Abstract

KOPPANALIL, JINSON JOSEPH

A Simple Mechanism for Detecting Ineffectual Instructions in Slipstream Processors

(Under the direction of Dr. Eric Rotenberg)

The slipstream paradigm harnesses multiple processing elements in a chip multiprocessor (CMP) to speed up a single, sequential program. It does this by running two redundant copies of the program, one slightly ahead of the other. The leading program is the Advanced Stream (A-stream) and the trailing program is the Redundant Stream (R-stream). Predicted non-essential computation is speculatively removed from the A-stream. The A-stream is sped up because it fetches and executes fewer instructions than the original program. The trailing 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. The R-stream also exploits the A-stream outcomes as accurate branch and value predictions. Therefore, 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.

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, non-modifying writes, and

correctly-predicted branches. A table similar to a conventional register rename table can easily detect unreferenced and non-modifying 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 connected to their producer instructions using a configurable interconnection network. 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 complex because it requires a configurable interconnection network. This thesis proposes a simpler implementation of back-propagation, called *implicit back-propagation*. The key idea is 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, non-modifying 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). After building up confidence, the freshly exposed unreferenced writes are also removed, exposing additional unreferenced writes. This process continues iteratively, until eventually entire non-essential 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 performs within 0.5% of explicit back-propagation.

A SIMPLE MECHANISM FOR DETECTING INEFFECTUAL INSTRUCTIONS IN SLIPSTREAM PROCESSORS

by

JINSON JOSEPH KOPPANALIL

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER ENGINEERING

Raleigh

2002

Approved by

____
Dr. Eric Rotenberg, Chair of Advisory Committee

____
Dr. Gregory T. Byrd

____
Dr. Thomas M. Conte

BIOGRAPHY

Jinson Joseph Koppanalil was born on August 19, 1977 in Cochin, India. He graduated with a B.Tech (Honours) degree in Instrumentation Engineering from the Indian Institute of Technology (IIT), Kharagpur, India, in June 1999. After graduation, he worked as a VLSI/system designer at Wipro Technologies, Bangalore, India, for one year.

He then joined the masters program in Computer Engineering at North Carolina State University, Raleigh, NC. There he was a part of the Slipstream Project, and worked on his thesis under the direction of Dr. Eric Rotenberg. While working towards the masters degree, he also spent a summer interning with Applied Micro Circuits Corporation, Raleigh, NC.

ACKNOWLEDGEMENTS

Graduate school at NC State has been a truly wonderful experience. I take this opportunity to thank the people who enriched the two years that I spent here.

First and foremost, I would like to express my sincere thanks to my advisor Dr. Eric Rotenberg, for his guidance, support, patience, and his constant encouragement. Working with him has been a fantastic learning experience. I learned from him several aspects of computer architecture, which I could not have learned in a normal class room. I also thank him for his confidence in me, and for giving me an opportunity to explore a variety of research topics, although not directly related to my thesis.

I would like to thank the other members of my advisory committee, Dr. Greg Byrd and Dr. Tom Conte, for reviewing my thesis, and for their valuable comments. I would like to thank both of them, along with all the other faculty members, for providing me with such an excellent education. Thanks again to Dr. Conte for introducing me to the computer architecture group, here at NC State.

I would like to thank the fellow members of the Slipstream project, Karthik Sundaramoorthy, Zach Purser, and Nikhil Gupta, for helping me better understand the various aspects of slipstream processors. Thanks again to Karthik and Zach for being patient and helping me whenever I had problems with the simulator.

I would also like to thank all my friends who made my stay at NC State more enjoyable. I am fortunate to have great friends like Saurabh Sharma, Nagendra Kumar, Sandeep Chandra, Uday Rao, Srikanth Jayanthi, Boby George, Satish Uppathil, Indradeep Sen, Vinay Mahadik, Vijay Iyer, Prabhas Sinha, and Girish Kulkarni. Thanks for your friendship. It means a lot to me.

Finally, I would like to thank my parents, Joseph and Moly, for their unconditional love and continual encouragement. I thank them for the active role they played in my education and also guiding me whenever I needed their advice. I cannot thank them enough.

My work has been supported by NSF CAREER grant No. CCR-0092832, and generous funding and equipment donations from Intel. I am truly grateful for their support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Recent trends in microarchitecture research reveal a move towards architectures that can efficiently leverage the billion transistor chips promised by future technologies. One such architecture is a chip multiprocessor (CMP) [5,10], which incorporates multiple processor cores on the same chip. A CMP can simultaneously run multiple independent jobs, multiple tasks from a parallel program, or multiple threads from a multithreaded program. But, a disadvantage of CMPs is that its multiple processors cannot be used to speedup a single sequential program.

The *slipstream paradigm* enables a CMP to also be used for improving single-program performance. A slipstream processor [11,16] runs two redundant copies of a program on a dual-processor CMP, one slightly ahead of the other. The leading program is the Advanced Stream (A-stream) and the trailing program is the Redundant Stream (R-stream). A significant number of predicted-ineffectual instructions are speculatively removed from the A-stream. The A-stream is sped up because it fetches and executes fewer instructions than the original program. The trailing 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. Therefore, 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.

Removing instructions that are not needed for correct forward progress is a key aspect of the slipstream processor. Many general purpose programs contain a significant number of removable 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. Then, there are instructions whose outcomes are highly predictable, for example, branch instructions. These instructions – unreferenced writes, non-modifying writes, and correctly-predicted branches 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 processor relies on hardware techniques for accurate instruction removal in the reduced program.

1.1 Contributions

The component of a slipstream processor that identifies non-essential instructions is called the *instruction-removal detector* (IR-detector) [11,12,16]. It 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, non-modifying writes, and correctly-predicted branches. Second, the computation chains that lead up to the selected instructions are also selected. This second step involves back-propagating selection status through chains of dependent instructions. This thesis is aimed at a simple, yet effective, implementation of back-propagation. The contributions are as follows.

- An explicit implementation of back-propagation involves constructing a reverse data flow graph in hardware, by buffering retired R-stream instructions and setting up connections from consumer instructions to their producers. Although the explicit implementation of back-propagation performs well, it is complex. An alternative implementation, based on a novel idea called *implicit back-propagation*, is proposed. It is much simpler and performs within 0.5% of the explicit implementation. The key idea is that back-propagation can be achieved implicitly, i.e., without the support of any explicit circuitry, if the IR-detector *logically* monitors the A-stream instead of the R-stream. Now, the IR-detector performs only the first step: it selects unreferenced writes, non-modifying 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 non-essential dependence chains are removed. Logically monitoring the A-stream reduces the problem of back-propagation to detecting unreferenced writes.

- We explain how an otherwise correct IR-detector algorithm can introduce many spurious A-stream mispredictions, by not accounting for artifacts of the IR-predictor. We describe IR-detector modifications that compensate for external artifacts beyond its control.

- Removal of stores [13] (e.g., removal of silent stores [6], unreferenced stores, and stores that are in the backward slices of loads which were removed） requires the IR-detector to track the history of load and store references to memory locations. This requires a cache-like structure since the number of memory locations to be tracked is potentially unbounded. The IR-detector is further simplified by opting not to support the removal of stores, eliminating the need for the cache-like structure. The results in this thesis demonstrate that performance is still within 1% of the original IR-detector implementation.

- The circuit complexity of IR-detectors based on explicit and implicit back-propagation is characterized. In the course of doing this, we had to design the logic for the reverse data flow graph in the case of explicit back-propagation. This is the first actual design of the explicit back-propagation mechanism.

- This thesis explores the design space for the IR-detector, which includes the instruction buffer size and confidence counter threshold. Simulations on the SPEC95 and SPEC2K benchmarks indicate that the best instruction buffer size 128, and the best confidence counter threshold is 64, for both the explicit and implicit IR-detectors.

1.2 Organization of the Thesis

Chapter 2 gives an overview of slipstream processors. The insight and rationale for implicit back-propagation, the key contribution of this thesis, is described in Chapter 3. Chapters 4 and 5 describe the IR-detectors based on explicit and implicit back-propagation, respectively. The simulation methodology and benchmarks used are described in Chapter 6, and Chapter 7 presents the experimental results. Other related work is presented in Chapter 8, focusing on hardware techniques for program data flow analysis. Chapter 9 summarizes the thesis and proposes future work.

Chapter 2

Slipstream Processors

This chapter reviews slipstream processors [11,12,13,16]. We begin with the underlying paradigm in Section 2.1, followed by the slipstream microarchitecture in Section 2.2.

2.1 Slipstream Paradigm

A general-purpose program typically contains computation that does not influence correct forward progress of the program, or whose influence is highly predictable [13]. Therefore, it is possible to make correct forward progress by executing only a subset of the original dynamic instruction stream. A slipstream processor is built around this idea. A slipstream processor creates a shorter instruction stream by skipping predicted-ineffectual computation. However, once skipped, there is no sure way of knowing whether the skipped instructions are truly ineffectual or correctly predicted, since they are not executed. The solution is to verify the speculative program by comparing it against the full version of the program.

Therefore, a slipstream processor runs two copies of the same program. The two redundant programs are executed simultaneously on a single-chip multiprocessor (CMP). Each copy has its own context. One of the programs is speculatively reduced and 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.

Retired R-stream instructions are monitored, and the instructions that repeatedly do not influence the correct forward progress of the program are identified. Future instances of these ineffectual instructions are removed from the A-stream. The reduced A-stream is sped up because it fetches and executes fewer instructions. All data and control outcomes from the A-stream are communicated to the R-stream. The R-stream compares the communicated values against its own outcomes. If a deviation is detected, the corrupted A-stream context is recovered from the R-stream context. The R-stream also uses the communicated values from the A-stream as predictions. Although the R-stream is unreduced in terms of retired instructions, it is sped up because it leverages the near perfect predictions from the A-stream. The result is that the two redundant programs together complete sooner than a single copy of the program.

2.2 Slipstream Microarchitecture

The slipstream microarchitecture must support separate architectural contexts for the A-stream and R-stream. It also requires components for managing instruction removal in the A-stream and for communicating A-stream outcomes to the R-stream.

A high-level block diagram of a slipstream processor implemented on a dual-processor chip-multiprocessor (CMP) is shown in Figure 2.1. The A-stream is shown on the left and the R-stream is shown on the right. The shaded blocks show the processors and the shared L2 cache which constitute the original CMP. Each processor core is a conventional superscalar or VLIW processor with a branch predictor, instruction and data caches, register file, and execution engine.

Slipstreaming requires three new components.

1. The *instruction-removal predictor*, or IR-predictor, is essentially 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.
2. The *instruction-removal detector*, or IR-detector, identifies instructions which were not essential for the R-stream's 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.
3. The *delay buffer* is used to communicate the data and control flow outcomes from the A-stream to the R-stream.

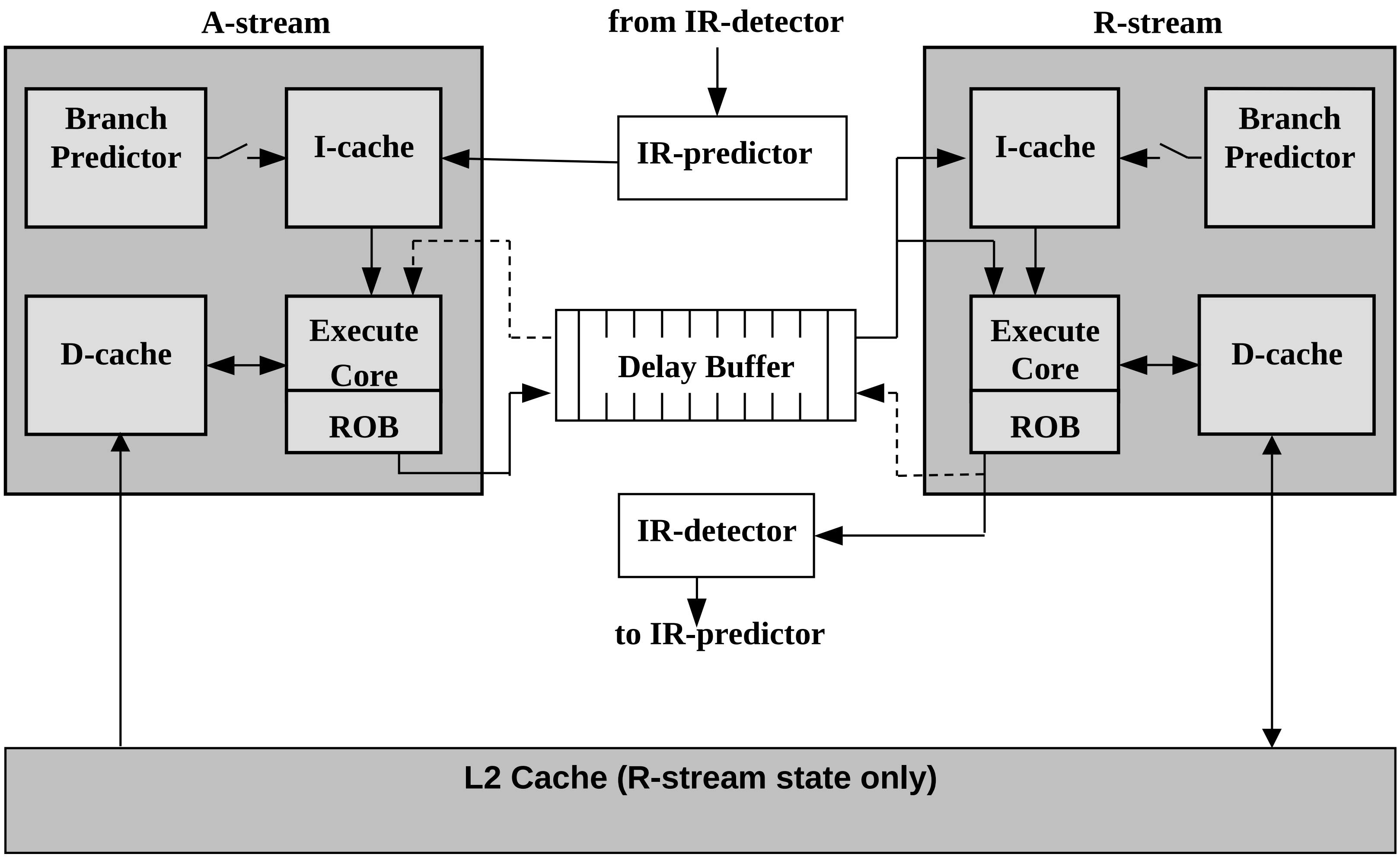


FIGURE 2.1 Slipstream processor using a dual-processor CMP [11,12,16].

