Abstract

PURSER, ZACHARY ROBERT

Slipstream Processors

(Under the direction of Eric Rotenberg)

Processors execute a program's full dynamic instruction stream to arrive at its final output, yet there exist shorter instruction streams that produce the same overall effect. This thesis proposes creating a shorter but otherwise equivalent version of the original program by removing ineffectual computation and computation related to highly-predictable control flow. The shortened program is run concurrently with and slightly ahead of a full copy of the program on a chip multiprocessor (CMP) or simultaneous multithreading (SMT) processor. The leading program passes all of its control-flow and data-flow outcomes to the trailing program for checking. This redundant program arrangement provides two key benefits.

1) *Improved single-program performance*. The leading program is sped up because it retires fewer instructions. Although the number of retired instructions is not reduced in the trailing program, it fetches and executes instructions more efficiently by virtue of having near-oracle branch and value predictions from the leading program. Thus, the trailing program is also sped up in the wake or "slipstream" of the leading program, at the same time validating the speculative leading program and redirecting it as needed. Slipstream execution using two processors of a CMP substrate outperforms conventional non-redundant execution using only one of the processors. Likewise, given a sufficiently reduced leading program, slipstream execution using two contexts of an SMT substrate outperforms conventional non-redundant execution using only one of the contexts.

2) *Fault tolerance*. The shorter program is a subset of the full program and this partial redundancy is exploited for detecting and recovering from transient hardware faults. This does not require any additional hardware support, since the same mechanisms used to detect and recover from misspeculation in the leading program apply equally well to transient fault detection and recovery. In fact, there is no way to distinguish between misspeculation and faults.

The broader rationale for slipstream is extending, not replacing, the capabilities of CMP/SMT processors, providing additional modes of execution. This thesis demonstrates the feasibility and benefits of the slipstream execution model.

SLIPSTREAM PROCESSORS

by

ZACHARY ROBERT PURSER

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

COMPUTER ENGINEERING

Raleigh

2003

APPROVED BY:

_____

Dr. Eric Rotenberg, Chair of Advisory Committee

_____ _____

Dr. Gregory T. Byrd Dr. Thomas M. Conte

_____

Dr. S. Purushothaman Iyer

Biography

Zachary Robert Purser was born on the 31st of May, 1976, in Charlotte, North Carolina. He attended North Carolina State University where he majored in Computer Engineering receiving a B.S. in 1998 and an M.S. with a focus on robotics in 1999. He continued his graduate education at NC State pursuing his Ph.D. in computer architecture. He has developed the Slipstream Processor under the guidance of Dr. Eric Rotenberg.

Acknowledgments

First, I would like to thank my parents and my brothers for their constant support. Without their help, none of this would have been possible.

I would like to thank my advisor, Dr. Eric Rotenberg for being an incredible teacher and mentor. His excitement about microarchitecture is contagious and has kept me motivated throughout my work. He has an excellent writing style that is conversational and yet clearly expresses his ideas; his input has greatly improved my writing. I am very appreciative of all he has taught me.

I would also like to thank Dr. Tom Conte, Dr. Greg Byrd, and Dr. Purush Iyer for serving on my committee. Their comments, suggestions, and criticisms have greatly improved the quality of my work.

I would like to thank the members of my research group: Karthik Sundaramoorthy for many things, including helping me get started on slipstream; Jinson Koppanalil for his work on the IR-detector; Nikhil Gupta for listening to all of my crazy ideas about improving slipstream performance and supplying a few crazy ideas of his own; Aravindh Anantaraman for being able to fix almost anything and show me where to find almost anything; and all the other members of the group who have contributed so much in so many different ways.

Mark Toburen and Huiyang Zhou of the Tinker group have provided me with insight into microarchitecture and helped me navigate the labyrinth of paperwork in graduate school.

Rajeev Ramanath deserves special thanks for putting up with me during long simulations and breaks, and for always being willing to talk about politics.

