A Simple Mechanism for Detecting Ineffectual Instructions in Slipstream Processors

Jinson J. Koppanalil and Eric Rotenberg, *Member*, *IEEE*

Abstract—A slipstream processor accelerates a program by speculatively removing repeatedly ineffectual instructions. Detecting the roots of ineffectual computation—unreferenced writes, nonmodifying writes, and correctly predicted branches—is straightforward. On the other hand, detecting ineffectual instructions in the backward slices of these root instructions currently requires complex back-propagation circuitry. We observe that, by logically monitoring the speculative program (instead of the original program), back-propagation can be reduced to detecting unreferenced writes. That is, once root instructions are actually removed, instructions at the next higher level in the backward slice become newly exposed unreferenced writes in the speculative program. This new algorithm, called implicit back-propagation, eliminates complex hardware and achieves an average performance improvement of 11.8 percent, only marginally lower than the 12.3 percent improvement achieved with explicit back-propagation. We further simplify the hardware component by electing not to detect ineffectual memory writes, focusing only on ineffectual register writes. A minimal implementation consisting of only a register-indexed table (similar to an architectural register file) achieves a good balance between complexity and performance (11.2 percent average performance improvement with implicit back-propagation and without detection of ineffectual memory writes).

Index Terms—Microarchitecture, multithreading, chip multiprocessor, slipstream, preexecution.

1 Introduction

With multiple processor cores on a single chip, a *chip multiprocessor* (CMP) [5], [10] delivers high throughput for multiprogrammed workloads and multithreaded/parallel programs. Each constituent processor core also delivers high single-program performance with a balanced design that emphasizes both instruction-level parallelism (ILP) and a fast clock. However, a single program cannot take advantage of more than one core, even if other cores are idle, which is often the case in the desktop environment [21].

The *slipstream paradigm* enables a second core in a CMP to be used for enhancing single-program performance. A slipstream processor [11], [16] runs two redundant copies of a program on a CMP substrate, one slightly ahead of the other. A significant number of ineffectual instructions are speculatively removed from the leading program, called the *advanced stream* (A-stream). Ineffectual instructions are instructions that are not essential for correct forward progress. The A-stream is sped up because it fetches and executes fewer instructions than the original program. The trailing program, called the *redundant stream* (R-stream), checks the control-flow and data-flow outcomes of the A-stream and redirects it when it fails to make correct forward progress (a rare event). The R-stream also exploits the outcomes from the A-stream as accurate branch and value predictions. Thus, although the R-stream retires the same number of instructions as the original program, it fetches and executes much more efficiently. As a result, both program copies finish sooner than the original program.

Many general-purpose programs contain a significant number of ineffectual instructions [13]. Some dynamic instructions produce results that are not referenced by subsequent instructions. Other dynamic instructions do not modify the value of a location. Finally, there are instructions whose outcomes are highly predictable, for example, branch instructions. These ineffectual instructions—unreferenced writes, nonmodifying writes, and correctly predicted branches, respectively—can be removed without affecting the correct forward progress of the program. Once they are removed, the dependence chains that lead up to them can also be removed.

A slipstream component called the *instruction-removal detector* (IR-detector) detects past-ineffectual instructions in the R-stream and selects them for possible removal from the A-stream in the future. The IR-detector uses a two-step selection process. First, it selects key trigger instructions—unreferenced writes, nonmodifying writes, and correctly predicted branches. A table similar to a conventional register rename table can easily detect unreferenced and nonmodifying writes. The second step, called back-propagation, selects computation chains feeding the trigger instructions. In an explicit implementation of back-propagation, retired R-stream instructions are buffered and consumer instructions are physically connected to their producer instructions. Consumers that are selected because they are ineffectual use these connections to propagate their ineffectual status to their producers so that they get selected, too. Explicit back-propagation is impractical.

This paper describes a practical implementation of back-propagation, called *implicit back-propagation*. The key idea is

- *J.J. Koppanalil is with ARM, Inc., Bldg. 3, Suite 560, 1250 Capital of Texas Highway, Austin, TX 78746. E-mail: jinson.koppanalil@arm.com.*
- *E. Rotenberg is with the Department of Electrical and Computer Engineering, North Carolina State University, Partners Bldg. I, Suite 2300, Campus Box 7256, Raleigh, NC 27695-7256. E-mail: ericro@ece.ncsu.edu.*

Manuscript received 7 Jan. 2003; revised 10 July 2003; accepted 15 Aug. 2003. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 118096.

0018-9340/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

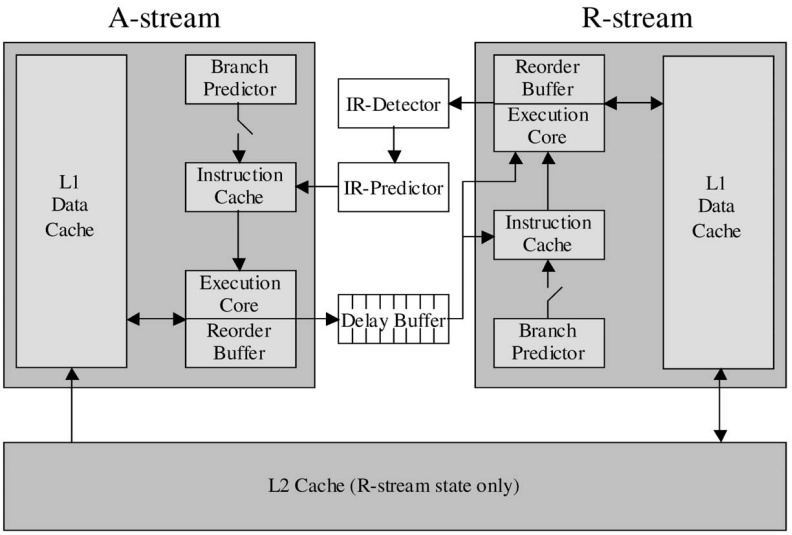


Fig. 1. Slipstream processor using a dual-processor CMP [11], [12], [16].

to *logically* monitor the A-stream instead of the R-stream. Now, the IR-detector only performs the first step, i.e., it selects unreferenced writes, nonmodifying writes, and correctly predicted branches. After building up confidence, these trigger instructions are removed from the A-stream. Once removed, their producers become unreferenced writes in the A-stream (because they no longer have consumers). The freshly exposed unreferenced writes are selected by the IR-detector and, after building up confidence, are removed themselves. Removing them, in turn, exposes more unreferenced writes and the process continues iteratively until, eventually, entire dependence chains are removed.

By logically monitoring the A-stream, back-propagation is reduced to detecting unreferenced writes. Implicit back-propagation eliminates complex hardware and achieves an average performance improvement close to that of explicit back-propagation (11.8 percent for implicit versus 12.3 percent for explicit).

The modified IR-detector is simplified further by not supporting removal of store instructions. Ineffectual stores [13] include silent stores [6], unreferenced stores, and stores in the backward slices of loads which are removed. Removal of stores requires a cache-like structure to track load and store references to memory locations. On average, performance improvement of slipstream without store removal is close to performance improvement with store removal (11.2 percent versus 11.8 percent, respectively), although the impact on individual benchmarks varies. For two of the benchmarks, the decrease in performance improvement is more substantial due to a significant drop in overall instruction removal. However, slipstream still yields large performance improvements in these benchmarks. In two other benchmarks, the performance improvement of slipstream actually increases without store removal, due to more A-stream-supplied value predictions and fewer A-stream deviations. Overall, the results support a minimal IR-detector design based on implicit back-propagation and without store-removal capability.

This paper also characterizes circuit complexity of the IR-detector based on implicit back-propagation. A minimal IR-detector design consists primarily of a table similar to a register rename table, indexed by logical registers. Read and write ports for accessing various fields are analyzed in detail. The table is shown to be no more complex than the register rename table.

Finally, this paper explores the design space for the IR-detector, which includes the confidence counter threshold (the number of consecutive times that an instruction must be removable before actually removing it in the A-stream) and instruction buffer size (the size of a FIFO that contains the ineffectual/effectual status of retired R-stream instructions). Simulations with the SPEC95 and SPEC2K benchmarks indicate that the best instruction buffer size is 128 and the best confidence counter threshold is 64 for both the explicit and implicit IR-detectors.

Section 2 briefly reviews the slipstream microarchitecture. Section 3 describes the IR-detector employing implicit back-propagation. The training time of IR-detectors based on implicit and explicit back-propagation is discussed in Section 4 to provide a context for explaining experimental results. Section 5 discusses hardware requirements for store removal, if supported. Section 6 assesses circuit complexity of the IR-detector based on implicit back-propagation. Simulation methodology and benchmarks are described in Section 7. Section 8 presents experimental results. Related work is presented in Section 9, focusing on hardware techniques for program data-flow analysis. Section 10 summarizes the paper.

2 Review of Slipstream Microarchitecture

This section briefly reviews slipstream processors [11], [12], [13], [16]. A high-level block diagram of a slipstream processor implemented on a dual-processor chip-multiprocessor (CMP) is shown in Fig. 1. The A-stream is shown on the left and the R-stream is shown on the right. The shaded blocks show the two processor cores and the shared L2 cache, which constitute the original CMP. Each processor core is a conventional superscalar processor with a branch

predictor, instruction and data caches, and execution engine, including register file and reorder buffer.