2.2.1 IR-predictor

The IR-predictor is a conventional branch predictor augmented to keep track of instruction removal information. It is indexed like the *gshare* predictor [8], by XORing the PC with the global branch history bits. Each table entry contains the following information for a dynamic basic block.

- *Tag*: This is the start PC of the basic block and is used to determine whether the entry contains information for the block being fetched. A partial tag can be used to reduce the total storage, if predictor aliasing is negligible.
- *2-bit counter*: If the basic block ends with a conditional branch, then the 2-bit counter predicts its outcome.
- *Confidence counters*: There is a resetting confidence counter [4] for each instruction in the block. The counter corresponding to a particular instruction is incremented, if the IR-detector detected that the instruction was ineffectual. Otherwise, the counter is reset to zero. Repeated indications by the IR-detector saturate the confidence counter, in which case the corresponding instruction is removed from the A-stream when it is next encountered.

The number of instructions *executed* in the A-stream can be reduced with the IR-predictor discussed above. But, the number of instructions *fetched* remains unreduced. In some cases, processor performance may be restricted by instruction fetch. The A-stream can be made more efficient, if instruction fetching is also reduced by the IR-predictor.

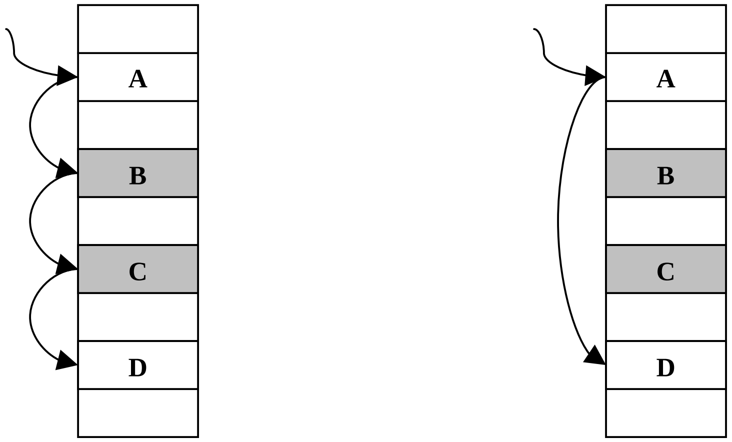


FIGURE 2.2 Bypassing fetching of ineffectual basic blocks.

The example in Figure 2.2 shows how the number of A-stream fetch cycles can be reduced. Consider four basic blocks, A, B, C, and D, which are in the predicted path of the A-stream. All the confidence counters within blocks B and C are saturated (indicated with shading), and therefore the blocks are predicted to be skipped entirely. A simple IR-predictor predicts and fetches the four blocks in sequence, and the process takes four cycles. After instruction fetch, blocks B and C are found to be predicted-ineffectual, and these blocks are removed before the decode stage. The IR-predictor can be modified to reduce the number of fetch cycles to two, by bypassing the fetch of blocks B and C. An improved IR-predictor does instruction fetching as shown on the right-hand side of Figure 2.2. This requires two additional pieces of information in block A’s IR-predictor entry.

- The predicted directions of branches in the bypassed blocks must be stored, i.e., the predicted directions of branches in blocks B and C. The reason is that a complete history of A-stream control flow needs to be passed to the R-stream, for checking, and also for directing the R-stream instruction fetch unit.
- A target address must be stored, in this case the start PC of block D. The target address overrides the branch target buffer in the fetch unit, so that block D can be fetched immediately following block A. The branch target buffer could be augmented to support multi-way branches, instead of storing targets in the IR-predictor [11].

The additional pieces of information are captured in block A’s entry as the entries of blocks B, C, and D are updated by the IR-detector.

2.2.2 IR-detector

The IR-detector consumes retired R-stream instructions and data. It then identifies instructions which were not essential for correct forward progress, in retrospect. The IR-detector watches for any of the following three triggering conditions for instruction removal.

- Unreferenced writes, i.e., a write followed by a write to the same location, with no intervening read.
- Non-modifying writes, i.e., a write that does not modify the value of a location.
- Correctly-predicted branches.

When any of the above conditions are observed, the corresponding instruction is selected for removal and this information is passed on to the IR-predictor.

Additional ineffectual instructions are removed from the A-stream by a technique called *back-propagation*. Back-propagation detects computation chains that feed the instructions selected for removal based on the triggering conditions mentioned above. An instruction can be selected for removal if all of its dependent instructions are selected for removal. For example, once a branch is selected, the computation leading to that branch is no longer needed and can be selected for removal, if no other instructions depend on the computation.

Efficient implementation of back-propagation, and the IR-detector as a whole, is the topic of this thesis.

2.2.3 Delay buffer

The delay buffer is a FIFO queue used to communicate control and data outcomes from the A-stream to the R-stream. The A-stream communicates a complete history of branch outcomes and a partial history of operand values through the delay buffer. This is shown in Figure 2.1 with a solid arrow from the A-stream retirement unit to the delay buffer. The control flow history is complete, since the A-stream predicts all branches even though it may not fetch all instructions, as described in Section 2.2.1. However, the data history is incomplete since the A-stream executes only a subset of the program.

The R-stream uses the control and data outcomes from the delay buffer as predictions. This is shown in Figure 2.1 with a solid arrow from the delay buffer to the instruction cache and execution core of the R-stream. The branch outcomes are used to direct instruction fetching from the instruction cache. The data outcomes (source operand values and load/store addresses) are used as value predictions in the execution core. To bind the data outcomes to corresponding R-stream instructions, the delay buffer also contains 1 bit per dynamic instruction that indicates whether the corresponding instruction was skipped or not.

2.2.4 Memory hierarchy

The A-stream and R-stream in a slipstream processor 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 allocate separate physical memory pages for each program. But, the simplicity of *software-based memory duplication* comes at the expense of doubling memory usage. And, this may be unacceptable in many commercial systems.

Therefore, a more elegant *hardware-based memory duplication* scheme is used in slipstream processors [12]. 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. The scheme works as follows.

- Both the A-stream and R-stream read/write their respective L1 caches normally.

- Both streams share a common L2 cache.
- 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 Figure 2.1, the R-stream reads and writes the L2 cache, but the A-stream only reads from it.

It turns out that the eviction of dirty lines in the A-stream L1 cache, and the corresponding loss of updates that they contain, 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 re-referenced by the A-stream. Occasionally, the A-stream re-references 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. The A-stream is speculative in any case, and all mispredictions are recoverable, without the need for discriminating the type of misprediction.

2.2.5 IR-mispredictions and 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 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 corrupted A-stream state needs to be recovered from the R-stream state, i.e., the A-stream needs to be re-synchronized with respect to the R-stream. This involves restoring A-stream register and memory state from R-stream registers and memory. Copying all register values from the R-stream to the A-stream is feasible, since the register file is finite. The movement of register file data occurs through the delay buffer, as shown in Figure 2.1 with dashed arrows. Or, software exception handlers simultaneously executed on both cores can copy data via shared-memory loads and stores from the R-stream register file to the A-stream register file.

Potentially, recovery of memory state is a more involved process. If software-based memory duplication is used, corrupt memory locations must be pin-pointed [11,16]. Fortunately, recovering is much simpler if hardware-based memory duplication is used. Hardware-based memory duplication is used in more recent slipstream processor implementations [12]. Two simple and effective schemes are given below.

- *Flush cache*: The A-stream memory state is recovered by invalidating all the lines in the A-stream's L1 cache. Compulsory misses in the L1 cache force the A-stream to access the correct and up-to-date R-stream memory state available in the shared L2 cache.
- *Flush dirty lines*: Invalidating all cache lines is inefficient because typically only a few lines are corrupted. Unnecessary compulsory misses degrade performance. A

good recovery heuristic is to invalidate only dirty lines. The approach does not recover A-stream state perfectly, but closely approximates complete recovery and performs almost as well as the previous method of identifying and flushing corrupted lines. A detailed rationale for this heuristic can be found elsewhere [12].

In both the flush and flush-dirty schemes, not all flushed cache lines were corrupt. Sometimes, only a handful of lines contain incorrect data. A line is flushed by resetting its valid bit; its data and tag remain intact after flushing. Preserved data is exploited to reduce the impact of recovery-induced compulsory misses in the A-stream. When the A-stream references a flushed line that would otherwise be a hit in the cache, the preserved data in the cache line is used as a value prediction. The L2 cache is accessed to service the L1 cache miss. The prediction is validated when the L2 access completes, and in most cases the prediction is correct.

Chapter 3

Thesis Contribution: Implicit Back-propagation

Removal of ineffectual computation from the A-stream is an essential function in a slipstream processor, and this is handled together by the IR-detector and IR-predictor. The IR-detector monitors retired R-stream instructions and data, and informs the IR-predictor about instructions that can potentially be removed from the A-stream in the future. This information is tracked in the IR-predictor using confidence counters, and instructions are removed from the A-stream after their confidence counters reach a certain threshold.

The selection of past-ineffectual instructions by the IR-detector involves two steps. First, unreferenced writes, non-modifying writes, and correctly-predicted branches are selected for possible removal from the A-stream in the future. Second, instructions in the backward slices of unreferenced writes, non-modifying writes, and correctly-predicted branches are also selected, by back-propagation. The first step is simple because it can be implemented using a register-indexed table that is managed like a register rename table. The table tracks references to registers, and detects unreferenced writes and non-modifying writes accordingly. An optional address-indexed table tracks references to memory locations similarly. Correctly-predicted branches are identified by comparing predictions to outcomes, which is done in any case.

The complexity of the IR-detector depends on the back-propagation method – explicit (complex) or implicit (not complex). The alternative approaches are discussed at a high level in this chapter, in order to highlight the "thesis" of this work: the IR-detector can monitor what is logically the A-stream version of the program to reduce the problem of back-propagation to the detection of unreferenced writes. Chapters 4 and 5 describe each approach in detail.

3.1 Explicit Back-propagation

To implement explicit back-propagation, retired R-stream instructions are queued in an instruction buffer, and a configurable interconnection network is programmed to establish direct links between consumers and producers in the buffer. When an unreferenced write, non-modifying write, or correctly-predicted branch is detected, the instruction is selected for removal in the instruction buffer. From that point onward, the interconnection network handles back-propagation automatically. Ineffectual status is propagated from consumers to producers along the links that were established initially.

Dedicated links from consumers to producers allow all instructions in an ineffectual dependence chain to be selected for removal at the same time. We observe that an iterative approach may perform nearly as well, without the interconnection network, which requires many wires and logic gates.

3.2 Implicit Back-propagation

Our “thesis” is that the IR-detector does not need to explicitly implement back-propagation if *the reduced version of the program (A-stream) is monitored by the IR-detector*, instead of the unreduced version (R-stream). The IR-detector detects unreferenced writes, non-modifying writes, and correctly-predicted branches, as before. Repeated indications by the IR-detector cause the IR-predictor to eventually remove these ineffectual instructions from the A-stream. Once removed, *their producers become unreferenced writes*, because their consumers were removed from the A-stream. Unreferenced writes are detectable using the register-indexed table (and optional address-indexed table), as described earlier. Repeated indications by the IR-detector cause the IR-predictor to eventually remove these freshly exposed unreferenced writes from the A-stream. Their producers, in turn, are exposed as unreferenced writes in the A-stream. This process repeats until all instructions in an ineffectual dependence chain are removed.

In practice, the IR-detector still monitors retired R-stream instructions. However, instructions that were removed from the A-stream by the IR-predictor are specially marked with an “R-bit”. The IR-detector uses the R-bit to hide a consumer instruction from its producers, and thereby expose its producers as unreferenced writes.

Chapter 4

IR-detector: Using Explicit Back-propagation

This chapter describes an IR-detector based on explicit back-propagation. Retired R-stream instructions are buffered. Buffering is finite, so the oldest instructions leave the buffer as new instructions are added. Status of the exiting instructions is used to update the IR-predictor. Within the buffer, complex circuits are dynamically configured to link consumer instructions with their producer instructions. In other words, the buffer is a literal hardware implementation of a reverse data flow graph (R-DFG). As new instructions are merged into the R-DFG, the IR-detector detects unreferenced writes, non-modifying writes, and correctly-predicted branches. These instructions are selected for removal. Ineffectual instructions in the backward slices of the selected instructions are also selected for removal. R-DFG connections facilitate back-propagation of ineffectual status from consumers back to producers.

4.1 Implementation

The IR-detector based on explicit back-propagation is shown in Figure 4.1. The operand rename table (ORT) plays two roles. First, it determines dependencies among instructions, and uses this information to link consumers to their producers as they are merged into the R-DFG (Section 4.1.1). Second, the ORT detects unreferenced and non-modifying writes (Section 4.1.2). The removal of instructions by back-propagation is handled autonomously by the R-DFG, via the links configured by the ORT (Section 4.1.3). If the B-bit of an incoming instruction is set, then the instruction is a correctly-predicted branch. If the R-bit is set, then the instruction was actually removed from the A-stream this time around, by the IR-predictor.

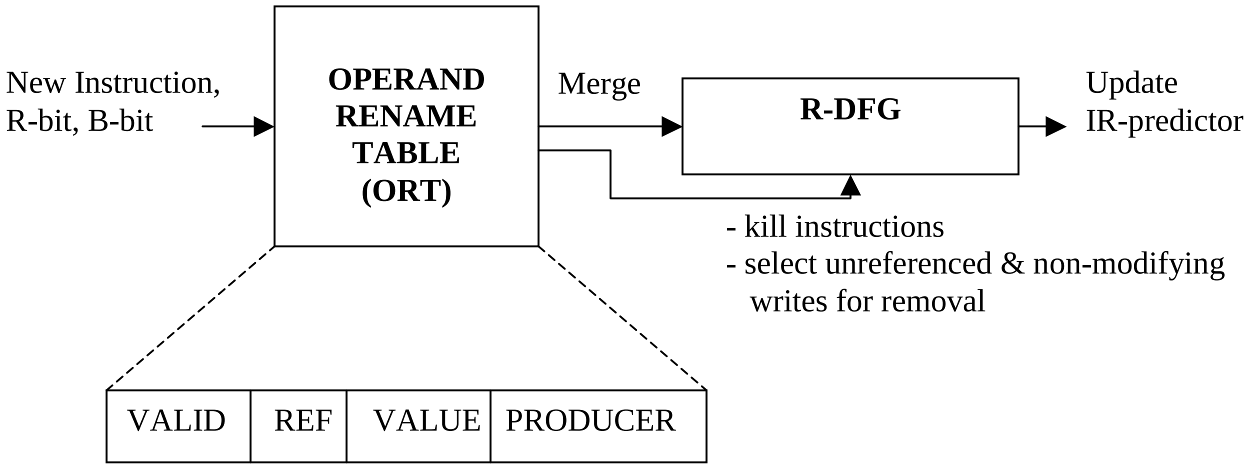


FIGURE 4.1 IR-detector based on explicit back-propagation.

