l Symp. on Comp. Arch.*, June 1998.

[25] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. *24th Int'l Symp. on Comp. Arch.*, June 1997.

[26] G. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. *22nd Intl. Symp. on Comp. Arch.*, June 1995.

[27] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. *ASPLOS-IX*, Nov. 2000.

[28] 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 Comp. Arch.*, May 1996.

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

[30] D. Wall. Limits of Instructional-Level Parallelism. *ASPLOS-IV*, April 1991.

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

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

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