The slipstream microarchitecture requires new components for facilitating instruction removal in the A-stream (IR-detector and IR-predictor) and for communicating A-stream outcomes to the R-stream (delay buffer). In addition, separate architectural contexts must be managed for the A-stream and R-stream and a recovery mechanism is needed to repair the A-stream state when it deviates. These aspects are described in the following subsections.

2.1 Creating a Reduced A-Stream

The *instruction-removal detector* (IR-detector) and *instruction-removal predictor* (IR-predictor) cooperate to create a reduced A-stream. The IR-detector identifies retired R-stream instructions that, in hindsight, were not needed for correct forward progress. The IR-detector then conveys to the IR-predictor that these instructions can potentially be skipped in the A-stream, in the future. The IR-predictor removes the corresponding instructions from the A-stream after repeated indications by the IR-detector, i.e., after a certain confidence threshold has been reached.

The IR-predictor is a branch predictor augmented for instruction removal. It generates the program counter (PC) for the next block of instructions to be fetched in the A-stream, like a conventional branch predictor. The only difference is that the generated PC may reflect skipping entire, predicted-ineffectual basic blocks. For basic blocks that are not entirely ineffectual, the IR-predictor also specifies a bit vector. The bit vector indicates which instructions within the fetched block are ineffectual. These instructions are removed from the fetched block before the decode stage.

The IR-predictor is indexed the same way as the *gshare* branch predictor [8] (XOR the PC and the global branch history register). Each entry in the IR-predictor corresponds to a single dynamic basic block. An entry contains resetting confidence counters [4] for each instruction in the basic block. These counters are the interface between the IR-detector and the IR-predictor. The counter for a dynamic instruction is incremented when the IR-detector detects that it is ineffectual. The counter is reset to zero when the IR-detector detects that it is effectual. If the counter is saturated, the IR-predictor is authorized to remove the instruction from the A-stream when it is next encountered.

2.2 Communicating A-Stream Outcomes to the R-Stream

The delay buffer is a FIFO queue used to communicate control-flow and data-flow outcomes from the A-stream to the R-stream. Control-flow history is complete because the IR-predictor predicts all branches, even though it may direct the A-stream fetch unit to skip instructions. This implies that only a portion of the control-flow conveyed via the delay buffer is verified by the A-stream (in particular, the unconfident portion of control-flow is verified by A-stream computation since it cannot be confidently removed [11]). Data-flow history is incomplete since the A-stream executes only a subset of the program.

The R-stream uses the control-flow and data-flow outcomes from the delay buffer as predictions. Control-flow outcomes are used to direct instruction fetching from the instruction cache. Data-flow outcomes are used as value predictions in the execution core. To bind values to corresponding R-stream instructions, the delay buffer also contains one bit per dynamic instruction that indicates whether or not the corresponding instruction was skipped in the A-stream.

2.3 Memory Management

The A-stream and R-stream are architecturally independent. A-stream loads and stores should not interfere with R-stream loads and stores. The simplest way to take care of this aspect is to have the operating system (O/S) allocate separate physical memory pages for each program. However, *software-based memory duplication* doubles memory usage.

Therefore, more recently, *hardware-based memory duplication* is used in slipstream processors [12]. It provides three key advantages. First, slipstream memory usage is bounded by the usage of a single copy of the program. Second, separating the A-stream state and R-stream state is as simple as before, while being transparent to the O/S. Third, recovery of A-stream memory state is greatly simplified, involving simple cache invalidation.

The approach exploits the typical memory hierarchy found in commercial dual-processor CMPs [5]. The memory hierarchy consists of private L1 caches for each of the processing elements and a shared L2 cache, as shown in Fig. 1. The approach works as follows:

- Both the A-stream and R-stream read and write their respective L1 caches normally.
- The R-stream L1 cache is write-through, i.e., if the R-stream performs a store in its L1 cache, it also performs the store in the shared L2 cache.
- The A-stream L1 cache is neither write-through nor write-back, i.e., A-stream stores are not propagated to the shared L2 cache. If a dirty line (a line modified by the A-stream) needs to be evicted from the A-stream L1 cache, it is not written back to the shared L2 cache. The dirty line is simply thrown out and the updated data it contains is lost. Notice, in Fig. 1, the R-stream reads and writes the L2 cache, but the A-stream only reads from it.

The temporary loss of A-stream updates is not a major issue. The R-stream is usually close behind the A-stream and the lost A-stream data is regenerated by the R-stream and propagated to the L2 cache before it is rereferenced by the A-stream. Occasionally, the A-stream rereferences an evicted line in the L2 cache before the R-stream has performed the corresponding redundant store. In this case, the A-stream gets stale data and diverges from the R-stream. All deviations are detectable as branch or value mispredictions in the R-stream (see Section 2.4 below), whether caused by incorrect instruction removal or references to stale data (i.e., there is no need to diagnose the cause of the divergence).

2.4 IR-Mispredictions and A-Stream Recovery

An *instruction-removal misprediction*, or IR-misprediction, occurs when A-stream instructions were removed that should not have been. IR-mispredictions cause the A-stream

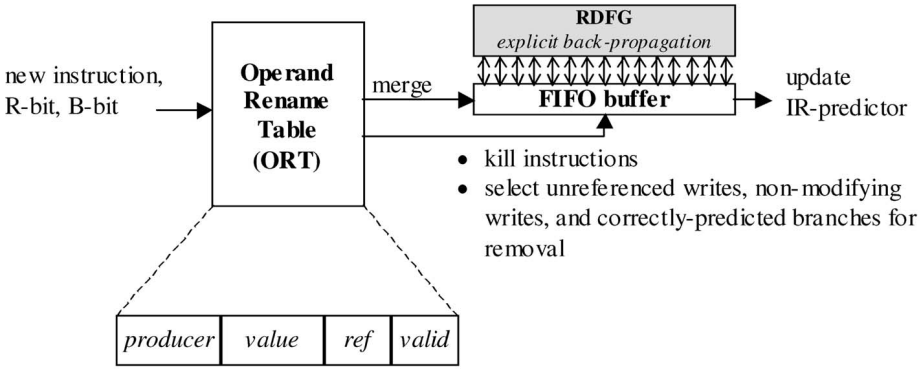


Fig. 2. Original IR-detector based on explicit back-propagation [11].

to not make correct forward progress and this is ultimately detected as branch or value mispredictions in the R-stream. When an IR-misprediction is detected, the A-stream needs to be resynchronized with the R-stream. This involves restoring A-stream register and memory state from R-stream registers and memory. R-stream register values may be transferred to the A-stream register file via the delay buffer or via shared-memory loads and stores initiated by light-weight exception handlers executed on both cores. A-stream memory is resynchronized with R-stream memory simply by invalidating the contents of the A-stream L1 cache. Several optimizations have been developed for reducing the impact of A-stream compulsory misses following recovery. These include invalidating only dirty lines (trading thoroughness for efficiency) and using invalidated data as highly reliable value predictions in the A-stream. Details can be found in a related technical report [12].

3 IR-Detector Implementation

The original slipstream IR-detector based on explicit back-propagation [11] is shown in Fig. 2. Retired R-stream instructions are initially processed by the operand rename table (ORT) to detect unreferenced writes and nonmodifying writes. The instructions are then held in a FIFO buffer. The oldest instructions in the buffer are evicted as new instructions are added. The final status (ineffectual/effectual) of exiting instructions is used to update the IR-predictor.

The ORT is similar to a conventional register rename table. It has as many entries as the number of logical registers. An entry contains information about the most recent write to the corresponding logical register. The *producer* field contains the FIFO entry number of the instruction that most recently wrote the register. The *value* field contains the value written by the producer. The *ref* bit indicates whether or not any later instruction has referenced the register. The *valid* bit indicates whether or not the producer is still in the FIFO. The ORT is managed as follows to detect unreferenced and nonmodifying writes. Below, source and destination registers refer to operands of the incoming instruction.

Step 1: The *ref* bits of source registers are set, indicating that these registers have been referenced.

Step 2: The *value* field of the destination register is compared with the value produced by the incoming instruction. If they match, the incoming instruction is a nonmodifying write and it is selected for removal as it is entered into the FIFO. If the values do not match, then the *ref* bit of the destination register is examined. If the *ref* bit is not set, then the previous producer is an unreferenced write. The previous producer is selected for removal within the FIFO, but only if it is still in the FIFO as indicated by the *valid* bit. Its location in the FIFO is indicated by the *producer* field.

Step 3: The various fields of the destination register are updated. The *producer* field is updated with the FIFO entry number of the incoming instruction. The *value* field is updated with the value of the incoming instruction. The *ref* bit is reset, indicating that there are no references yet. Finally, the *valid* bit is set, indicating that the new producer is in the FIFO. However, the various fields of the destination register are *not* updated if the incoming instruction is a nonmodifying write, as determined in Step 2 above. The nonmodifying write is selected for removal, which means the previous producer is still "live."

ORT entries whose producers exit the FIFO must be invalidated. A rotating pointer points to the oldest instruction in the FIFO. The oldest instruction is evicted to make room for the next incoming instruction. The rotating pointer is broadcast to all entries in the ORT. Each ORT entry has a comparator to compare the pointer to its *producer* field. If they match, then the *valid* bit is reset.