4.1.1 Merging instructions into the R-DFG

The operand rename table (ORT) resembles a register renamer, but it tracks both registers and memory locations. Thus, there are actually two operand rename tables, one for registers and one for memory locations. The register ORT has as many entries as the number of architectural registers. The memory ORT is a cache-like structure, since the number of locations is potentially unbounded. When a new instruction is merged into the R-DFG, the following steps are performed.

- The source operand(s) are looked up in the ORT to get their most recent producer instructions, in order to link an instruction with its producer instructions in the R-DFG. If the *valid* bit is set, then the *producer* field in the ORT indicates the producer of the source operand.
- The *ref* bit corresponding to each source operand is set, indicating that the values have been referenced.
- The destination operand is looked up in the ORT to find the previous producer. The previous producer in the R-DFG is marked as "killed". (Note, however, that the previous producer is not killed in the case of a non-modifying write, as described in Section 4.1.2.)
- The ORT entry corresponding to the destination operand is updated with the latest producer and value. The *valid* bit is set. The *producer* field is updated with the R-DFG location where the incoming instruction is being placed. Finally, the *value* field is updated with the destination operand's value. (Note, however, that the update of the ORT entry occurs only after checking for unreferenced and non-modifying writes, as described in Section 4.1.2.)

4.1.2 Detecting the trigger instructions

Before updating the ORT entry corresponding to the destination operand (with the latest producer and value), the IR-detector checks for a non-modifying write or unreferenced write, in that order.

- If the *valid* bit is set and the value produced by the incoming instruction is the same as the *value* field in the ORT, then the incoming instruction is a non-modifying write and it is immediately selected for removal. In this case, the old producer is not killed and the ORT entry corresponding to the destination operand is not updated.
- If the *valid* bit is set and the *ref* bit is not set, then the previous producer is an unreferenced write. The previous producer, whose location in the R-DFG is indicated by the *producer* field, is selected for removal.

If the B-bit associated with the incoming instruction is set, then it is a correctly predicted branch and it is immediately selected for removal.

4.1.3 Explicit back-propagation using the R-DFG

The R-DFG autonomously selects ineffectual instructions by back-propagation. The R-DFG selects an instruction for removal if all of its dependent instructions are known and all of them are selected for removal. The R-DFG buffers instructions in the program order (FIFO), but also has a network of wires that connects each buffer entry to every other buffer entry. The network is configured to link consumer instructions to their producer instructions. The structure of the R-DFG is shown in Figure 4.2, and the logic within a single R-DFG entry is shown in Figure 4.3.

The R-DFG establishes connections from consumers to producers. A producer instruction may have as consumers any number of instructions after it in the R-DFG. So, a potential connection is required from every other R-DFG entry to the producer. The producer has a dedicated propagation bus, or p-bus, to support these potential connections. The producer is the receiver, or sink, of the p-bus. All other instructions in the R-DFG (potential consumers) are drivers, or sources, of the p-bus. Therefore, the p-bus has as many wires as the number of instructions in the R-DFG, less one (because the producer cannot depend on itself), so that each potential consumer can signal to the producer that there is dependence. A non-consumer drives its wire low to signal that it does not depend on the producer; a consumer instruction drives its wire high to signal that it depends on the producer. If all the wires in the producer's p-bus are low, then the producer does not have any consumers within the R-DFG; and, if the producer has also been killed by the ORT, then we know there are no consumers at all (it is ineffectual).

Because every instruction in the R-DFG is a potential producer instruction, each has its own dedicated p-bus. Therefore, as shown in Figure 4.2, for an N-entry R-DFG, there are N p-buses and each p-bus contains N-1 wires (a wire for each potential consumer).

Figure 4.3 shows the logic within a single R-DFG entry. The top of Figure 4.3 shows the logic for signaling the instruction's producers, via the p-buses. When the instruction is merged into the R-DFG, the ORT renames its source operands so that the instruction knows where its producers are located in the R-DFG. The instruction uses this information to assert its wire within the p-bus of each of its producers. The signals to its producers are deasserted if and when the instruction is selected for removal. As shown at the bottom of Figure 4.3, removal status gates the signaling mechanism.

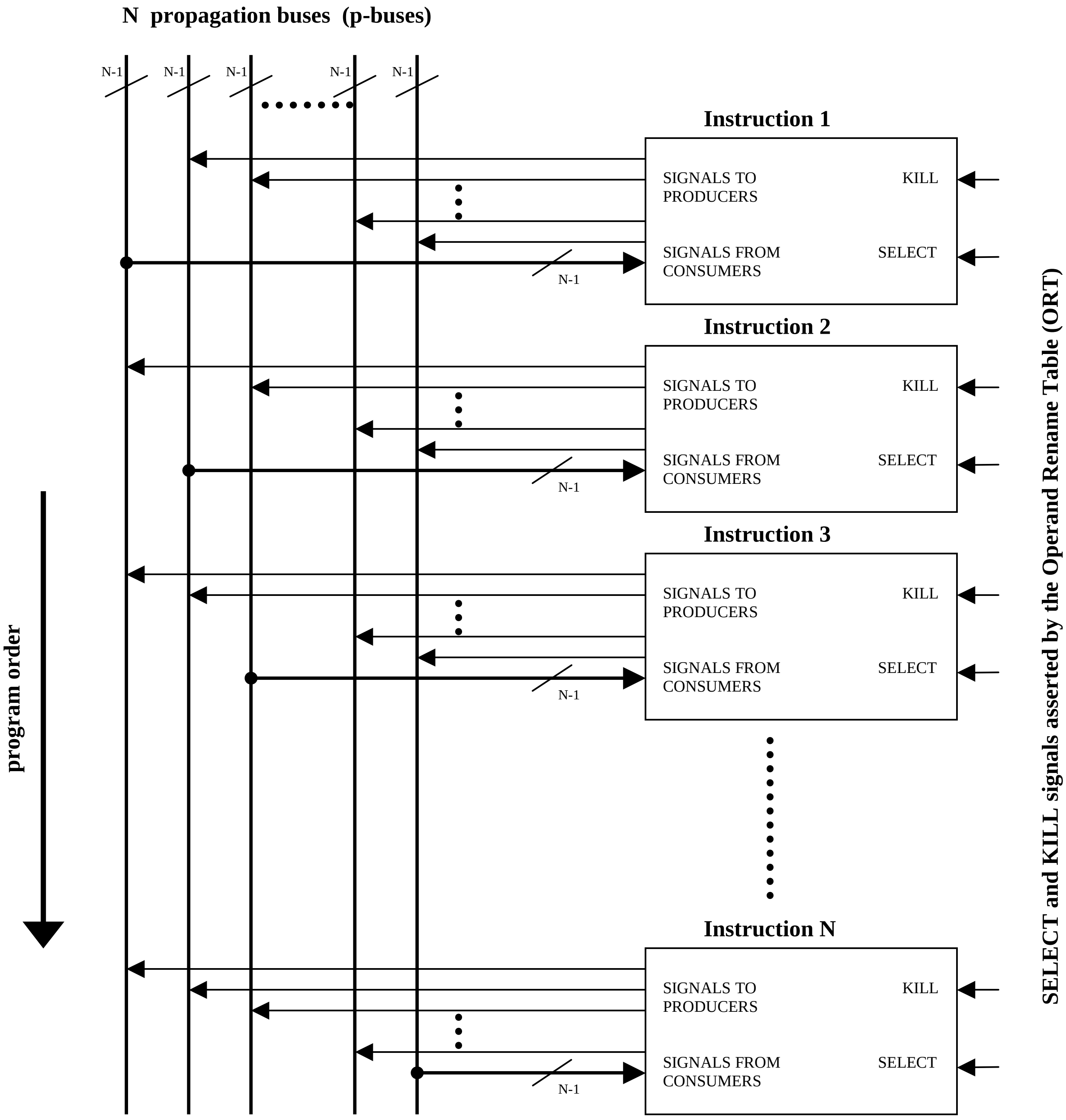


FIGURE 4.2 R-DFG circuit.

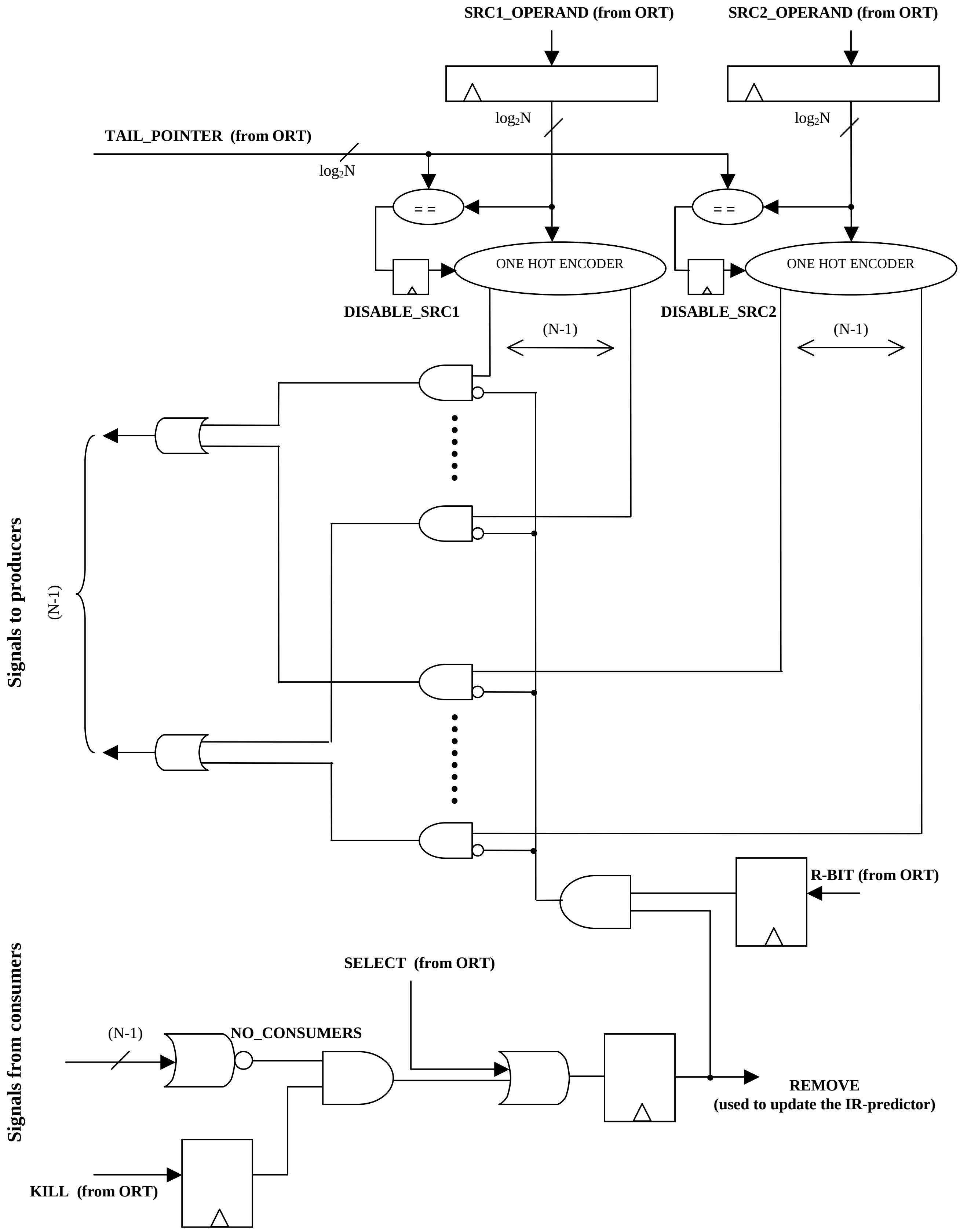


FIGURE 4.3 Single entry within the R-DFG. (N is the number of entries in the R-DFG.)

The logic at the bottom of Figure 4.3 selects instructions for removal by back-propagation. An instruction is selected for removal by back-propagation if the following three conditions are met.

- All of the dependent instructions are known, i.e., the instruction is killed by a subsequent instruction.
- All of the dependent instructions are selected for removal.
- All of the dependent instructions were removed from the A-stream (by the IR-predictor) this time around, i.e., the R-bits corresponding to all of the dependent instructions must be set.

The ORT sends a *kill* signal to an instruction when it is killed by a later instruction. The *kill* signal is latched to remember this fact. The second and third conditions are true when the *no_consumers* signal is high. When the second and third conditions are true, the *remove* and *R-bit* latches of all consumer instructions are high, and this prevents the consumer instructions from asserting their wires in the producer’s p-bus. The *no_consumers* signal of the producer is high as a result.

When an instruction exits the R-DFG, the state of the *remove* latch is used to update the IR-predictor. The latch gets set either through triggering conditions (*select* is asserted by the ORT due to an unreferenced write, non-modifying write, or correctly predicted branch), or through back-propagation (*no_consumers* and *kill* are high).

The third condition for back-propagation (R-bit) is needed because a producer instruction should know that its consumers are removed *in practice*, by the IR-predictor, and not just that the consumers are theoretically removable, which is what the IR-detector determines. The IR-detector may find that a producer and consumer pair is always theoretically removable. However, it is possible for their IR-predictor counters to get "*out of sync*". For example, aliasing may evict the consumer's counter. In this case the producer should not be removed, because the IR-predictor does not remove the consumer until its evicted counter gets saturated again.

The R-bit criterion is a simple method for implicitly synchronizing the counters of producers and consumers. Synchronizing counters is described in the next section, Section 4.2.

4.1.4 Handling producer instructions that leave the R-DFG

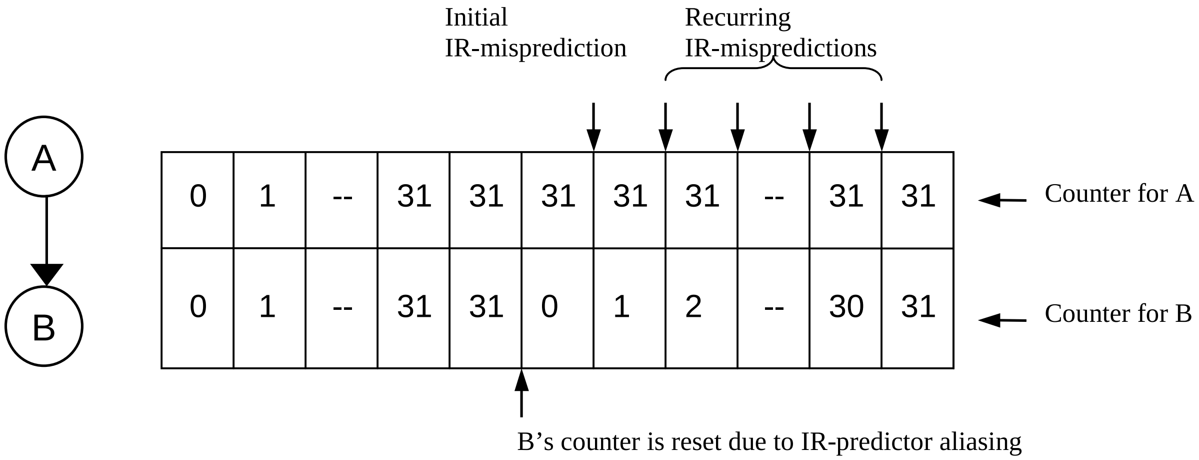
Two things need to be taken care of when a producer instruction leaves the R-DFG. First, the consumer instructions in the R-DFG should be notified that their producer instruction has left the window, so that they stop driving the p-bus corresponding to their producer. This is achieved by broadcasting the *tail_pointer* which points to the instruction leaving the window, and generating the *disable_src* signals as shown in Figure 4.3. Second, the ORT entry corresponding to the destination operand of the exiting producer should be made invalid. Otherwise, there is a possibility that the ORT may *kill* or *select* incorrect instructions, later on. One solution is to broadcast the *tail_pointer* in the ORT. ORT entries that have the *tail_pointer* in their producer fields are invalidated.

Alternatively, sequence numbers can be used to implicitly invalidate ORT entries. A global sequence number is incremented whenever the *tail_pointer* wraps around. The *producer* field of an operand in the ORT consists of the index of its producer in the R-DFG concatenated with the global sequence number at the time the producer instruction is merged into the R-DFG. The producer of an operand in the ORT is still within the R-DFG only if (1) the sequence number in the *producer* field is equal to the current global sequence number, and the index in the *producer* field is less than or equal to the *tail_pointer*, or (2) the sequence number in the *producer* field is equal to the previous global sequence number, and the index in the *producer* field is greater than the *tail_pointer*. So, only if either of these two conditions is satisfied, can the ORT *kill* or *select* an instruction in the R-DFG. A sufficiently large sequence number reduces the chance of incorrectly killing or selecting instructions in the R-DFG.

4.2 Synchronizing Counters

One of the criteria for selecting an instruction for removal by back-propagation is that all of its consumer instructions must have been removed from the A-stream *in practice*. This criterion implicitly synchronizes the counters of producers and consumers instructions, even if they get “out of sync”. The criterion is checked in the R-DFG by making sure that the R-bits of all of the consumer instructions are set before the producer is selected for removal.

IR-predictor aliasing is one way for producer-consumer counters to become unsynchronized. Aliasing occurs when the confidence counter for an instruction is evicted by some other instruction. Consider the example in Figure 4.4. Instruction B is a branch that is always predicted correctly, so it is always selected for removal by the IR-detector. Instruction B is the only consumer of instruction A. Using the first two criteria for back-propagation given in Section 4.1.3, instruction A will always be selected for removal (by the R-DFG), too.



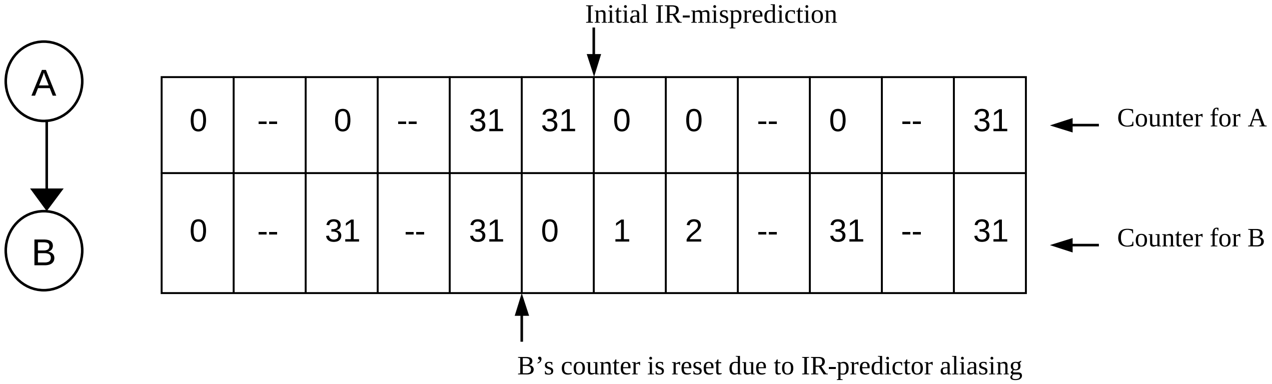
0	1	--	31	31	31	31	31	--	31	31
0	1	--	31	31	0	1	2	--	30	31

FIGURE 4.4 Recurring IR-mispredictions.

The counters of both instructions are incremented in lock-step, and they saturate at the value 31 at the same time. Future instances of both instructions are skipped in the A-stream. Now, suppose B's counter is displaced from the IR-predictor by another basic block (aliasing). When the basic block containing B is eventually re-allocated an entry in the IR-predictor, B’s counter is reinitialized to 0. Meanwhile, the counter corresponding to instruction A is still saturated. This leads to the removal of the producer but not the consumer, which causes an IR-misprediction. The IR-detector, unaware of this, keeps selecting both of the instructions for removal. Instruction A’s counter remains in saturation, but instruction B’s counter increments gradually from 0 to 31 (saturation). Until instruction B’s counter saturates again, the situation in which A is removed and B is not removed keeps repeating in the A-stream, leading to more IR-mispredictions. A single predictor aliasing event causes 31 IR-mispredictions, 30 of which could have been averted by re-synchronizing A’s counter with B’s counter.

The recurrence of IR-mispredictions caused by unsynchronized counters can be prevented if the third condition for back-propagation is also satisfied, in addition to the first two conditions. Instruction A is selected for removal by the IR-detector if instruction B is selected for removal *and* if B was actually removed by the IR-predictor in practice (R-bit = 1). Now, if instruction B’s counter is reset due to aliasing (or other scenarios), A’s counter will also be reset.

The same example is shown in Figure 4.5, the only difference being that all three conditions for back-propagation are considered. An instruction is selected for removal by back-propagation only if all of its dependent instructions are selected for removal and the R-bits of all of its dependent instructions are true. In this case, instruction B’s counter saturates first, and then future instances of instruction B are skipped in the A-stream. Instruction A’s counter starts incrementing only after instruction B’s counter is saturated (since instruction A waits for instruction B’s R-bit to be true). Future instances of instruction A are also skipped after its counter saturates. Now, suppose B’s counter is reset due to aliasing. This leads to the removal of instruction A but not instruction B, causing an IR-misprediction. The R-bit of instruction B is false, indicating to the IR-detector that instruction B was not removed from the A-stream this time. The IR-detector selects instruction B for removal. However, instruction A is not selected for removal because its consumer (instruction B) did not have its R-bit set. Instruction A’s confidence counter is reset as a result, and instruction B’s counter starts incrementing from zero. This prevents recurring IR-mispredictions due to the removal of a producer instruction and not its consumer instruction. A single predictor aliasing event causes only a single IR-misprediction due to unsynchronized counters.



0	--	0	--	31	31	0	0	--	0	--	31	Counter for A
0	--	31	--	31	0	1	2	--	31	--	31	Counter for B

FIGURE 4.5 Preventing recurring IR-mispredictions caused by unsynchronized counters.

Other scenarios besides IR-predictor aliasing can cause counters to become unsynchronized, requiring the R-bit to re-synchronize them. Consider the program snippet shown in Figure 4.6 and assume the R-bit condition for back-propagation is not used. The program takes path 1 initially. Instruction B is a highly predictable branch and it is the only consumer of instruction A (on path 1). Therefore, both instructions A and B are selected for removal by the IR-detector. The confidence counters are incremented in lock-step, and they saturate at the same time. Future instances of both instructions A and B are removed from the A-stream. Now, suppose the program takes path 2. Instruction C is a highly predictable branch and it is the only consumer of instruction A (on path 2). Instruction A will be removed from the A-stream since its counter is saturated. But, instruction C will not be removed since the counter value is 0. As a result, branch C will be evaluated incorrectly, leading to an IR-misprediction. The IR-detector is unaware of this. It keeps selecting both instructions A and C for removal. Instruction A’s counter remains saturated, while instruction C’s counter increments from 0 to 31. Until instruction C’s counter saturates, the IR-misprediction keeps repeating.

Here again, using the R-bit condition for back-propagation prevents the recurrence of IR-mispredictions caused by unsynchronized counters. When the program takes path 2 for the first time, instruction A will be removed without removing instruction C. This causes an IR-misprediction. But now that instruction C’s R-bit is false, indicating that it was not removed from the A-stream, instruction A is not selected for removal until instruction C’s counter saturates again. This resets instruction A’s counter, avoiding recurring IR-mispredictions.

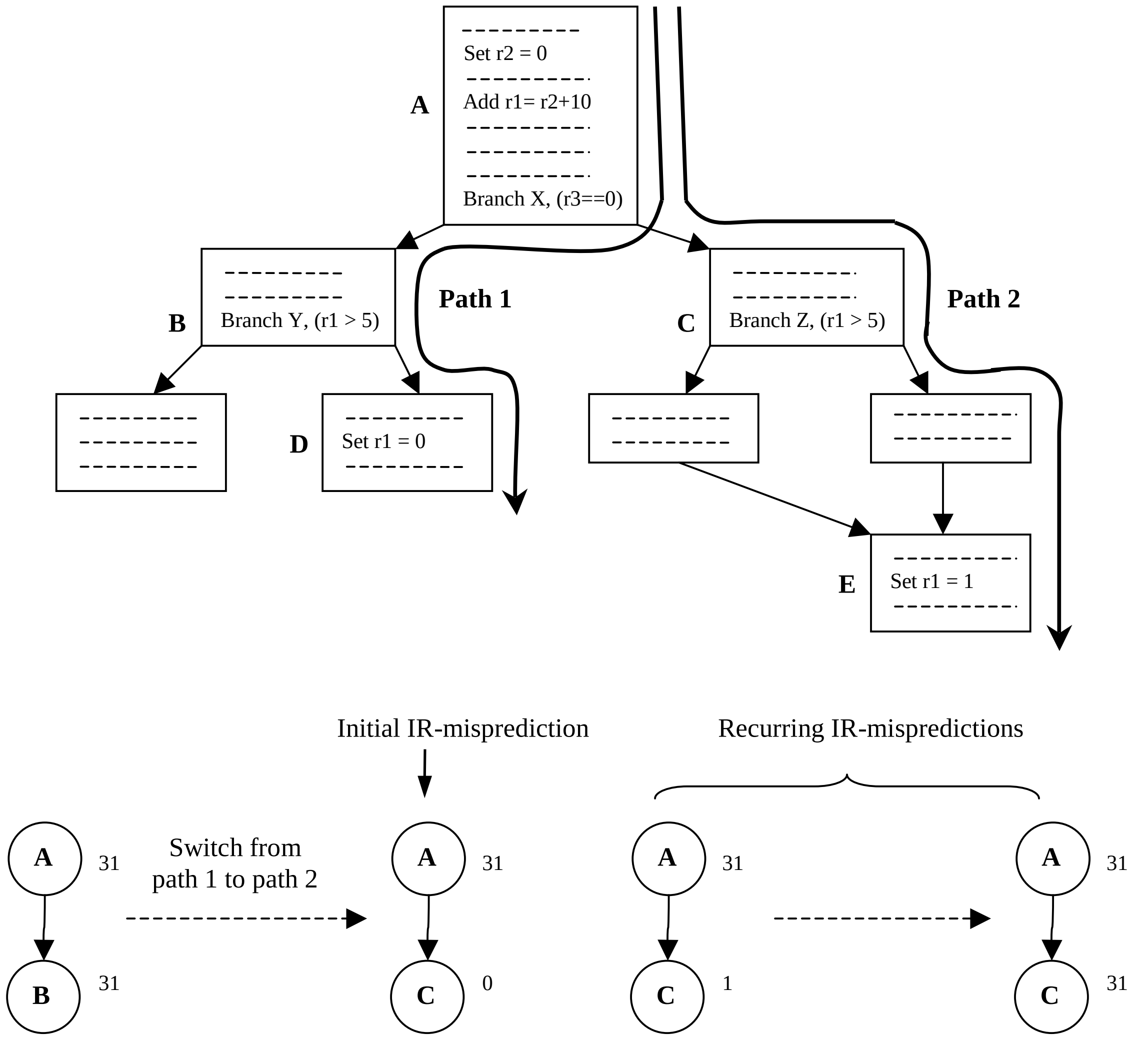


FIGURE 4.6 Another example of recurring IR-mispredictions caused by unsynchronized producer and consumer counters.

4.3 Characterizing Circuit Complexity

The register ORT is similar to a map table used in register renaming. It has as many entries as the number of architectural registers. Each field in the entry is stored in a separate physical structure, and the following are the port requirements for a 4-way superscalar processor.

- The *producer* field requires 12 read ports and 4 write ports. The read ports are for reading out producers of the 8 source operands (for setting up links to producer instructions in the R-DFG), as well as the previous producers of the 4 destination operands (for killing and possibly selecting previous producers in the R-DFG). The write ports are for updating destination operands with new producers.
- The *valid* field also requires 12 read ports and 4 write ports. The read ports are for checking whether the 8 source operands and 4 destination operands have valid producers in the R-DFG. The write ports are for setting the valid bits of the 4 destination operands.
- The *ref* field requires 4 read ports and 12 write ports. The read ports are for detecting whether or not the previous producers of the 4 destination operands are unreferenced writes. The write ports are for setting the *ref* bits of the 8 source operands, and for resetting the *ref* bits of the 4 destination operands.
- The *value* field requires 4 read ports and 4 write ports. The read ports are for detecting non-modifying writes of the 4 destination operands, and the write ports are for updating destination operands with the latest values.

A maximum of four instructions in the R-DFG can be killed or selected for removal by the ORT every cycle. The ORT has logic to detect four ineffectual instructions per cycle based on triggering criteria, and decodes four producer IDs per cycle to assert the appropriate *select* and *kill* lines in the R-DFG. The memory ORT has similar logic, but requires a cache-like structure. The memory ORT has 4 fewer read ports for the *produce*r and *valid* fields and 4 fewer write ports for the *ref* field, compared to the register ORT.