Correctly predicted branches are selected for removal as they are entered into the FIFO. Correctly predicted branches are flagged in the R-stream with the B-bit, as shown in Fig. 2.

So far, we have explained how trigger instructions are selected for removal by the ORT. The shaded component in Fig. 2 facilitates automatic selection of instructions in the backward slices of trigger instructions, via explicit back-propagation. This component consists of wires and logic connecting every instruction with every other instruction. The producer information available in the ORT is used to configure the logic in such a way as to link consumer instructions with their producer instructions as they are merged into the FIFO. Thus, this network of wires and logic is a hardware implementation of a *reverse data-flow graph* (RDFG). Initially, all consumers of a producer assert their links to the producer. One by one, consumers may deassert their links as they are selected for removal. When the links of all consumers are deasserted, the producer selects itself

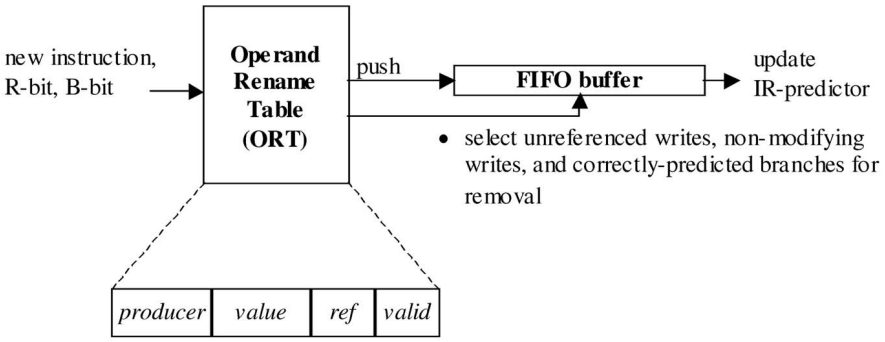


Fig. 3. New IR-detector based on implicit back-propagation.

for removal as well (assuming it has been killed, as signaled by the ORT). The producer, in turn, deasserts links to its producers and so on.

We defer presenting a detailed implementation of the RDFG, which is available in the master's thesis on which this paper is based [19]. Instead, we shift focus to the new IR-detector. It uses *implicit back-propagation*, which makes the RDFG obsolete.

The new IR-detector is shown in Fig. 3. The RDFG component has been eliminated. As before, there is a FIFO buffer, but its contents are trivial. The buffer consists of only a single bit per instruction indicating whether the instruction has been selected for removal by the ORT.

Earlier, we implied that the new IR-detector monitors retired A-stream instructions instead of R-stream instructions, thereby reducing back-propagation to the detection of unreferenced writes. In fact, retired R-stream instructions are monitored as before. However, a flag associated with each incoming instruction, called the R-bit, is used to "extract" the A-stream from the R-stream. The R-bit of the incoming instruction indicates whether or not it was actually removed from the A-stream by the IR-predictor this time around. The three steps described above are still used to manage the ORT, with one crucial modification to Step 1: An instruction whose R-bit is set does not set the *ref* bits of its source operands in the ORT. The instruction is not part of the A-stream and should be hidden from its producers in the ORT. This may cause its producers to be selected for removal as unreferenced writes, according to Step 2. Eventually, after reaching the confidence threshold, the producer instruction itself will be removed from the A-stream by the IR-predictor. The next instance of the producer instruction, when it is brought into the IR-detector, will make itself invisible to its producer because its R-bit is set and implicit back-propagation continues. Implicit back-propagation effectively converts the detection of an ineffectual computation chain into the detection of a sequence of unreferenced writes.

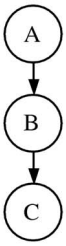
An example of implicit back-propagation is shown in Fig. 4. Instructions A, B, and C form a dependence chain and they write to registers R_A, R_B, and R_C, respectively. Instruction C is an unreferenced write and the only consumer of B. Instruction B is the only consumer of A. The sequence of events is as follows. First, C is selected for removal by the ORT since it is an unreferenced write. It is eventually removed from the A-stream after its confidence counter saturates (counter reaches a value of 31). The next time, when the IR-detector analyzes the sequence A, B, and C, it finds that the R-bit associated with instruction C is set (which says that the IR-predictor skipped C in the A-stream). During the processing of instruction C by the ORT, the *ref* bit corresponding to register R_B is not set, i.e., instruction C is not added as a consumer of instruction B, speculatively. This assumes that instruction C will be selected for removal by the IR-detector even this time. When register R_B is killed by some later instruction, the ORT finds that the register is not referenced by any other instruction and it selects instruction B for removal as an unreferenced write. Eventually, instruction B is also removed from the A-stream after its counter saturates. The next time the sequence A, B, and C is analyzed by the IR-detector, it finds that the R-bits of instructions B and C are set. Instruction C is not added as a consumer of B and instruction B is not added as a consumer of A. This assumes that B (and C) will be selected for removal by the IR-detector even this time. Now, instruction A is selected for removal as an unreferenced write when R_A is killed by a later instruction. Eventually, instruction A is also removed from the A-stream after its counter saturates.

4 Training Time for Building and Dismantling Chains

This section compares explicit and implicit back-propagation, in terms of training time for building up and dismantling chains of predicted-ineffectual instructions. This analysis is useful for understanding similarities and differences in the amount of dynamic instruction removal and rate of IR-mispredictions in the experimental results section.

An example of building up and dismantling a chain of predicted-ineffectual instructions is shown in Fig. 5, for implicit back-propagation. The IR-predictor uses a confidence counter threshold of 31 (corresponding to a 5-bit resetting counter). Instruction C is repeatedly ineffectual. It takes 32 passes through the IR-detector for its counter to saturate, at which time it is finally removed from the A-stream. When it is removed from the A-stream, instruction C's R-bit is set, which causes it to be transparent to instruction B, its producer. Instruction B is repeatedly ineffectual as a result. It takes another 32 passes through the IR-detector for instruction B's counter to saturate, at which time it is finally removed from the A-stream. At this point,

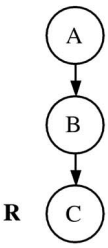
Stage 1. Instruction C is an unreferenced write.



ORT

	producer	value	ref	valid
R_A	A	V_A	1	1
R_B	B	V_B	1	1
R_C	C	V_C	0	1

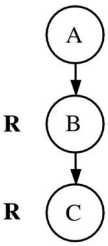
Stage 2. B and C are unreferenced writes. C is invisible to B (because its R-bit is set).



ORT

	producer	value	ref	valid
R_A	A	V_A	1	1
R_B	B	V_B	0	1
R_C	C	V_C	0	1

Stage 3. A, B, and C are unreferenced writes. C is invisible to B, and B is invisible to A.



ORT

	producer	value	ref	valid
R_A	A	V_A	0	1
R_B	B	V_B	0	1
R_C	C	V_C	0	1

Fig. 4. ORT during the stages of implicit back-propagation. An instruction whose R-bit is set (R) was removed from the A-stream by the IR-predictor.

instruction B becomes transparent to instruction A, making instruction A repeatedly ineffectual. It takes another 32 passes through the IR-detector for instruction A's counter to saturate, at which time it is finally removed from the A-stream.

To summarize, counters are built up one at a time in the case of implicit back-propagation. However, explicit back-propagation also incurs a lengthy training time, although we did not dwell upon this aspect in Section 3. As described at length in earlier slipstream papers [11], [19], a consumer deasserts the link to its producer if it is ineffectual *and* if it was removed by the IR-predictor this time around—as indicated by the R-bit. The latter back-propagation criterion prevents recurring IR-mispredictions, caused by producer and consumer counters getting "out-of-sync." An example of this condition is shown in Fig. 6. Producer A has two

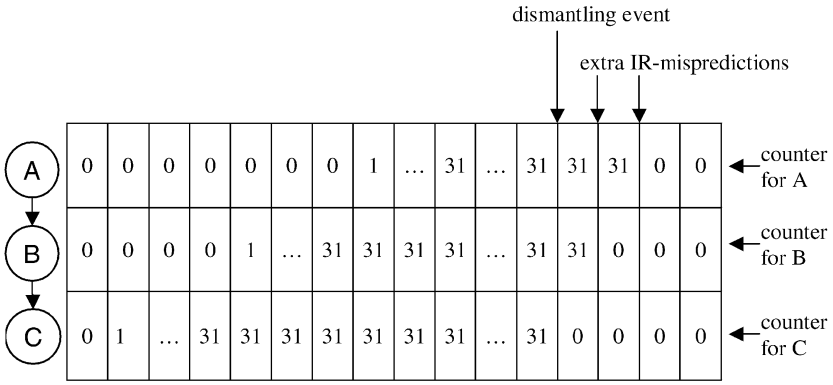


Fig. 5. Training time for implicit back-propagation. Counters are built up one at a time. Counters are reset sequentially, which causes extra IR-mispredictions for a single dismantling event.