The R-DFG is essentially a reverse data flow graph constructed in hardware, with physical connectivity from consumer instructions to their producers. As described in Section 4.1.3, for an R-DFG that can buffer N instructions, there are N propagation buses to signal dependences. And, each of the propagation buses contains (N-1) lines. Therefore, the number of wires in the R-DFG back-propagation network is N(N-1). Each entry within the R-DFG contains the following: the renamed source operands, the *R-bit* latch, the *kill* latch, and the *remove* latch. The primary combinational logic complexity includes two source operand decoders for driving the p-bus signals, and the (N-1)-input NOR gate for generating the *no_consumers* signal.

Chapter 5

IR-detector: Using Implicit Back-propagation

As described in Chapter 4, explicit back-propagation requires a complex configurable interconnection network, the R-DFG. *Implicit back-propagation*, on the other hand, does not require the R-DFG. The key innovation is to *logically* monitor the A-stream for past-ineffectual instructions, instead of the R-stream. As before, the IR-detector selects unreferenced writes, non-modifying writes, and correctly-predicted branches for possible removal from the A-stream in the future. After repeated indications by the IR-detector, the IR-predictor will remove these trigger instructions from the A-stream. Once the trigger instructions are removed, their producers become unreferenced writes in the A-stream. These freshly exposed unreferenced writes are themselves eventually removed, exposing more unreferenced writes, and the process continues iteratively. Eventually, entire non-essential dependence chains are removed from the A-stream. Logically monitoring the A-stream eliminates the need for an explicit back-propagation network. Instead, the ORT implicitly handles back-propagation through the detection of (newly-exposed) unreferenced writes, which is done in any case.

5.1 Implementation

The IR-detector based on implicit back-propagation is shown in Figure 5.1. Earlier, we implied that the IR-detector processes retired A-stream instructions. In fact, it processes retired R-stream instructions, as before. However, the R-bit is used to “extract” the A-stream from the R-stream. Recall, the R-bit of the incoming instruction indicates whether it was removed from the A-stream by the IR-predictor this time around. An instruction whose R-bit is set is not part of the A-stream, and can be hidden from its producers. The B-bit is set if the incoming instruction is a correctly predicted branch. And, as before, the operand rename table (ORT) selects instructions for removal based on the three triggering conditions: unreferenced writes, non-modifying writes, and correctly-predicted branches. There is no longer an R-DFG because back-propagation is reduced to the detection of unreferenced writes. The ORT handles unreferenced writes indiscriminately. The R-DFG is replaced by a FIFO instruction queue, which is needed only to update the IR-predictor in program order. Each entry in the FIFO contains a single bit, indicating whether or not the instruction has been selected for removal by the IR-detector.

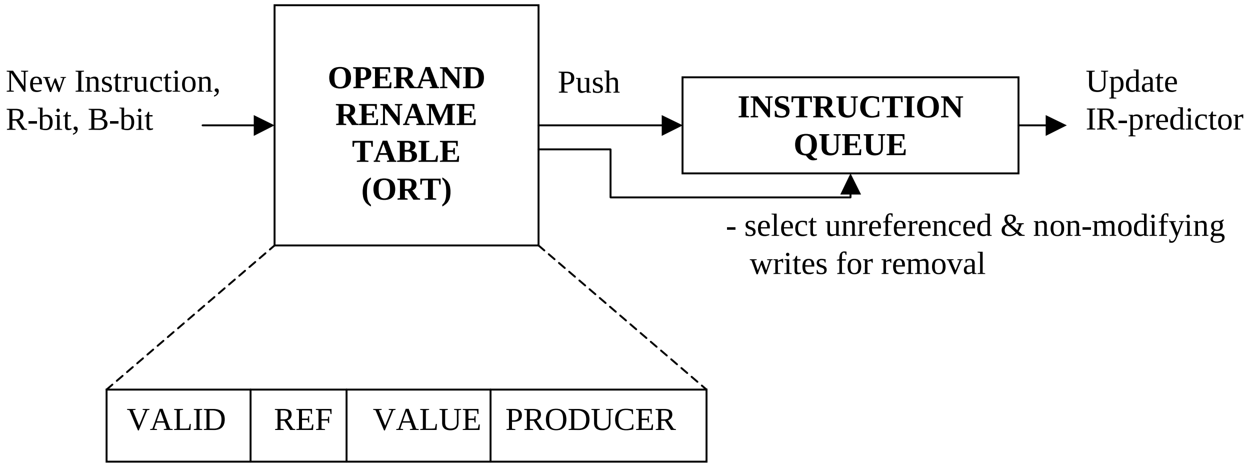


FIGURE 5.1 IR-detector based on implicit back-propagation.

5.1.1 Pushing instructions into the queue

As before, the operand rename table (ORT) tracks references to both registers and memory locations. New instructions are pushed into the instruction queue after the following steps are performed.

- The *ref* bit corresponding to each source operand is set *only if the incoming instruction's R-bit is not set,* indicating that the values were referenced in the A-stream. Not setting the *ref* bits implicitly performs back-propagation, and is described further in Sub-section 5.1.3. Note, the *ref* bits are set unconditionally in the previous approach.
- The ORT entry corresponding to the destination operand is updated with the latest producer and value. The *producer* field is updated with the location in the instruction queue where the incoming instruction is being placed. The *value* field is updated with the incoming instruction's result. Note, the update of the ORT entry occurs only after checking for unreferenced writes and non-modifying writes, as described in Sub-section 5.1.2.

5.1.2 Detecting trigger instructions

Trigger instructions are detected the same way as before. Unreferenced and non-modifying writes are detected by the ORT, as follows. Before updating the ORT entry corresponding to the destination operand (with the latest producer and value), the IR-detector checks for a non-modifying write or unreferenced write, in that order.

- If the *valid* bit is set and the value produced by the incoming instruction is the same as the *value* field in the ORT, then the incoming instruction is a non-modifying write and it is immediately selected for removal. In this case, the ORT entry corresponding to the destination operand is not updated.
- If the *valid* bit is set and the *ref* bit is not set, then the previous producer is an unreferenced write. The previous producer, whose location in the instruction queue is indicated by the *producer* field, is selected for removal.

If the B-bit associated with the incoming instruction is set, then it is a correctly-predicted branch and it is immediately selected for removal.

5.1.3 Implicit back-propagation

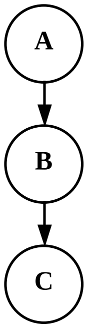
Back-propagation is handled implicitly by the ORT. Previously, as described in Chapter 4, the *ref* bit corresponding to each source operand was set unconditionally, i.e., the new instruction always "announced" to its producers that it was a consumer. Now, if the new instruction was actually removed from the A-stream *this time around* by the IR-predictor (R-bit is set), then the *ref* bits corresponding to its source operands are not set. This way, the consumer instruction makes itself invisible to its producer instructions speculatively, predicting that it will be selected for removal even this time. When one of its source operands is killed by a subsequent instruction, the ORT will observe that the old producer (indicated by the *producer* field) is an unreferenced write and select it for removal (if no other consumer set the *ref* bit). The *producer* field is used to index into the instruction queue and select the old producer for removal. Eventually, after reaching the counter 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 speculatively make itself invisible to its producer because its R-bit is set, and the 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 Figure 5.2. 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 associated with 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.

As described in the above example, if an instruction is removed from the A-stream (R-bit is set), it speculatively makes itself invisible to its producer instructions in the IR-detector. This assumes that the consumer instruction is ineffectual and will be selected for removal this time, too. However, the consumer instruction may not be ineffectual this time. This does not go unchecked. The implicit IR-detector can detect the transition of any instruction from ineffectual status to effectual status, because the destination operands of all instructions are always tracked for detecting unreferenced and non-modifying writes, and the B-bits of the branch instructions are always checked. Once an instruction changes status from ineffectual to effectual, the dependence chain leading up to it should also be made effectual. Unfortunately, this task takes multiple passes in the implicit IR-detector because we speculatively use the R-bit to implicitly back-propagate the status of consumer instructions to their producers. This may cause extra IR-mispredictions as discussed in Section 5.2. Fortunately, the R-bit based prediction is extremely accurate.

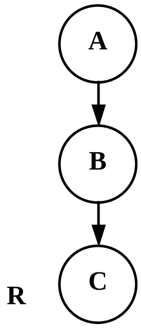
Stage 1. Instruction C is an unreferenced write.



ORT

	VALID	REF	VALUE	PRODUCER
R_A	1	1	V_A	A
R_B	1	1	V_B	B
R_C	1	0	V_C	C

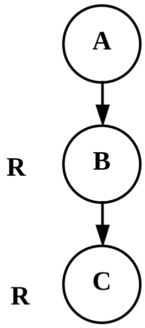
Stage 2. B and C are unreferenced writes. C is invisible to B (because its R-bit is set).



ORT

	VALID	REF	VALUE	PRODUCER
R_A	1	1	V_A	A
R_B	1	0	V_B	B
R_C	1	0	V_C	C

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



ORT

	VALID	REF	VALUE	PRODUCER
R_A	1	0	V_A	A
R_B	1	0	V_B	B
R_C	1	0	V_C	C

FIGURE 5.2 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.

5.1.4 Updating the IR-predictor

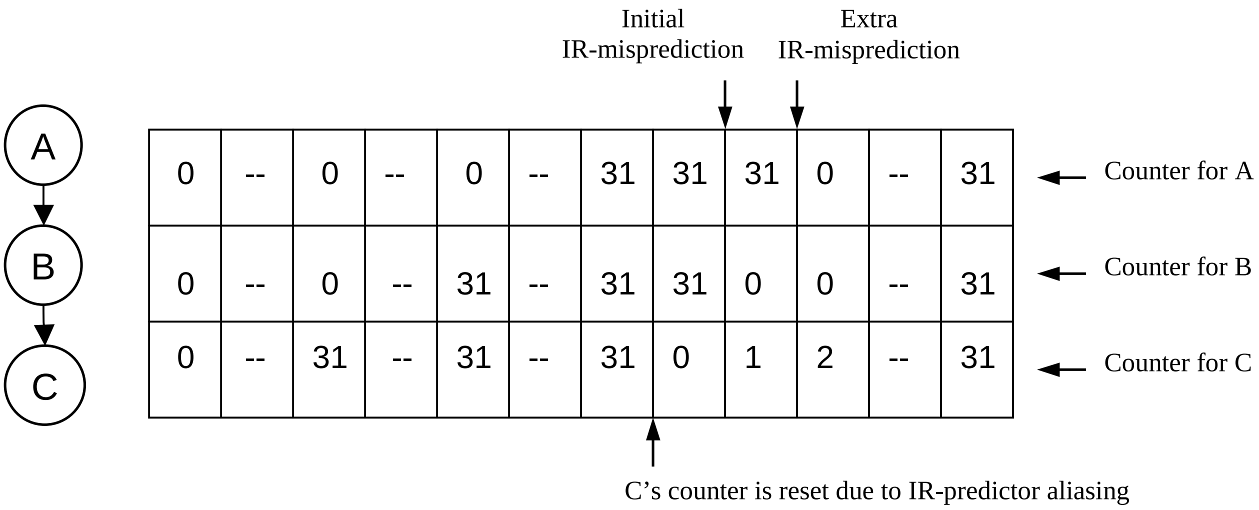
The function of the instruction queue is simply to update the IR-predictor in program order, which minimizes the number of IR-predictor accesses. Instructions are selected for removal by the ORT out of program order, depending on when registers (or memory locations) are killed. The instruction queue, however, ensures IR-predictor updates occur in program order. The oldest instructions are removed from the queue to make room for new instructions. The selection status of exiting instructions is used to update the IR-predictor. However, before updating the IR-predictor, exiting instructions are re-grouped into fetch blocks. This is necessary for efficient IR-predictor accesses. A single access updates all counters for the fetch block. Without the instruction queue, regrouping instructions into fetch blocks would be difficult.

5.1.5 Handling producer instructions that leave the instruction queue

The ORT entry corresponding to the destination operand of an exiting producer should be invalidated. Otherwise, the ORT may select incorrect instructions for removal. The solutions proposed in Section 4.1.4 for the explicit IR-detector also work for the implicit IR-detector. One solution is to broadcast the *tail_pointer* to all ORT entries. ORT entries that have the *tail_pointer* in their producer fields are invalidated. Alternatively, sequence numbers can be used, as described in Section 4.1.4. A sufficiently large sequence number reduces the chance of the ORT incorrectly selecting an instruction in the queue.

5.2 IR-mispredictions due to Unsynchronized Counters

As described in Section 4.2, IR-predictor aliasing and other events can cause the confidence counters of producer and consumer instructions to become unsynchronized, and this may lead to extra IR-mispredictions. For an IR-predictor that uses explicit back-propagation, the R-bit back-propagation criterion re-synchronizes the counters of an entire dependence chain in a single pass, limiting the number of IR-mispredictions to one per aliasing event. An IR-detector that uses implicit back-propagation, however, requires a lengthier re-synchronization time, and the number of IR-mispredictions per aliasing event is equal to the length of the dependence chain. Figure 5.3 shows the effect of a single aliasing event, in the context of implicit back-propagation.



0	--	0	--	0	--	31	31	31	0	--	31	Counter for A
0	--	0	--	31	--	31	31	0	0	--	31	Counter for B
0	--	31	--	31	--	31	0	1	2	--	31	Counter for C

FIGURE 5.3 Extra IR-mispredictions caused by unsynchronized counters.

Instruction C is a highly predictable branch and the only consumer of instruction B. Instruction B is the only consumer of instruction A. Taken together as a whole, the three instructions are ineffectual and can be selected for removal. The sequence of events is as follows. First, instruction C is selected for removal by the ORT. It is removed from the A-stream after its confidence counter saturates. Next, instruction B is selected for removal, since instruction C does not add itself as a consumer of B. Instruction B is also removed from the A-stream after its counter saturates. This causes instruction A to be selected for removal, since instruction B does not add itself as a consumer. Instruction A is also removed from the A-stream when its counter saturates. Ultimately, all three instructions are removed from the A-stream as a whole.

Now, suppose C's counter is displaced from the IR-predictor by another basic block due to aliasing. When the basic block containing instruction C is re-allocated an entry in the IR-predictor, instruction C's counter is reinitialized to zero. This means producer B is removed without also removing consumer C, which causes the first IR-misprediction. Next pass, when the IR-detector analyzes the sequence A, B, and C again, it finds that the R-bits corresponding to instructions A and B are set, but not the R-bit of instruction C. Instruction B does not add itself as a consumer of instruction A, since the R-bit of instruction B is set. However, instruction C adds itself as a consumer of instruction B, since instruction C's R-bit is not set. The outcome is that instruction B is not selected for removal and instruction B's counter is reset. Unfortunately, instruction A is still removed from the A-stream (because B still made itself invisible – its R-bit was set), without also removing instructions B and C. This leads to yet another (extra) IR-misprediction. Next pass, the IR-detector finds that the R-bit of instruction A is set, but not the R-bits of instructions B and C. Instruction C adds itself as a consumer of instruction B, and instruction B adds itself as a consumer of instruction A. Instruction A and B are not selected for removal as a result. None of the three instructions are removed from the A-stream. The learning process repeats and the whole dependence chain eventually gets selected for removal again. In general, a single event that causes counters to become unsynchronized results in multiple IR-mispredictions, equal in number to the chain length. Re-synchronizing takes multiple passes in the IR-detector as a direct consequence of using the R-bit speculatively.