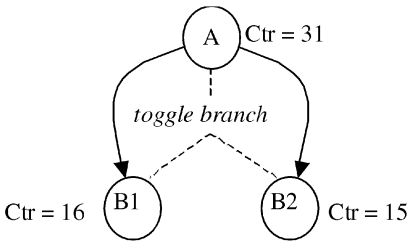


Fig. 6. Example of a producer counter saturating before its consumer counters if explicit back-propagation is based only on ineffectualness. This is prevented by also considering actual removal of consumers by the IR-predictor.

consumers, B1 and B2, on alternate control-flow paths. B1 and B2 are always ineffectual. The branch between producer A and its two consumers is a toggling branch (taken, not-taken, taken, not-taken, etc.). If B1 and B2 deassert their links to A solely on the basis of their ineffectualness, then A's counter will saturate before either of B1's and B2's counters saturate. This causes A to be incorrectly removed before its consumers are removed, causing an IR-misprediction. In fact, it causes recurring IR-mispredictions until B1's and B2's counters finally saturate. The key observation is that instruction A is removable if its consumers are *actually* removed, not just if they are hypothetically removable. This means even explicit back-propagation must use R-bit information, which in turn results in a lengthy training time for building up chains of predicted-ineffectual instructions. This is confirmed by experimentation in Section 8: Explicit and implicit back-propagation remove almost exactly the same number of dynamic instructions from the A-stream.

Chains of predicted-ineffectual instructions are dismantled when the trigger instruction becomes effectual or when its counter or the counter of another instruction in the chain is evicted from the IR-predictor, forcing another training sequence. Dismantling a chain requires multiple passes through the IR-detector in the case of implicit back-propagation, as shown in the latter portion of Fig. 5. We begin with the IR-detector pass that discovers instruction C is effectual. When C is brought into the IR-detector, its R-bit is set because the IR-predictor incorrectly removed it even though it is effectual this time around. The IR-detector prematurely makes instruction C transparent to instruction B in the ORT before finding out that instruction C is effectual. Thus, although instruction C's counter gets reset during this pass, the counter of instruction B remains saturated. This leads to an extra IR-misprediction next time because the IR-predictor will remove instructions A and B without also removing instruction C. The next time the IR-detector sees instructions A, B, and C, the R-bit of instruction C is not set (not removed by IR-predictor) and the R-bit of instruction B is set (still removed by IR-predictor). Instruction C is made visible to instruction B, so instruction B's counter gets reset during this pass. However, instruction B is not made visible to instruction A during this pass, so its counter remains saturated. This leads to yet another IR-misprediction because the IR-predictor will remove instruction A without also removing instructions B and C. A final pass resets instruction A's counter because instruction B's R-bit is not set, finally making it visible to instruction A. In general, the number of extra IR-mispredictions introduced by a single dismantling event is equal to the length of the dependence chain, less one (for the instruction that started the dismantling process).

On the other hand, explicit back-propagation does not incur these extra IR-mispredictions because the RDFG dismantles an entire dependence chain in a single pass through the IR-detector. This is confirmed by experimentation in Section 8: The rate of IR-mispredictions is higher for the IR-detector based on implicit back-propagation than the one based on explicit back-propagation.

5 Store Removal

Section 3 only described the operand rename table (ORT) for tracking register references. There is also a separate ORT for tracking memory references, which is needed to detect ineffectual stores. Entries in the memory ORT are the same as entries in the register ORT. The same algorithm is used to detect unreferenced and nonmodifying writes (stores), except loads from memory addresses are substituted for register reads and stores to memory addresses are substituted for register writes.

The address space for memory references is large, so the memory ORT must be managed like a set-associative cache instead of like a register file. In this paper, we use an unbounded memory ORT and then provide evidence that removal of ineffectual stores and their computation chains is not needed to achieve most of the performance. In the final analysis, we recommend a minimal IR-detector design that uses implicit back-propagation and does not support store removal (no memory ORT).

6 Characterizing Circuit Complexity

For a minimal IR-detector design that employs implicit back-propagation and does not support store removal, there are three hardware components: 1) the register ORT, 2) the FIFO buffer, and 3) a module that implements the 3-step algorithm, which consists of combinational logic and a pipeline of a few stages for reading and updating the ORT. In this paper, we only characterize the register ORT and the FIFO buffer. We characterize complexity in the context of a 4-issue superscalar processor core. The IR-detector must accept up to four new instructions per cycle, i.e., eight source operands and four destination operands per cycle.

The ORT is subdivided into two tables to optimize access to various fields. The first table contains only the *ref* bits. This table consists of 64 1-bit entries (assuming 32 integer and 32 floating-point registers). Up to 8 different bits can be set (for source operands) and 4 different bits reset (for destination operands) each cycle. Also, up to 4 different *ref* bits must be read each cycle (for destination operands) to detect potential unreferenced writes. Conceptually, this table has 12 write ports and four read ports, although the 12 write ports are misleading because an array of SR-latches specialized for set/reset operations is more efficient than an SRAM with arbitrary write ports.

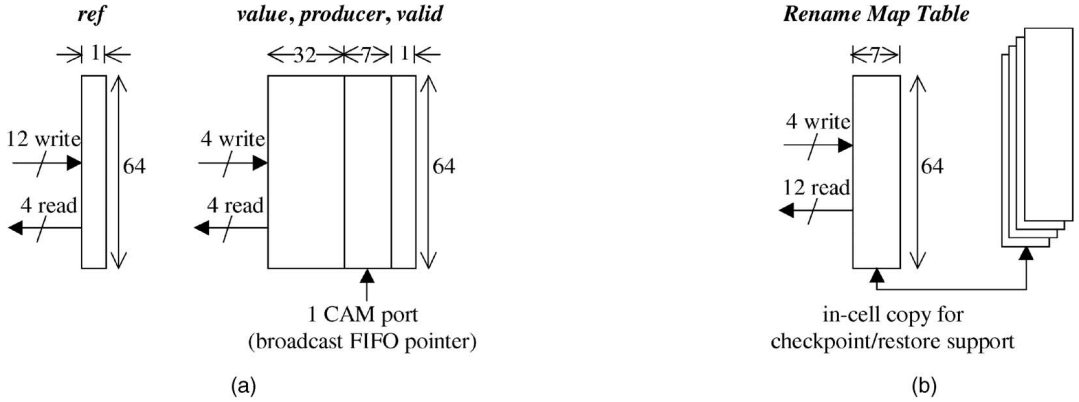


Fig. 7. (a) Ports to the four fields (two subtables) of the ORT. (b) (For comparison): Ports to the rename map table in a contemporary superscalar processor.

The second table contains the *valid* bits and *producer* and *value* fields. In Section 8.1, we determine that a FIFO of 128 instructions gives the best performance, so the *producer* is encoded with 7 bits. Values are 32 bits. Thus, this table consists of 64 40-bit entries. The table has four read ports (for destination operands) and four write ports (also for destination operands). The read ports are needed for 1) comparing previous values to new values, for detecting nonmodifying writes, and 2) locating previous producers in the FIFO if they are unreferenced writes. The write ports are needed for updating producer information.

The second table also has a single CAM port to implement the invalidation mechanism described earlier in Section 3. The CAM port facilitates comparing the rotating FIFO pointer with the *producer* field of each entry. A match causes the corresponding *valid* bit to be reset.

In terms of number of bits of storage, the ORT has a total of 64 $\times$ (7-bit *producer* + 32-bit *value* + 1 *ref* bit + 1 *valid* bit) = 2,624 bits. The read and write ports to each of the two subtables are shown in Fig. 7a. In addition to read and write ports, the figure also shows the single CAM port used to broadcast the FIFO pointer. For comparison, Fig. 7b shows read and write ports to a conventional rename map table found in contemporary superscalar processors. The rename map table has 16 ports total: eight read ports to rename logical source registers, an additional four read ports to obtain the previous mappings of logical destination registers, and four write ports to update the mappings of logical destination registers. Neither ORT subtable has more than 16 ports and only the *ref* bit table has 16 ports. Moreover, the single CAM port is far less complex than the in-cell copy functionality embedded within the rename map table, needed for checkpointing and restoring shadow map tables. Finally, the total number of bits in the ORT and rename map table are comparable: 41 bits $\times$ 64 entries for the ORT versus 63 bits $\times$ 64 entries for the rename map table, assuming eight shadow maps and 128 physical registers. The actual area of the rename map table is significantly affected by the in-cell copy functionality. In conclusion, the complexity of the ORT is comparable to the complexity of the rename map table, if not less.

The FIFO consists of 128 1-bit entries. The bits indicate whether or not the corresponding instructions have been selected for removal. Up to 4 bits in adjacent entries can be reset for incoming instructions. Up to 4 arbitrary bits can be set for unreferenced writes, nonmodifying writes, and/or correctly predicted branches.

7 Simulation Methodology

A detailed cycle-accurate simulator is used to study the slipstream processor [11], [12]. The simulator models the microarchitecture given in Fig. 1. Ineffectual instructions are speculatively skipped in the A-stream, the correct forward progress of the A-stream is checked by the R-stream, and the streams are resynchronized whenever there is an IR-misprediction. The cycle-accurate simulator is validated by a functional simulator run independently and in parallel with it [20]. The functional simulator asserts the correctness of retired R-stream control-flow and data-flow outcomes.

7.1 Microarchitecture Configuration

The slipstream microarchitecture parameters are listed in Table 1. The top-left portion describes individual processors within the CMP. The bottom-left portion describes the slipstream components. The right portion describes the slipstream memory hierarchy.