5.3 Characterizing Circuit Complexity

The ORT is similar to the one used by the IR-detector based on explicit back-propagation (Section 4.3). The difference is that, for a 4-way superscalar processor, the ORT has 8 fewer read ports for the *producer* and *valid* fields, compared to the ORT in the explicit IR-detector. The reason is that we no longer need to explicitly link consumers to producers in the queue, so source operands do not need to be renamed to producer indices. This means that, for the implicit IR-detector, all ORT fields except the *ref* field have 4 read and 4 write ports: the *ref* field has 4 read and 12 write ports, as before. Another difference is that the ORT has to assert only the *select* lines to the instruction queue (the *kill* lines are not needed).

The instruction queue, which is a simple FIFO, is used to buffer the retired instructions in program order. There is no back-propagation network. Each FIFO entry contains a single bit, which indicates whether or not the instruction is selected for removal.

Chapter 6

Simulation Methodology

A detailed execution-driven simulator is used to study the slipstream processor [12]. The simulator models the architecture given in Figure 2.1. 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 re-synchronized whenever there is an IR-misprediction. The execution-driven simulator is validated by a functional simulator running independently and parallel with it. The functional simulator verifies retired R-stream control and data outcomes.

6.1 Microarchitecture Configuration

The slipstream microarchitecture parameters are listed in Table 6.1. The top-left portion lists parameters for 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-way 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 flushing dirty lines with value prediction [12]. Recovery of the register file takes 21 cycles.

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
Recovery	Memory: Flush dirty lines with value prediction		L2 hit = 12 cycles (min.)
	Register File: • Recovery latency = 21 cycles • 5 cycles to start up recovery pipeline • 4 reg. restores/cycle (total 64 regs)		L2 miss = 70 cycles (min.)
		# out. misses	Unlimited for all caches
		Duplication	Hardware-based memory duplication

TABLE 6.1 Microarchitecture configuration.

6.2 Benchmarks

A combination of SPEC2000 and SPEC95 integer benchmarks are used for the simulations, 12 benchmarks in all. 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 used and their input datasets are given in Table 6.2.

Benchmark	Suite	Input dataset
gap	SEPC2000	-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)

TABLE 6.2 Benchmarks and input datasets.

Chapter 7

Experimental Results

This chapter 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 effect 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 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.

7.1 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.

7.1.1 Explicit back-propagation

The performance of the slipstream processor with explicit back-propagation is shown in Figure 7.1. 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% to 12.3% 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% to 21.3%, 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% to 65.6% instruction removal), while there is a 58.5% 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 Figure 7.2. On average, %IPC improvement does not vary much with instruction buffer size. The peak IPC improvement of 12.3% occurs at a buffer size of 128 instructions. We conclude that the best configuration, on average, is a counter threshold of 64 and an instruction buffer size of 128 instructions, for the explicit IR-detector. This configuration is used for the remainder of the study.

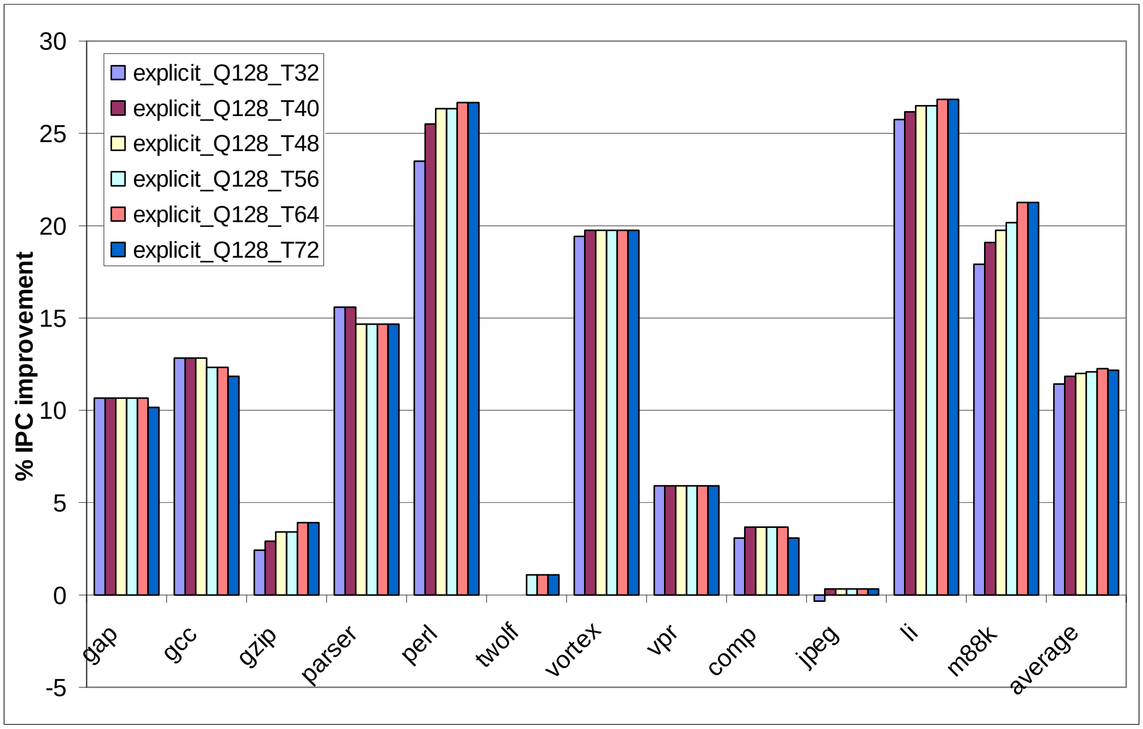


FIGURE 7.1 Performance of the slipstream processor with explicit back-propagation, for an instruction buffer size of 128 and counter threshold varying from 32 to 72.

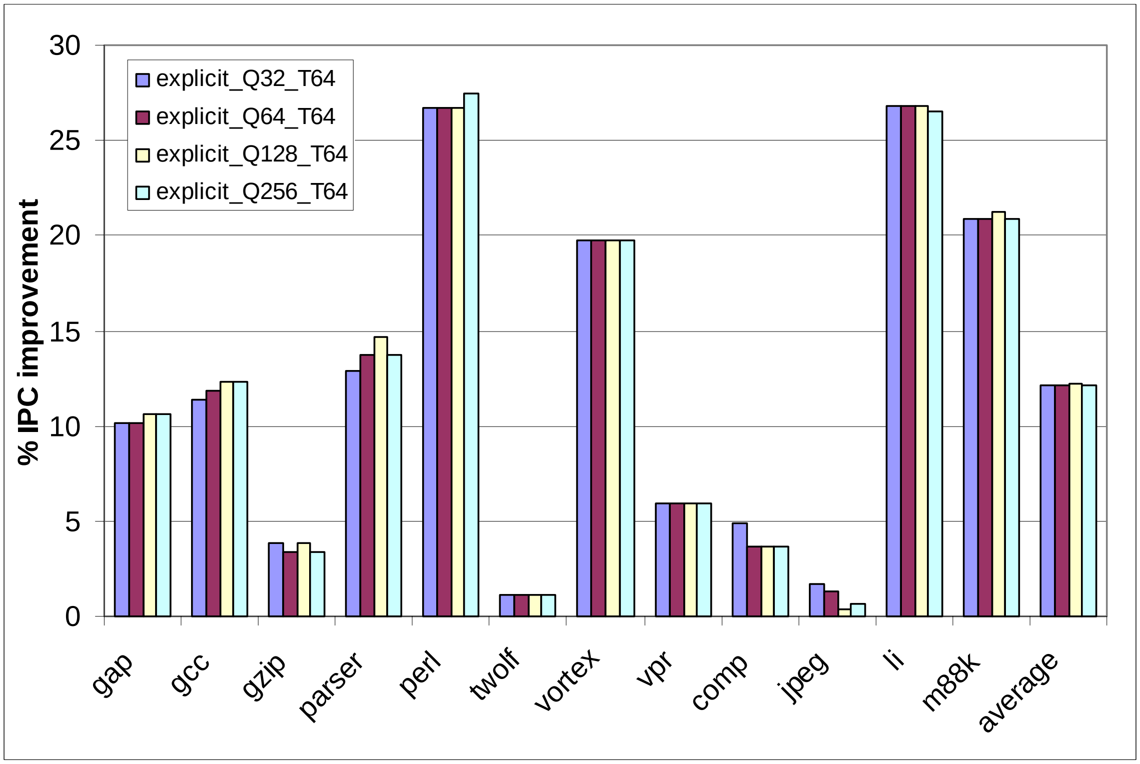


FIGURE 7.2 Performance of the slipstream processor with explicit back-propagation, for a counter threshold of 64 and instruction buffer size varying from 32 to 256.

7.1.2 Implicit back-propagation

The performance of the slipstream processor with implicit back-propagation is shown in Figure 7.3. As before, 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% to 11.8% 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% to 20.2%, 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% to 65.6% of instruction removal), while there is a 55.4% 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 Figure 7.4. On average, the %IPC improvement varies from 11.3% to 11.8%. The peak %IPC improvement of 11.8% occurs at an instruction buffer size of 128. Therefore, on average, the best counter threshold (64) and instruction buffer size (128) for the implicit and explicit IR-detectors are the same.

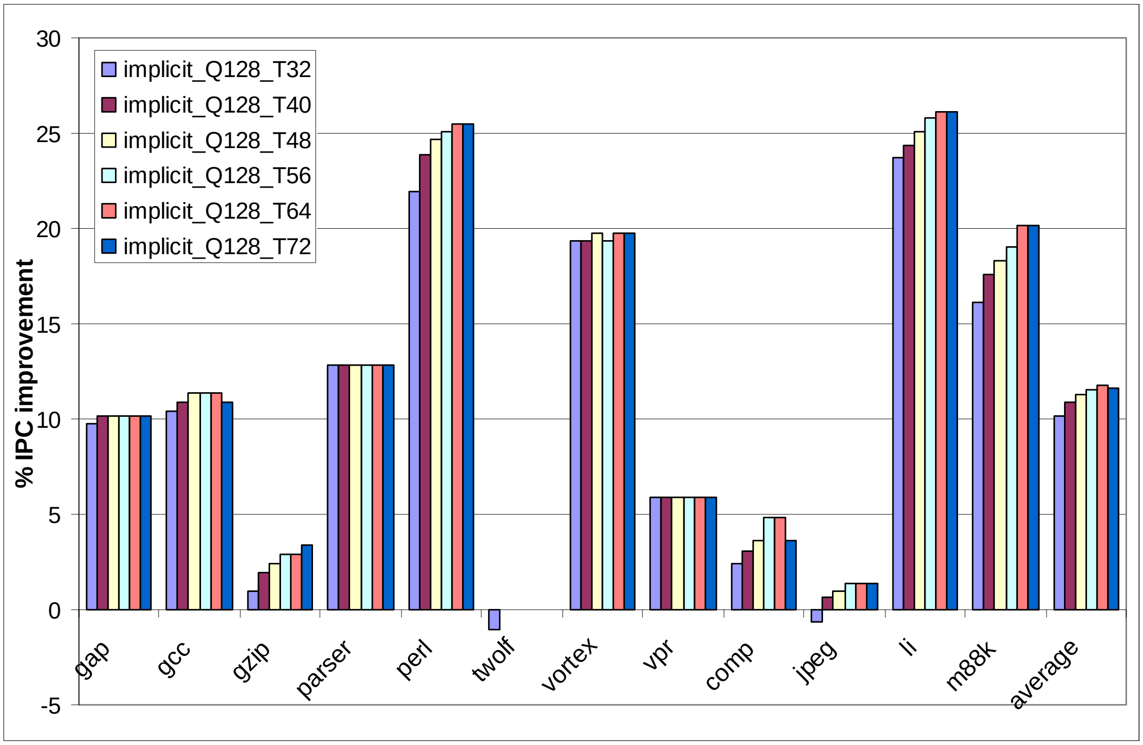


FIGURE 7.3 Performance of the slipstream processor with implicit back-propagation, for an instruction buffer size of 128 and counter threshold varying from 32 to 72.

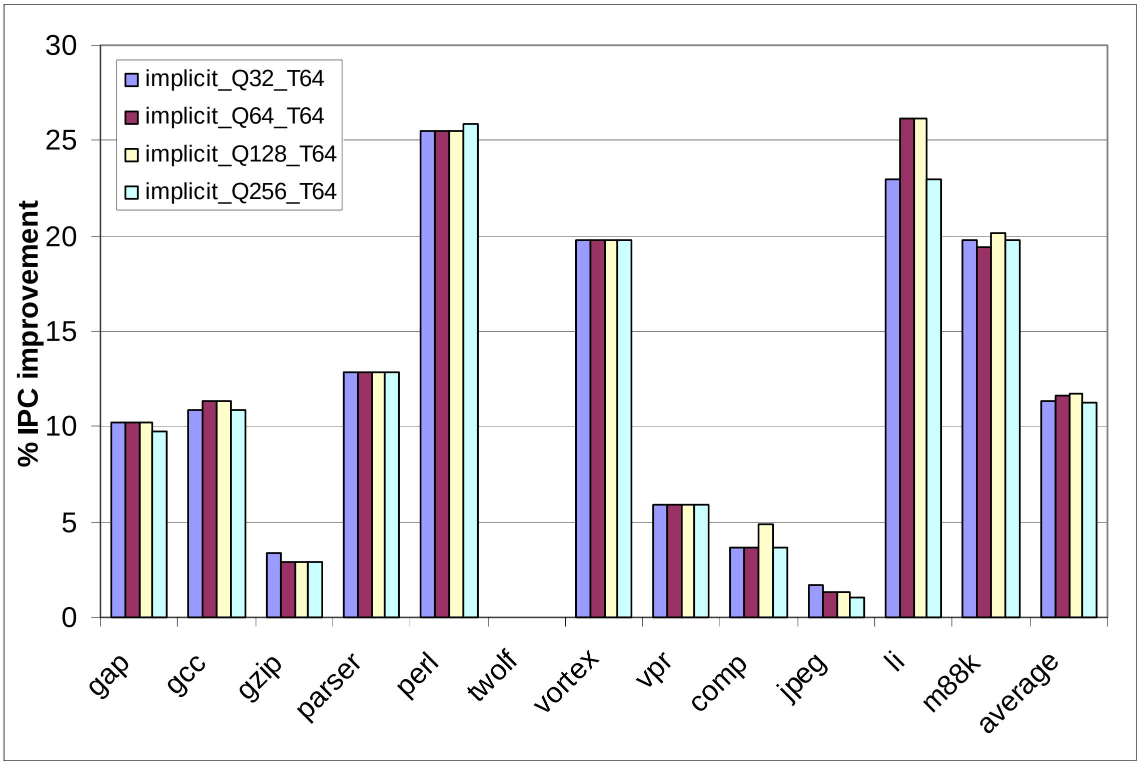


FIGURE 7.4 Performance of the slipstream processor with implicit back-propagation, for a counter threshold of 64 and instruction buffer size varying from 32 to 256.