The CMP is composed of two processors. Each one is a 4-issue superscalar processor with a 64-instruction reorder buffer. Each processor has its own L1 instruction and data caches. The processors share a unified L2 cache.

A large IR-predictor is used for accurate instruction removal. IR-detectors based on both explicit and implicit back-propagation are simulated. The delay buffer stores values for up to 256 instructions and stores up to 4K branch predictions. Hardware-based memory duplication is used and A-stream memory state is recovered by invalidating dirty lines and using the invalidated data as value predictions [12]. Recovery of registers takes 21 cycles (the table gives a breakdown).

The IR-predictor is virtually unbounded in size, hence, it is infeasible. The IR-predictor is the one remaining slipstream component to be engineered and this research is underway. Until a practical implementation is produced, we do not want IR-predictor artifacts to obscure the performance evaluations in this paper. Thus, the large

TABLE 1
Microarchitecture Configuration

SINGLE PROCESSOR CORE (PE)		SLIPSTREAM MEMORY HIERARCHY	
caches	Private L1 data cache	**L1 I-cache**	Size = 64KB
	Private L1 instruction cache		Associativity = 4-way
superscalar core	Reorder Buffer: 64 instructions		Replacement = LRU
	Dispatch/issue/retire bandwidth: 4		Line size = 64 bytes
	4 universal function units	**L1 D-cache**	Size = 64KB
	4 loads/stores per cycle		
execution latencies	Address generation = 1 cycle		Associativity = 4-way
	Load access = 2 cycles		Replacement = LRU
	Integer ALU ops = 1 cycle		Line size = 64 bytes
	Complex ops = MIPS R10000 latencies	**L2 cache**	Unified instruction/data
SLIPSTREAM COMPONENTS			Shared among the PEs
IR-predictor	2^{20} entries, gshare-indexed		Size = 256KB
	Block size = 16		Associativity = 4-way
	16 confidence counters per entry		Replacement = LRU
	Confidence threshold = 32, 64,… (varied)		Line size = 64 bytes
IR-detector	Number of instructions buffered = 32, 64, 128, and 256 (varied)		Write-back policy
delay buffer	Data-flow buffer: 256 instruction entries	**memory access times**	L1 instruction hit = 1 cycle
	Control-flow buffer: 4K branch predictions		L1 data hit = 2 cycles
A-stream recovery	Memory: Invalidate dirty lines, use invalidated data as value predictions [12]		L2 hit = 12 cycles (min.)
	Register File: • Recovery latency = 21 cycles • 5 cycles to start up recovery pipeline • 4 reg. restores/cycle (64 registers total)		L2 miss = 70 cycles (min.)
		# out. misses	Unlimited for all caches
		duplication	Hardware-based memory duplication [12]

IR-predictor is used to highlight, as much as possible, the effectiveness of various IR-detector implementations.

Nonetheless, we can report our initial experiences with efficient IR-predictors, which are encouraging. Preliminary experiments indicate that a single confidence counter per entry, time-shared among instructions in the fetch block, is sufficient. Preliminary experiments also indicate that only a small fraction of all dynamic fetch blocks contribute most of the instruction removal in a program. A table of 2-bit or 3-bit confidence counters can filter out fetch blocks that are likely to make large contributions and only these fetch blocks are assigned semipermanent entries in a small cache of tagged removal-confidence counters. The filter-and-cache design yields removal rates approaching those of the unbounded IR-predictor, e.g., 35-45 percent removal, with 56 KB of storage. Preliminary experiments with the simplest design of all—associating a confidence counter with each instruction in the instruction cache—yield instruction removal rates of 30 percent or higher on benchmarks that normally achieve 40 percent removal with unbounded predictors, with only 12 KB of storage. These latter experiments suggest that deep global branch history is not so much needed for the confidence counters themselves (as previously thought) as for maximizing the prediction accuracy and, hence, removability of branches.

7.2 Benchmarks

A combination of SPEC2000 and SPEC95 integer benchmarks are used for the simulations. Our selection of benchmarks achieves nearly complete coverage of both the SPEC95 and SPEC2000 integer benchmarks within a total budget of a dozen benchmarks, our aim. Three of the 12 SPEC2000 benchmarks were already unusable. We could not get *eon* or *crafty* to compile with the Simplescalar compiler. *Mcf* causes the simulator to exhaust virtual memory on our machines because of its large footprint, coupled with the fact that we maintain two full memory images (one for the cycle-accurate simulator and one for the functional simulator). Although only four of the eight SPEC95 integer benchmarks are included, three additional ones are implicitly included due to overlap with SPEC2000 (*gcc*, *perl*, *vortex*). Thus, effectively, only two benchmarks are arbitrarily omitted to achieve the dozen-benchmark goal, *bzip* from SPEC2000 and *go* from SPEC95.

The benchmarks are compiled with –O3 optimization using the Simplescalar compiler [2]. For the SPEC2000 benchmarks, the first billion instructions are skipped and then 100 million instructions are simulated. The SPEC95 benchmarks are run to completion. The benchmarks and their inputs are given in Table 2.

8 Experimental Results

This section begins with a study of the IR-detector design space. For our set of benchmarks, the best confidence counter threshold and instruction buffer size are identified for both the explicit and implicit IR-detectors. Next, we compare the performance of the explicit and implicit IR-detectors, using their best-performing configurations. Finally, the impact of the removal of ineffectual stores on the performance of the implicit IR-detector is measured.

The performance metric is %IPC improvement of slipstream execution on two processor cores relative to

TABLE 2
Benchmarks

Benchmark	Suite	Input dataset
gap	SPEC2000	-1./ -q –m 8M ref.in
gcc	SPEC2000	expr.i –o expr.s (-O3 is hardwired)
gzip	SPEC2000	input.program 16
parser	SPEC2000	2.1.dict –batch
perl	SPEC2000	-I./lib splitmail.pl 850 5 19 18 1500
twolf	SPEC2000	Ref
vortex	SPEC2000	bendian1.raw
vpr	SPEC2000	net.in arch.in place.out dum.out –nodisp –place_only –init_t 5 –exit_t 0.005 –alpha_t 0.9412 –inner_num 2
compress	SPEC95	40000 e 2231
jpeg	SPEC95	vigo.ppm
li	SPEC95	test.lsp (queens 7)
m88ksim	SPEC95	-c < ctl.in (dcrand.big)

single-program execution on one of the processor cores. For slipstream, IPC is computed by dividing the number of retired R-stream instructions (i.e., the original program) by the number of cycles for the A-stream and R-stream combination to complete.

8.1 IR-Detector Design Space Study

Choosing the best confidence counter threshold involves balancing two competing goals—maximizing the number of removed instructions (favoring a low threshold) while minimizing the number of IR-mispredictions (favoring a high threshold). In the first part of the study, the counter threshold is increased while keeping the IR-detector instruction buffer size fixed at 128 instructions. We identify the best counter threshold for our benchmarks. After finding the best counter threshold, the threshold is held constant and the instruction buffer size is varied to find its best value.

8.1.1 Explicit Back-Propagation

The performance of the slipstream processor with explicit back-propagation is shown in Fig. 8. The confidence counter threshold is varied from 32 to 72. The instruction buffer size is 128 instructions. The notation *Qx* indicates the instruction buffer size and *Ty* indicates the counter threshold. On average, a slipstream processor with the explicit IR-detector improves IPC from 11.4 percent to 12.3 percent as the counter threshold increases from 32 to 72, with peak improvement occurring at 64. The trend is very distinct in the case of *m88ksim*, where the %IPC improvement increases from 18 percent to 21.3 percent, with the peak improvement occurring at a counter threshold of 64. The change is due to the fact that the percentage of instruction removal in the A-stream decreases negligibly as the counter threshold increases from 32 to 64 (from 66.6 percent to 65.6 percent instruction removal), while there is a 58.5 percent decrease in the number of IR-mispredictions.

Next, the instruction buffer size is varied from 32 to 256 instructions while keeping the counter threshold fixed at 64. The results are shown in Fig. 9. On average, %IPC improvement does not vary much with instruction buffer size. The peak IPC improvement of 12.3 percent occurs at a buffer size of 128 instructions. We conclude that the best configuration, on average, is a counter threshold of 64 and

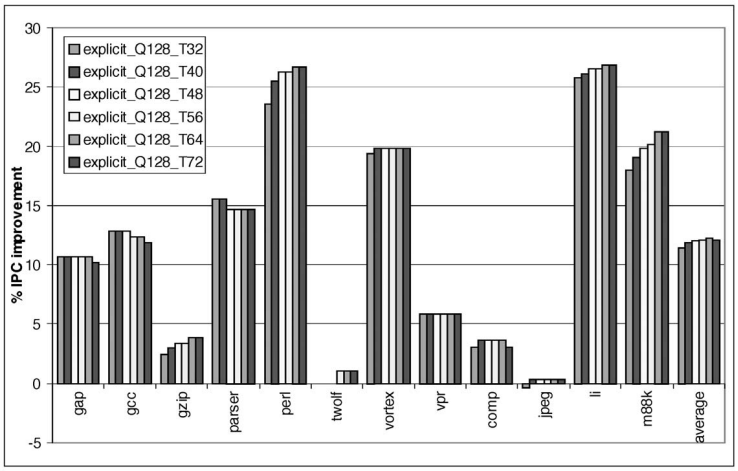


Fig. 8. Performance of the slipstream processor with explicit back-propagation, for an instruction buffer size of 128 and counter threshold varying from 32 to 72.