7.2 Comparison of Explicit and Implicit IR-detectors

The comparison of slipstream processors with the best explicit (*explicit_Q128_T64*) and implicit (*implicit_Q128_T64*) IR-detectors is given in Figure 7.5. The *avg* bar represents the average %IPC improvement for all 12 benchmarks. The *avg_1/3* bar is the average %IPC improvement for the 6 benchmarks which have more than 1/3 instruction removal in the A-stream: *gcc*, *parser*, *perl*, *vortex*, *li*, and *m88ksim*. 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.

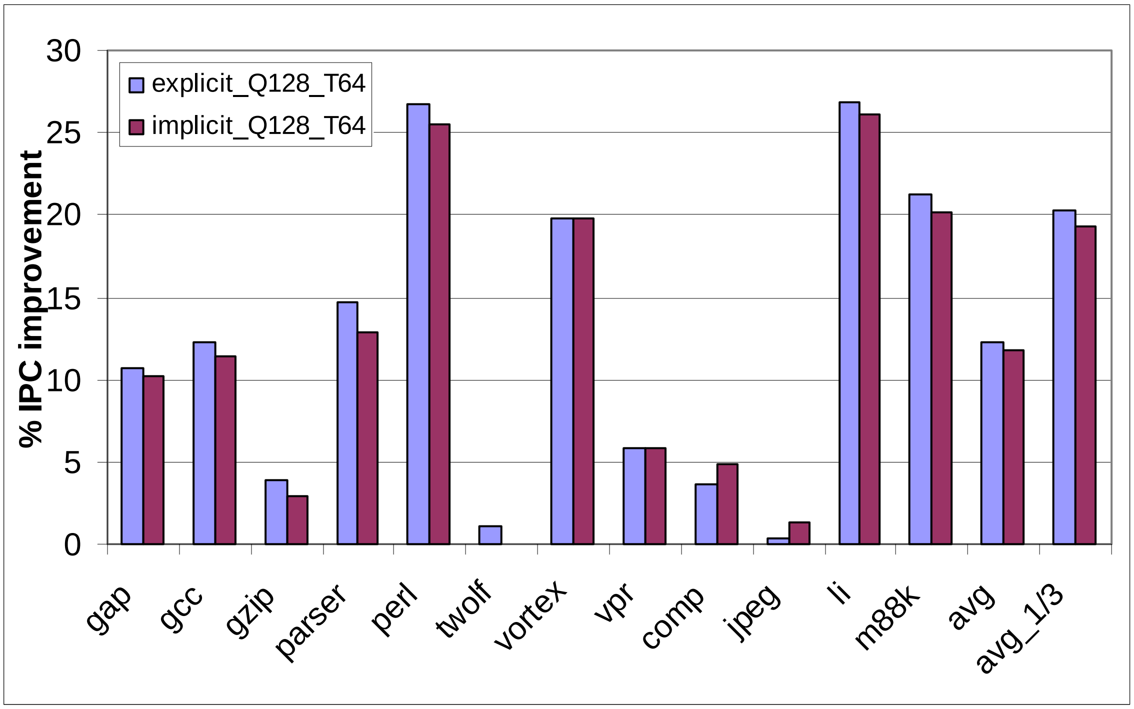


FIGURE 7.5 Comparison of the explicit and implicit IR-detectors.

The slipstream processor with the explicit IR-detector gives an average performance improvement of 12.3%, while the one with the implicit IR-detector gives an average performance improvement of 11.8%. The average performance improvement for the 6 benchmarks with more than 1/3 instruction removal is 20.3% with explicit back-propagation and 19.3% 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.

The slight performance difference between implicit back-propagation and explicit back-propagation may be due to two reasons. The first reason is that implicit back-propagation may remove fewer A-stream instructions than explicit back-propagation. The second reason is that implicit back-propagation may cause more IR-mispredictions than explicit back-propagation. Figure 7.6 shows the percentage of instruction removal is almost the same for both IR-detectors. But, there is a noticeable increase in the number of IR-mispredictions for implicit back-propagation, as shown in Figure 7.7, 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. As discussed in Chapter 5, when the last instruction in an ineffectual dependence chain causes an IR-misprediction, there is a cascade of N-1 additional IR-mispredictions (if the chain is N instructions long). On the other hand, the explicit IR-detector does not incur additional IR-mispredictions after the initial one.

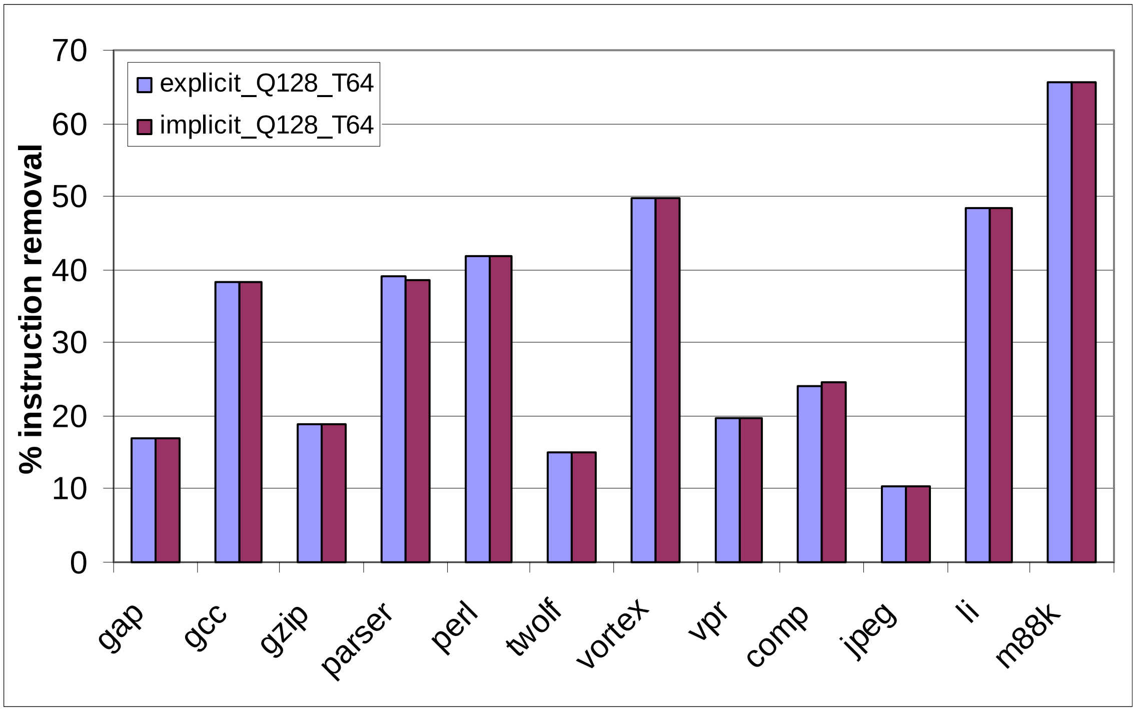


FIGURE 7.6 Comparison of the amount of instruction removal.

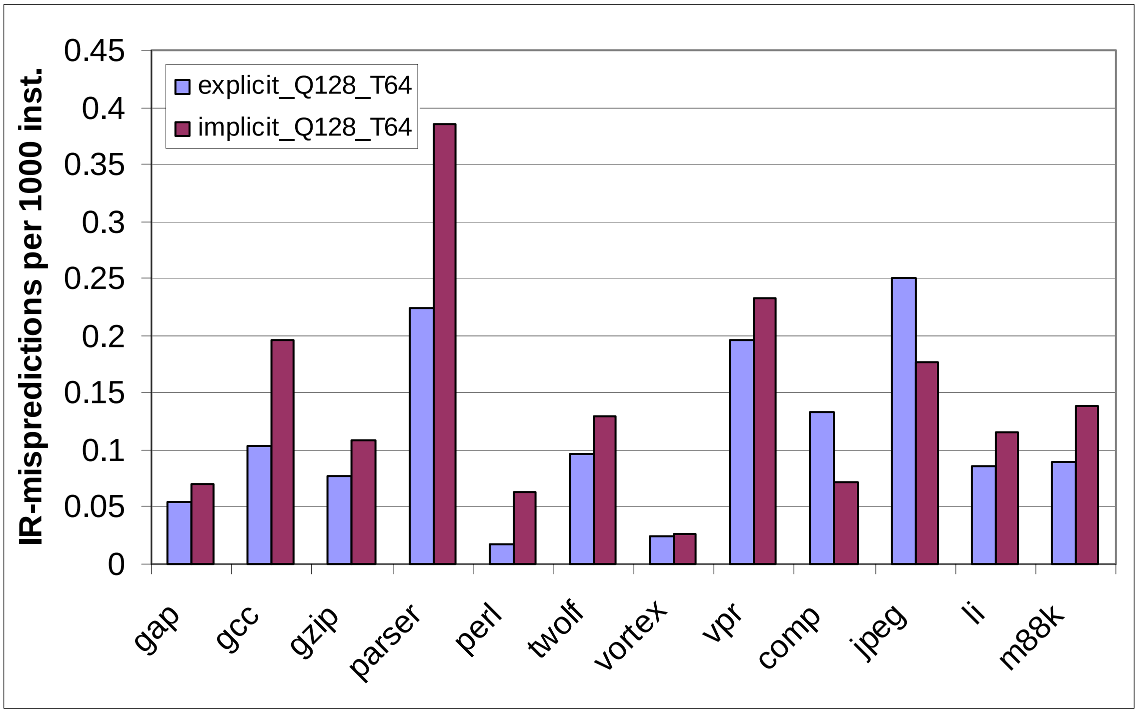


FIGURE 7.7 Comparison of IR-mispredictions per 1000 instructions.

7.3 Breakdown of Dynamic Instructions in the A-stream

The breakdown of dynamic instructions in the A-stream, is given in Figure 7.8. 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 non-modifying 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 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 Figure 7.8, branches, non-modifying writes, and back-propagation are the major sources of instruction removal. Original unreferenced writes are not as significant as the other three components.

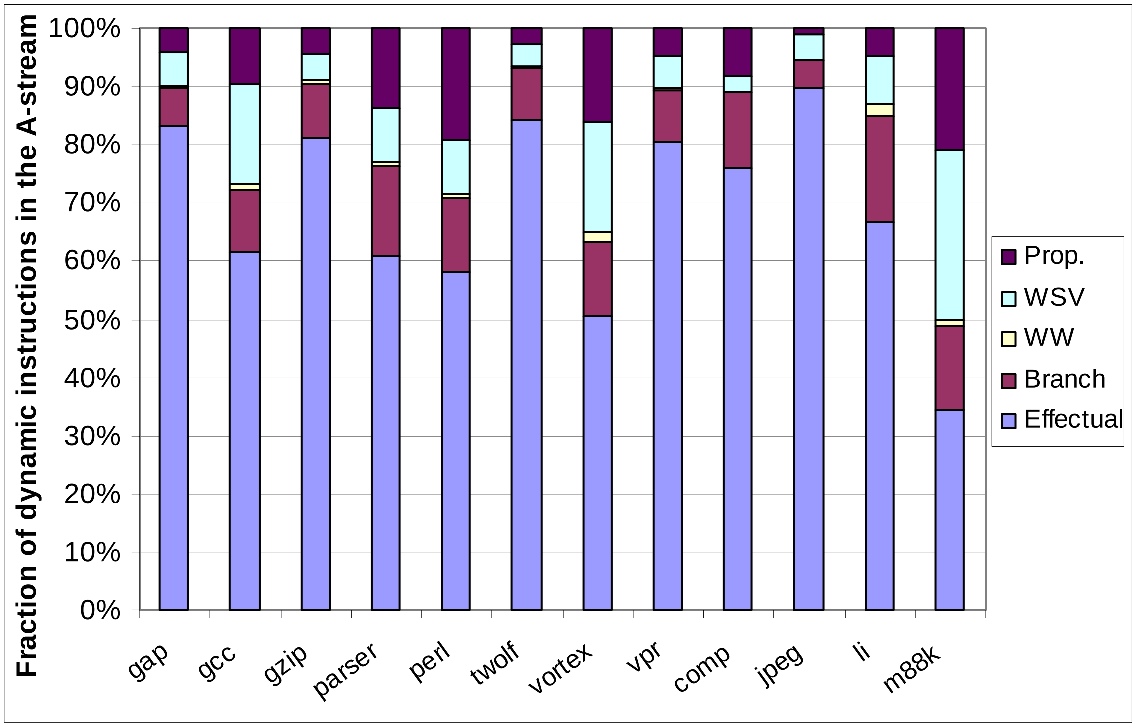


FIGURE 7.8 Breakdown of dynamic instructions in the A-stream.

7.4 Removal of Stores

An address-indexed memory operand rename table tracks references to memory locations. It is needed to detect and remove unreferenced stores, non-modifying stores, and stores that are in the backward slices of loads that were removed. The number of memory locations is potentially large, therefore, a cache-like structure is required. 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*) the removal of stores, is shown in Figure 7.9. The breakdown of instructions in the A-stream with and without the removal of stores is shown in Figure 7.10. (Suffix _*ns* indicates no removal of store instructions).

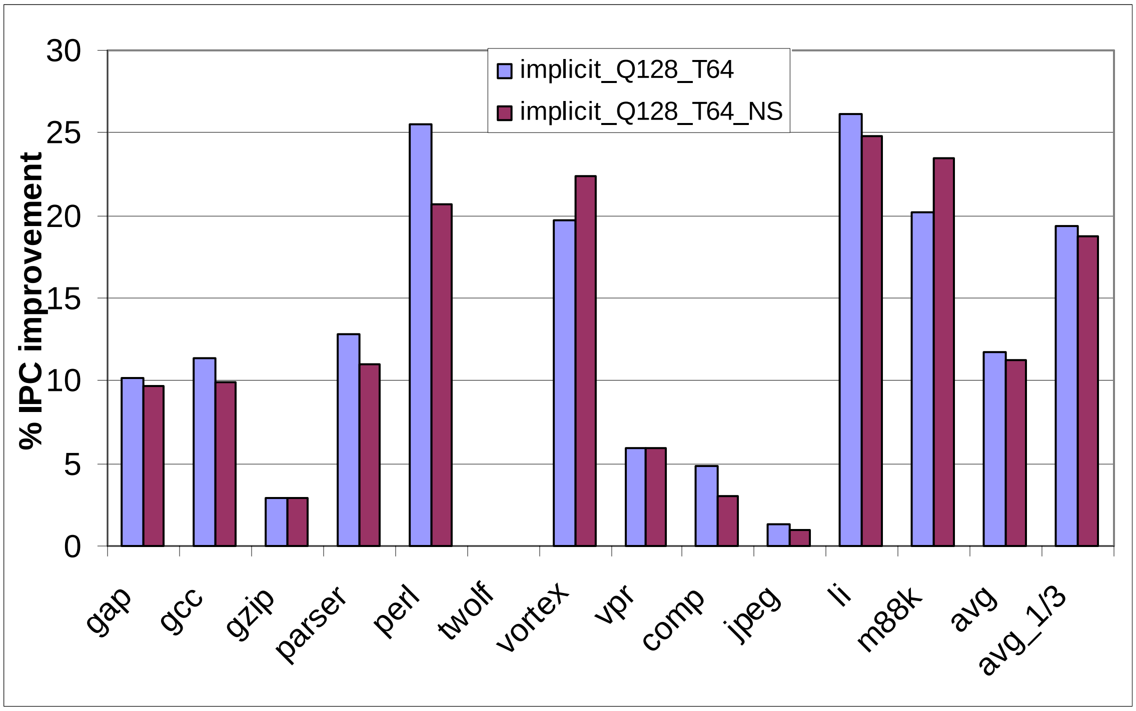


FIGURE 7.9 Effect of the removal of stores on slipstream processor performance.

The average slipstream performance improvement decreases from 11.8% to 11.2%, without store-related removal. The average slipstream performance decreases from 19.3% to 18.7% for the 6 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. Figure 7.10 shows a noticeable decrease in the number of instructions removed, the categories of non-modifying 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% in *m88ksim* and 15% 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.

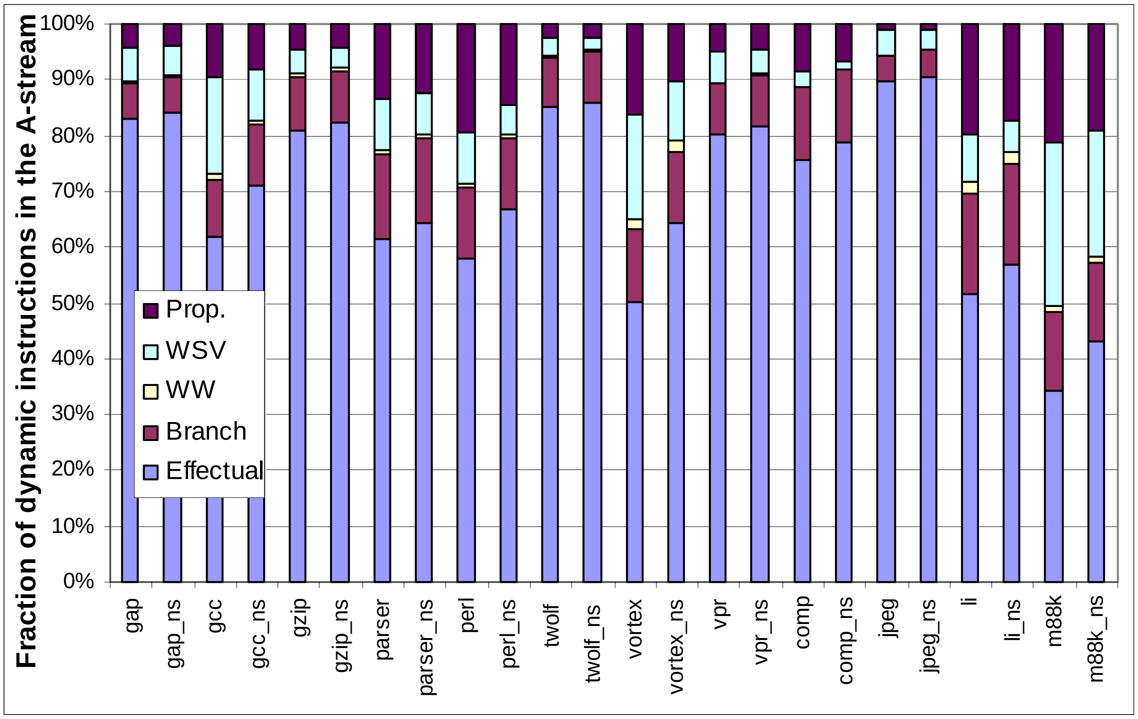


FIGURE 7.10 Breakdown of dynamic instructions in the A-stream, with and without the removal of stores. The implicit IR-detector is used.

Chapter 8

Related Work

Sundaramoorthy, Purser, and Rotenberg [11,13,16] proposed the first IR-detector for slipstream processors. They defined the criteria for instruction removal in slipstream processors. Key ineffectual instructions (unreferenced writes, non-modifying writes, and correctly-predicted branches) are selected for removal first, using an operand rename table (ORT). Ineffectual instructions in the backward program slices of the triggering instructions are selected using a reverse data flow graph (R-DFG). The R-DFG buffers retired instructions, and then establishes physical connections from consumer instructions to their producers. The implicit IR-detector proposed in this thesis inherits some aspects of the original explicit IR-detector, specifically, (1) the instruction-removal criteria, (2) the ORT, which determines data dependences and identifies unreferenced and non-modifying writes, and (3) the R-bit criterion, which is required to synchronize producer and consumer confidence counters. Yet, the new IR-detector significantly improves upon the old one. Implicit back-propagation eliminates the need for the complex back-propagation network (i.e., the R-DFG). In addition to implicit back-propagation, this thesis makes other novel contributions. Specifically, (1) we designed a gate-level implementation of the R-DFG to characterize its complexity, whereas previously there was no published gate-level design, and (2) we provided extensive results for both the new and old IR-detectors, whereas previously there were no published experiments documenting sensitivity to confidence threshold, buffer size, and removal of ineffectual stores.

Roth and Sohi [14,15] proposed Speculative Data-Driven Multithreading, an architecture for pre-executing threads to resolve likely-mispredicted branches early and prefetch possible cache misses. They do not propose a hardware mechanism for constructing pre-execution threads. Instead, they use an off-line 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 pre-execution to reduce the impact of performance-degrading instructions. They used profiling and manual analysis to construct the pre-execution threads.

Collins, Tullsen, Wang, and Shen [3] proposed a hardware mechanism for dynamically constructing pre-computation 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 backwards. 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 pre-computation architectures [e.g.,1,9] use similar approaches for dynamically constructing p-slices. Others have proposed compiler construction of p-slices [7].

Many retired instructions are "dropped" during the time that the RIB is busy analyzing a region of the dynamic instruction stream. Passing over dynamic instructions is not a problem in the context of pre-computation. 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, we simply wait until the region is seen again to construct its p-slice.

A slipstream processor, on the other hand, must update its IR-predictor for each and every dynamic instruction. Therefore, all dynamic instructions must pass through the IR-detector, and the IR-detector must keep up with instruction retirement. Scanning instructions serially and backward (as the RIB does) is simple, but it is also incompatible with the throughput requirement of IR-detectors. The implicit IR-detector is simple and provides high throughput. In fact, the implicit IR-detector is even simpler than backward scanning: the RIB moves forward to initially buffer instructions and then scans backward when a region is terminated; the implicit IR-detector moves forward continuously.

Another limitation of the RIB is that it uses a non-sliding analysis window, which is incompatible with the need to maximize detection of ineffectual instructions. There are two requirements for detecting an ineffectual dependence chain leading to a trigger instruction. First, many dynamic instructions before the trigger instruction need to be buffered. This ensures a large scope for identifying the dependence chain leading to the trigger instruction. Second, a large number of dynamic instructions after the trigger instruction need to be buffered. This increases the scope for killing values produced by instructions that lead up to the trigger instruction. And, both requirements are equally important for back-propagation. A non-sliding analysis window such as the RIB cannot effectively guarantee both of these requirements. Trigger instructions near the top of the non-sliding window have a small scope for identifying the dependence chains that feed them. And, instructions feeding trigger instructions located near the bottom of the non-sliding window have a small scope for getting killed. A sliding window analysis, like in the case of the IR-detector, is the best choice, since the act of sliding through the instruction stream ensures a large scope for analyzing dependence chains and also a large scope for killing values, and thereby gives the effect of an equally large window for all dynamic instructions.

Chapter 9

Summary and Future Work

9.1 Summary

The IR-detector detects past-ineffectual instructions in the R-stream, and selects them for possible removal from the A-stream in the future. The confidence counters of selected instructions are incremented in the IR-predictor, and the IR-predictor actually removes the instructions from the A-stream when their counters saturate.

The IR-detector uses a two-step selection process. First, it selects key trigger instructions -- unreferenced writes, non-modifying writes, and correctly-predicted branches. The operand rename table (ORT) can easily detect unreferenced and non-modifying writes. The second step, called back-propagation, selects computation chains feeding the trigger instructions.

An explicit implementation of back-propagation buffers retired R-stream instructions, and connects consumers to their producers using a configurable interconnection network. Consumers that are selected for removal use these connections to signal their producers, so that they get selected, too. This explicit signaling mechanism is the complex part, requiring a network of N(N-1) wires (where N is the instruction buffer size) and a complex logic block for each instruction in the buffer. The first gate-level design for the R-DFG was developed in this thesis, and presented in Chapter 4.

This thesis proposes a simpler implementation of back-propagation, called *implicit back-propagation*. The key idea is 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, non-modifying 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). After building up confidence, the freshly exposed unreferenced writes are also removed, exposing additional unreferenced writes. This process continues iteratively, until eventually entire non-essential dependence chains are removed. Logically monitoring the A-stream eliminates the need for an explicit back-propagation network. Instead, the ORT implicitly handles back-propagation through the detection of (newly-exposed) unreferenced writes, which is done in any case.

A slipstream processor with explicit back-propagation improves performance by an average of 12.3% (relative to conventional non-redundant execution), while a slipstream processor with implicit back-propagation improves performance by an average of 11.8%. Performance improvements are 20.3% and 19.3%, respectively, for benchmarks with more than 1/3 instruction removal. The difference in performance is due to the fact that implicit back-propagation incurs more IR-mispredictions than explicit back-propagation, as explained in Section 5.2. But, hardware complexity is significantly reduced with only moderate performance impact.

Removal of ineffectual stores requires a cache-like structure to track references to memory locations. The complexity of the IR-detector is further reduced by opting not to implement the removal of stores. The average performance improvement drops from 11.8% to 11.2% without the removal of stores. The average performance improvement drops from 19.3% to 18.7%, for benchmarks with more than 1/3 instruction removal.

9.2 Future Work

Some of the benchmarks (like *jpeg* and *twolf*) do not have enough instruction removal. Future work includes understanding why this is the case, and possibly defining additional instruction-removal criteria.

Currently, the slipstream processor removes the maximum number of instructions realistically possible. But, peak slipstream performance does not always occur with maximum instruction removal. For benchmarks like *m88ksim* and *vortex*, less-than-maximum instruction removal is preferred, because the A-stream provides more value predictions to the R-stream. Dynamic throttling of instruction removal has significant potential. Future work includes understanding the amount of instruction removal required for programs to achieve peak slipstream performance, and then dynamically throttling instruction removal accordingly.

In its present form, the slipstream processor relies on hardware mechanisms for identifying (IR-detector) and removing (IR-predictor) non-essential instructions from the A-stream. In future work, we would like to enlist the compiler to assist instruction removal. For example, the compiler can identify instructions that are likely to be repeatedly removable, so that IR-predictor resources are not wasted on these instructions. The compiler can assist bypassing instruction fetching by presenting the A-stream fetch unit with one or more compressed versions of the program. The compiler can also use more sophisticated analysis to increase dynamic instruction removal.

Finally, as part of future work, we would like to design the implicit IR-detector using a hardware description language, such as Verilog or VHDL. This design could be synthesized, and we could measure cost (die area) and critical path delay.

Bibliography

[1] M. Annavaram, J. Patel, and E. Davidson. Data Prefetching by Dependence Graph Precomputation. *28th International Symposium on Computer Architecture*, July 2001.

[2] D. C. Burger, T. M. Austin, and S. Bennett. The Simplescalar Tool Set, Version 2.0. Tech. Rep. 1342, Computer Science Department, University of Wisconsin-Madison, 1997.

[3] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic Speculative Precomputation. *34th International Symposium on Microarchitecture*, Dec. 2000.

[4] E. Jacobsen, E. Rotenberg, and J. Smith. Assigning Confidence to Conditional Branch Predictions. *29th International Symposium on Microarchitecture*, Dec. 1996.

[5] J. Kahle. Power4: A Dual-CPU Processor Chip. *Microprocessor Forum*, October 1999.

[6] K. M. Lepak and M. H. Lipasti. On the Value Locality of Store Instructions. *27th International Symposium on Computer Architecture*, June 2000.

[7] C.–K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. *28th International Symposium on Computer Architecture*, July 2001.

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

[9] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice Processors: An Implementation of Operation-Based Prediction. *15th International Conference on Supercomputing*, June 2001.

[10] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. *7th International Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[11] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A Study of Slipstream Processors. *33rd International Symposium on Microarchitecture*, Dec. 2000.

[12] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. Slipstream Memory Hierarchies. Tech. Rep. CESR-TR-02-3, Center for Embedded Systems Research, North Carolina State University, February 2002.

[13] E. Rotenberg. Exploiting Large Ineffectual Instruction Sequences. Tech. Rep., North Carolina State University, November 1999.

[14] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. *7th International Conference on High Performance Computer Architecture*, Jan. 2001.

[15] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. Tech. Rep. CS-TR-00-1414, University of Wisconsin-Madison, Feb. 2000.

[16] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[17] C. B. Zilles and G. S. Sohi. Execution-based Prediction Using Speculative Slices. *28th International Symposium on Computer Architecture*, July 2001.

[18] C. B. Zilles and G. S. Sohi. Understanding the Backward Slices of Performance-Degrading Instructions. *27th International Symposium on Computer Architecture*, June 2000.