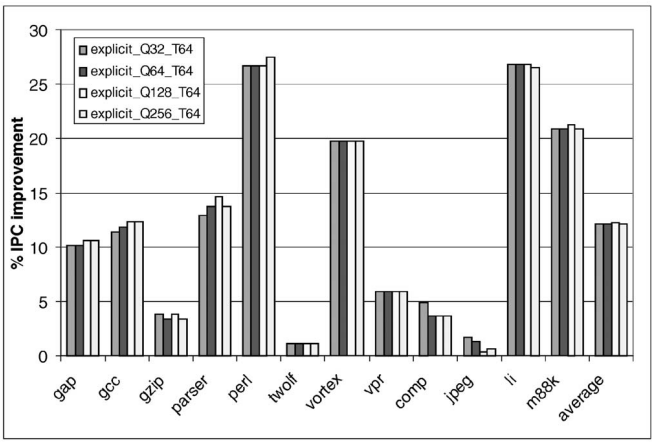


Fig. 9. Performance of the slipstream processor with explicit back-propagation, for a counter threshold of 64 and instruction buffer size varying from 32 to 256.

an instruction buffer size of 128 instructions, for the explicit IR-detector. This configuration is used for the remainder of the study.

8.1.2 Implicit Back-Propagation

The performance of the slipstream processor with implicit back-propagation is shown in Fig. 10. As before, the confidence counter threshold is varied from 32 to 72. The instruction buffer size is 128 instructions. On average, a slipstream processor with the implicit IR-detector improves IPC from 10.1 percent to 11.8 percent as the counter threshold increases from 32 to 72, with peak improvement occurring at 64. Again *m88ksim* shows a distinct trend, where the %IPC improvement increases from 16.1 percent to 20.2 percent, with the peak improvement occurring at a counter threshold of 64. The increase in %IPC improvement is due to the fact that the percentage of instruction removal in the A-stream decreases negligibly as the counter threshold increases from 32 to 64 (from 66.5 percent to 65.6 percent instruction removal), while there is a 55.4 percent decrease in the number of IR-mispredictions.

Next, the instruction buffer size is varied from 32 to 256 instructions while keeping the counter threshold fixed at 64. The results are shown in Fig. 11. On average, the %IPC improvement varies from 11.3 percent to 11.8 percent. The peak %IPC improvement of 11.8 percent occurs at an instruction buffer size of 128. Therefore, on average, the best counter threshold (64) and instruction buffer size (128) are the same for the implicit and explicit IR-detectors.

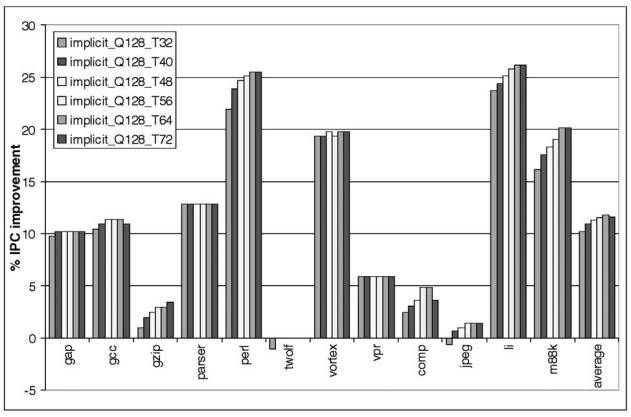


Fig. 10. Performance of the slipstream processor with implicit back-propagation, for an instruction buffer size of 128 and counter threshold varying from 32 to 72.

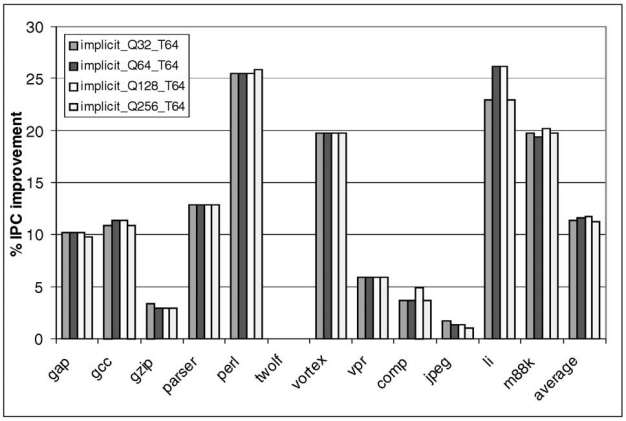


Fig. 11. Performance of the slipstream processor with implicit back-propagation, for a counter threshold of 64 and instruction buffer size varying from 32 to 256.

8.2 Comparison of Explicit and Implicit IR-Detectors

The performance comparison of slipstream processors with the best explicit (*explicit_Q128_T64*) and implicit (*implicit_Q128_T64*) IR-detectors is given in Fig. 12. The *avg* bar represents the average %IPC improvement for all 12 benchmarks. The *avg_1/3* bar is the average %IPC improvement for the six benchmarks which have more than 1/3 instruction removal in the A-stream: *gcc*, *parser*, *perl*, *vortex*, *li*, and *m88ksim*. (We provide a second average for benchmarks with more than 1/3 removal because benchmarks with little removal otherwise mask sensitivity to IR-detector configuration.) The comparison is done at a counter threshold of 64 and an instruction buffer size of 128 instructions, the best configuration for both the explicit and implicit IR-detectors.

The slipstream processor with the explicit IR-detector gives an average performance improvement of 12.3 percent, while the one with the implicit IR-detector gives an average

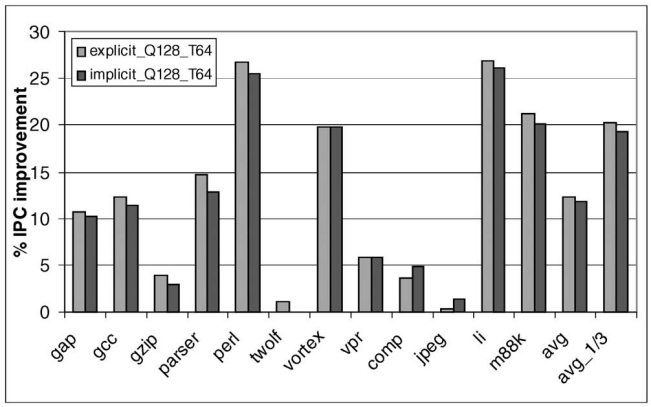


Fig. 12. Performance comparison of the explicit and implicit IR-detectors.

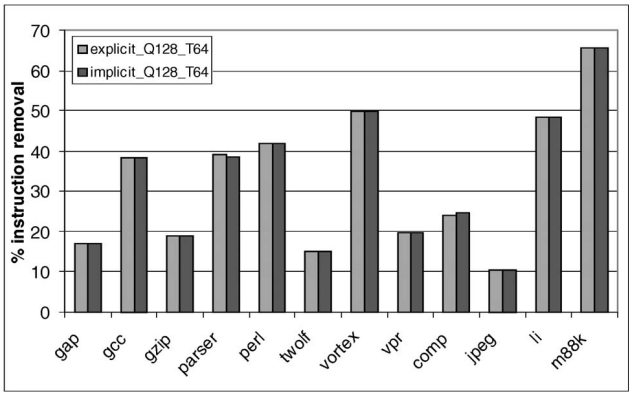


Fig. 13. Percentage of dynamic instructions removed in the A-stream using explicit and implicit back-propagation.

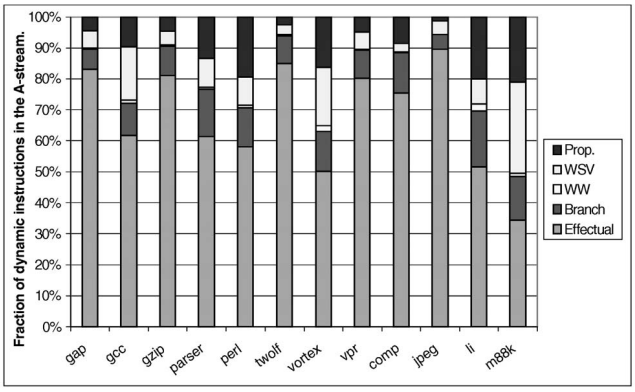


Fig. 15. Breakdown of dynamic instructions in the A-stream.

performance improvement of 11.8 percent. The average performance improvement for the six benchmarks with more than 1/3 instruction removal is 20.3 percent with explicit back-propagation and 19.3 percent with implicit back-propagation. The results show that the performance improvement of the slipstream processor with the implicit IR-detector is comparable to the performance improvement of the slipstream processor with the explicit IR-detector—within one percentage point or less, on average.

There are several possible explanations for the slight performance difference between implicit and explicit back-propagation. Either implicit back-propagation removes fewer A-stream instructions, causes more IR-mispredictions, or both. Fig. 13 shows that the percentage of dynamic instruction removal is almost the same for both IR-detectors, as we anticipated in Section 4. But, there is a noticeable increase in the number of IR-mispredictions per 1,000 instructions for implicit back-propagation, as shown in Fig. 14, except for *compress* and *jpeg*. And, for these two benchmarks, the implicit IR-detector outperforms the explicit IR-detector. The typically lower performance of the implicit IR-detector can be attributed to the increase in the number of IR-mispredictions, confirming the analysis in Section 4. When the last instruction in a chain causes an IR-misprediction, there is a cascade of N-1 additional IR-mispredictions for a chain N instructions long. On the other hand, the explicit IR-detector does not incur additional IR-mispredictions after the initial one.

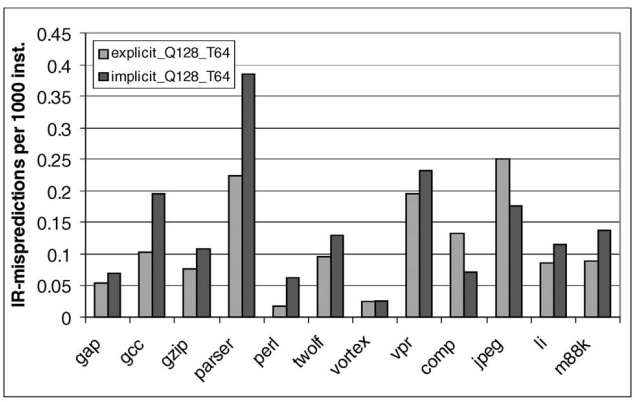


Fig. 14. IR-mispredictions per 1,000 instructions using explicit and implicit back-propagation.

It is purely by chance that *jpeg* has fewer IR-mispredictions with implicit back-propagation than with explicit back-propagation. The anomaly occurs inside a tight loop in the *gen_huffman_coding* function. An instruction A that was previously ineffectual becomes effectual again. Unfortunately, many new instances of A continue to be removed in the tight loop because it takes a trip through the FIFO before the IR-predictor is updated to reset A's counter. This usually causes repeated IR-mispredictions until the IR-predictor is updated. However, due to delayed dismantling of an *unrelated* ineffectual strand in the case of the implicit IR-detector, another instruction B is also declared ineffectual when it is not. Serendipitously, this convergence of "two wrongs" make a "right." While the two instructions do not meet *our* ineffectual criteria, removing both A and B did not actually affect correct forward progress, whereas removing A and not removing B does affect correct forward progress.

Two conclusions can be drawn from the *jpeg* anomaly. First, delayed IR-predictor updates are potentially pathological in tight loops and this aspect should be addressed in future work. Implicit back-propagation serendipitously masked the pathological behavior in the specific scenario discussed above, but masking is not guaranteed in general. Second, observing the masking effect made us realize that there are more general ineffectual criteria yet to be exploited. Our three ineffectual criteria are easy to detect, but they do not capture all ineffectual behavior.

8.3 Breakdown of Dynamic Instructions in the A-Stream

The breakdown of dynamic instructions in the A-stream is given in Fig. 15. The breakdown was measured for a slipstream processor with the implicit IR-detector. Results are almost the same for the explicit IR-detector. The *effectual* component is the fraction of instructions in the A-stream that were not removed. The *branch* component is the fraction of instructions that were removed due to correctly predicted branches. The *WSV* component is the fraction of instructions that were removed due to nonmodifying writes ("write-same-value"). The *WW* component is the fraction of instructions that were removed due to original unreferenced writes ("write-write"). The *propagation* component is

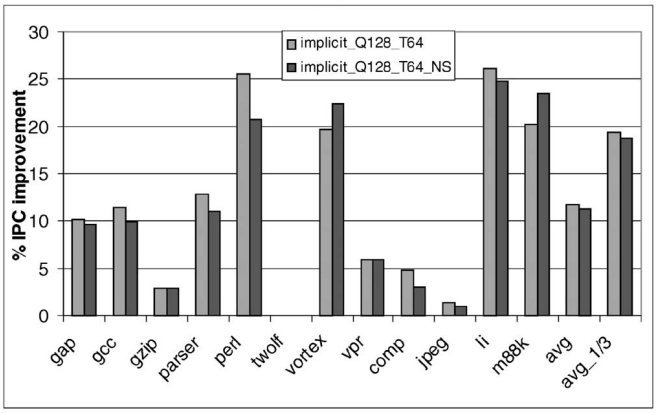


Fig. 16. Effect of store removal on slipstream processor performance.

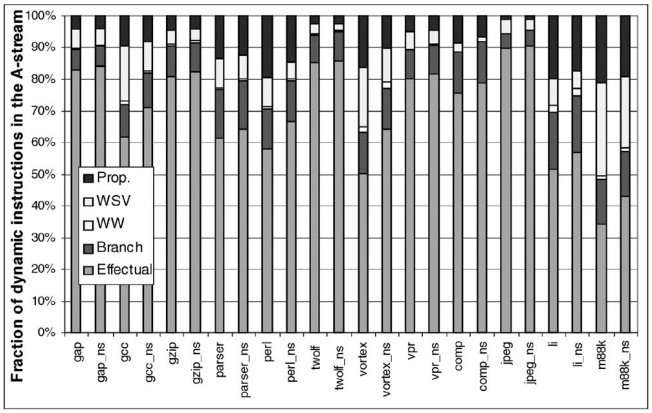


Fig. 17. Breakdown of dynamic instructions in the A-stream, with and without store removal. The implicit IR-detector is used.

the fraction of instructions removed due to back-propagation; note, these are exposed as unreferenced writes by the implicit IR-detector, but we separate this component from the *WW* component. As shown in Fig. 15, branches, nonmodifying writes, and back-propagation are the major sources of instruction removal. Original unreferenced writes are not as significant as the other three components.

8.4 Removal of Stores

A memory operand rename table tracks references to memory locations. It is needed to detect unreferenced stores, nonmodifying stores, and stores that are referenced by ineffectual loads (back-propagation). The IR-detector can be simplified further if we choose not to remove store instructions. In this case, only a register operand rename table is used and it is similar to the rename table in conventional superscalar processors. The %IPC improvement of a slipstream processor with the implicit IR-detector, both with (*implicit_Q128_T64*) and without (*implicit_Q128_T64_NS*) store removal, is shown in Fig. 16. The breakdown of instructions in the A-stream with and without store removal is shown in Fig. 17. (The suffix *_ns* indicates no store removal.)

The average slipstream performance improvement decreases from 11.8 percent to 11.2 percent, without store removal. The average slipstream performance improvement decreases from 19.3 percent to 18.7 percent for the six benchmarks with more than 1/3 instruction removal. This is primarily due to a decrease in the number of instructions removed from the A-stream. Fig. 17 shows a noticeable decrease in the number of instructions removed, the categories of nonmodifying writes (*WSV*) and propagation (*Prop.*) being impacted the most. But, *m88ksim* and *vortex* actually perform better without the removal of ineffectual stores. Although the percentage of instructions removed from the A-stream decreases by 9 percent in *m88ksim* and 15 percent in *vortex*, overall performance improves as a result of an increase in the number of value predictions communicated from the A-stream to the R-stream.

Note that the trade off between the extra complexity and performance benefits of store removal may shift for different assumptions about the number of L1 cache ports, MHSRs, etc.

9 Related Work

Sundaramoorthy et al. [11], [13], [16] proposed the first IR-detector for slipstream processors. Key ineffectual instructions (unreferenced writes, nonmodifying writes, and correctly predicted branches) are selected for removal first, using an operand rename table (ORT). Instructions feeding the trigger instructions are selected using a reverse data-flow graph (RDFG). The RDFG consists of physical connections from consumer instructions to their producers. For N instructions in the RDFG, there are N^2 wires and N complex logic blocks [19]. Implicit back-propagation eliminates the RDFG component.

This paper is a condensed version of Koppanalil's MS thesis [19], which contains more details, including a gate-level implementation of the RDFG and a full description of the old IR-detector based on explicit back-propagation.

Roth and Sohi [14], [15] proposed Speculative Data-Driven Multithreading, an architecture for preexecuting threads to resolve likely mispredicted branches early and prefetch possible cache misses. They do not propose a hardware mechanism for constructing preexecution threads. Instead, they use an offline profile-driven approach for identifying the backward slices of unpredictable branches and loads that tend to miss frequently. Zilles and Sohi [17], [18] also studied the use of preexecution to reduce the impact of performance-degrading instructions. They used profiling and manual analysis to construct the preexecution threads.

Collins et al. [3] proposed a hardware mechanism for dynamically constructing precomputation slices (p-slices) of delinquent loads (loads that tend to miss). A table records the miss rates of loads and they are dynamically classified as delinquent or not. A Retired Instruction Buffer (RIB) buffers retired instructions between two instances of a delinquent load. When the second instance is detected, the RIB stops receiving new retired instructions and begins analyzing buffered instructions. Instructions in the RIB are scanned serially, starting from the second instance of the delinquent load and moving steadily backward. Scanning identifies instructions in the backward slice of the delinquent load. When the p-slice is constructed, it is optionally optimized and then stored in a p-slice cache. Other precomputation architectures [1], [9] use similar approaches

for dynamically constructing p-slices or use compiler construction of p-slices [7].

Many retired instructions are "dropped" while the RIB is busy analyzing a region of the dynamic instruction stream. Passing over dynamic instructions is not a problem in the context of precomputation. A region only has to be analyzed once because its p-slice does not change. Therefore, there is no urgency to construct p-slices. If a region containing a new p-slice is passed over because the RIB is busy constructing another p-slice, the new p-slice is simply constructed the next time its containing region is seen again.

On the other hand, a slipstream processor must continuously monitor the ineffectualness of all dynamic instructions in order to build up confidence. The IR-detector based on implicit back-propagation is a practical means for continuous monitoring. The key innovation is not dealing explicitly with dependence chains. This yields an approach that avoids buffering actual instructions in an RDFG or RIB and enables forward analysis by means of a rename table.

The slice-processor developed by Moshovos et al. [9] employs scout threads to preexecute problem loads and branches. Scout threads are constructed using the *slicer*. The slicer buffers retired instructions. When a candidate instruction is retired, the buffer is sequentially scanned backward to collect instructions in its backward slice. The backward scan involves propagating a dependence vector from one slicer entry to the next. The slicer is conceptually similar to the RIB. However, a domino logic implementation of dependence vector propagation is proposed. Furthermore, two slicers are used. A working slicer receives committed instructions, while a shadow slicer performs slice detection. The working slicer reduces the chance that a candidate instruction is dropped while a slice is being constructed.

10 Summary and Future Work

The single-chip multiprocessor is a promising framework for future microprocessors. CMPs deliver high performance for multiprogrammed and parallel/multithreaded workloads. Each constituent processor delivers high single-program performance with a balanced design that combines a fast clock and moderate ILP. Slipstream extends the capability of CMPs by enabling a single program to exploit a second idle core. Slipstream runs two redundant copies of the program, one a speculatively reduced version (A-stream) and the other a checker (R-stream). The redundant programs collaborate to finish faster than conventional nonredundant execution.

In this paper, we examined in depth the slipstream component responsible for detecting past-ineffectual instructions, the IR-detector. The IR-detector first selects unreferenced writes, nonmodifying writes, and correctly predicted branches. It then selects instructions feeding these trigger instructions. Unreferenced and nonmodifying writes are easily selected using a table indexed by logical registers. Selecting instructions in their backward slices previously required an elaborate mechanism, called *explicit back-propagation*, whereby consumer instructions signal ineffectual status to their producers via direct links.

We proposed a new method, called *implicit back-propagation*, that reduces back-propagation to the detection of unreferenced writes, eliminating a complex subcomponent of the IR-detector. The key idea is to logically monitor the reduced A-stream and only select unreferenced writes, nonmodifying writes, and correctly predicted branches. Once these trigger instructions are removed from the A-stream, their producers are exposed as unreferenced writes. The same method for selecting original unreferenced writes works for freshly exposed unreferenced writes. Once they are removed, additional unreferenced writes are exposed and implicit back-propagation proceeds iteratively until entire ineffectual dependence chains are removed.

The analysis in this paper reveals several major conclusions.

- A slipstream processor with explicit back-propagation improves performance by an average of 12.3 percent (relative to conventional nonredundant execution), while a slipstream processor with implicit back-propagation improves performance by an average of 11.8 percent. Performance improvements are 20.3 percent and 19.3 percent, respectively, for benchmarks with more than 1/3 instruction removal. In other words, with implicit back-propagation, hardware complexity is significantly reduced with only minor performance impact.
- Explicit and implicit back-propagation take the same amount of time to build predicted-ineffectual chains, but implicit back-propagation is less timely in dismantling them. Measurements of the amount of instruction removal and rate of IR-mispredictions confirm this analysis. The two approaches achieve about the same amount of instruction removal (same training time for building chains), but implicit back-propagation has significantly more IR-mispredictions (dismantles chains slower). Thus, the slight performance difference is due to IR-mispredictions.
- Removal of ineffectual stores requires a cache-like structure to track references to memory locations. Average performance improvement drops from 11.8 percent to 11.2 percent without store removal. Average performance improvement drops from 19.3 percent to 18.7 percent, for benchmarks with more than 1/3 instruction removal. Thus, a minimal IR-detector that employs implicit back-propagation and does not support store removal achieves a good balance between complexity and performance.
- A minimal IR-detector design consists primarily of a table indexed by logical register. Complexity analysis reveals three out of four fields in the table require four read and four write ports (assuming a 4-issue superscalar core). The fourth 1-bit field requires four read and 12 write ports, although the write ports can be tailored for set/reset operations on the bit.

By diagnosing the *jpeg* anomaly, we uncovered the potential for new ineffectual criteria. For example, the nonmodifying write criterion can be expanded to include writes that modify the value in a location if skipping them does not change the course of the program. Developing new ineffectual criteria is part of our ongoing work.

Currently, we are working on dynamically enabling/disabling slipstream mode by continuously gauging potential instruction removal, as measured by the IR-detector/IR-predictor.

ACKNOWLEDGMENTS

This research was supported in part by US National Science Foundation CAREER grant CCR-0092832 and generous funding and equipment donations from Intel Corporation.

REFERENCES

[1] M. Annavaram, J. Patel, and E. Davidson, "Data Prefetching by Dependence Graph Precomputation," *Proc. 28th Int'l Symp. Computer Architecture,* July 2001.
[2] D.C. Burger, T.M. Austin, and S. Bennett, "The Simplescalar Tool Set, Version 2.0," Technical Report 1342, Computer Sciences Dept., Univ. of Wisconsin-Madison, 1997.
[3] J.D. Collins, D.M. Tullsen, H. Wang, and J.P. Shen, "Dynamic Speculative Precomputation," *Proc. 34th Int'l Symp. Microarchitecture,* Dec. 2001.
[4] E. Jacobsen, E. Rotenberg, and J. Smith, "Assigning Confidence to Conditional Branch Predictions," *Proc. 29th Int'l Symp. Microarchitecture,* Dec. 1996.
[5] J. Kahle, "Power4: A Dual-CPU Processor Chip," *Microprocessor Forum,* Oct. 1999.
[6] K.M. Lepak and M.H. Lipasti, "On the Value Locality of Store Instructions," *Proc. 27th Int'l Symp. Computer Architecture,* June 2000.
[7] C.-K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," *Proc. 28th Int'l Symp. Computer Architecture,* July 2001.
[8] S. McFarling, "Combining Branch Predictors," Technical Report TN-36, WRL, June 1993.
[9] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi, "Slice Processors: An Implementation of Operation-Based Prediction," *Proc. 15th Int'l Conf. Supercomputing,* June 2001.
[10] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang, "The Case for a Single-Chip Multiprocessor," *Proc. Seventh Int'l Symp. Architectural Support for Programming Languages and Operating Systems,* Oct. 1996.
[11] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A Study of Slipstream Processors," *Proc. 33rd Int'l Symp. Microarchitecture,* Dec. 2000.
[12] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "Slipstream Memory Hierarchies," Technical Report CESR-TR-02-3, Center for Embedded Systems Research, North Carolina State Univ., Feb. 2002.
[13] E. Rotenberg, "Exploiting Large Ineffectual Instruction Sequences" technical report, North Carolina State Univ., Nov. 1999.
[14] A. Roth and G.S. Sohi, "Speculative Data-Driven Multithreading," *Proc. Seventh Int'l Conf. High-Performance Computer Architecture,* Jan. 2001.
[15] A. Roth and G.S. Sohi, "Speculative Data-Driven Multithreading," Technical Report CS-TR-00-1414, Computer Sciences Dept., Univ. of Wisconsin-Madison, Feb. 2000.
[16] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream Processors: Improving Both Performance and Fault Tolerance," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* Nov. 2000.
[17] C.B. Zilles and G.S. Sohi, "Execution-Based Prediction Using Speculative Slices," *Proc. 28th Int'l Symp. Computer Architecture,* July 2001.
[18] C.B. Zilles and G.S. Sohi, "Understanding the Backward Slices of Performance-Degrading Instructions," *Proc. 27th Int'l Symp. Computer Architecture,* June 2000.
[19] J.J. Koppanalil, "A Simple Mechanism for Detecting Ineffectual Instructions in Slipstream Processors," MS thesis, North Carolina State Univ., May 2002.
[20] E. Rotenberg, "Trace Processors: Exploiting Hierarchy and Speculation," PhD thesis, Univ. of Wisconsin-Madison, 1999.
[21] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge, "Thread Level Parallelism of Desktop Applications," *Proc. Workshop Multithreaded Execution, Architecture, and Compilation,* Jan. 2000.

Jinson J. Koppanalil received the BTech (honors) degree in instrumentation engineering from the Indian Institute of Technology, Kharagpur, India, in 1999 and the MS degree in computer engineering from North Carolina State University, Raleigh, in 2002. He is currently with ARM in Austin, Texas, where his responsibilities include logic and physical design of microprocessor cores. His main research interests are in the area of high-performance microarchitecture.

Eric Rotenberg received the BS degree in electrical engineering (1991) and MS and PhD degrees in computer sciences (1996, 1999) from the University of Wisconsin-Madison. He has been an assistant professor of electrical and computer engineering at North Carolina State University since August 1999. From 1992 to 1994, he participated in the design of IBM's AS/400 computer in Rochester, Minnesota. His main research interests are in high-performance computer architecture. His current projects cover a variety of topics in architecture, including new processor architectures for performance and fault tolerance, multiprocessors, embedded and real-time systems, and molecular electronics. He is a member of the IEEE and the IEEE Computer Society.