I would like to thank my girlfriend Jen for her support and encouragement of my Ph.D. work. I hope to be equally supportive of her Ph.D. work.

I would also like to thank my friends Emily, Chris, Katie, and Susan. They have remained close friends even when my work schedule has kept us apart for long periods of time.

A special thanks to Anu Vaidyanathan. Her stubborn persistence is an inspiration and I am thankful for her constant support.

And a final thanks goes to Dr. Eddie Grant, the advisor for my master's degree. I would like to thank him not only for his support throughout my master's, but also for providing me with a lifelong love of robotics.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

The slipstream paradigm [36][41][52] is based on the observation that only a fraction of the dynamic instruction stream is needed for a program to make full, correct, forward progress. For example, some instruction sequences have no observable effect. They produce results that are not subsequently referenced, or results that do not change the state of the machine. And then there are instruction sequences whose effects are observable, but the effects are invariably predictable. Computation influencing control flow is the most notable example.

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

Therefore, a slipstream processor concurrently runs two copies of the program, leveraging either a single-chip multiprocessor (CMP) [34] or a simultaneous multithreading processor (SMT) [56][59]. (The user program is instantiated twice by the operating system and each copy has its own context.) One program always runs slightly ahead of the other. The leading program is called the *advanced stream*, or A-stream, and the trailing program is called the *redundant stream*, or R-stream. Hardware monitors the R-stream and detects (1) instructions that repeatedly and predictably have no observable effect (e.g., unreferenced writes, non-modifying writes) and

(2) branches whose outcomes are consistently predicted correctly. Future instances of the ineffectual instructions, branch instructions, and the computation chains leading up to them are speculatively removed in the A-stream — but only if there is high confidence correct forward progress can still be made, in spite of removing the instructions.

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

A key point is that the R-stream uses the outcomes it is checking as predictions [40]. This has two advantages.

- First, the R-stream fetches and executes more efficiently due to having near-ideal predictions from the A-stream. Thus, although the unreduced R-stream retires more instructions, it keeps pace with the A-stream and the two programs combined finish sooner than a single copy of the program would. The slipstream processor's approach of speeding up a single program via redundancy is analogous to "slipstreaming" in car racing, where two cars race nose-to-tail to increase the speed of *both* cars [39].
- Second, by using A-stream outcomes as predictions, *the R-stream leverages existing speculation mechanisms for checking the A-stream*. Conventional processors typically have mechanisms in place to check control flow speculation, and future processors may incorporate value prediction and mechanisms to check data flow speculation.

An analogy to the slipstream paradigm (and the source of its name) is "slipstreaming" in stock-car racing (e.g., NASCAR) [39]. At speeds in excess of 190 m.p.h., high air pressure forms at the front of a race car and a partial vacuum forms behind it. This creates drag and limits the car's top speed. A second car can position itself close behind the first (a process called slipstreaming or drafting). This fills the vacuum behind the lead car, reducing its drag. Likewise, the trailing car has less wind resistance in front. As a result, both cars speed up by several m.p.h.: the two combined go faster than either can alone.

In addition to potential performance improvements, slipstreaming provides fault-tolerant capabilities. The trends of very high clock speeds and very small transistors may make the entire chip prone to transient faults [45], and there is renewed interest in fault-tolerant architectures for commodity, high-performance microprocessors [3][38][40].

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

1.1 Understanding slipstream: Why it works and what are its limits

We present two different interpretations of slipstreaming to better understand the paradigm. In Section 1.1.1, the A-stream is considered to be the "main" thread and the R-stream "assists"

the A-stream. In Section 1.1.2, roles are reversed, where the R-stream is considered to be the "main" thread and the A-stream "assists" the R-stream. Actually, the two programs in a slipstream processor are functionally equivalent and mutually beneficial, so either interpretation is valid.

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

1.1.1 R-stream: A fast checker

The A-stream does not explicitly derive any performance benefit from the R-stream. Rather, the R-stream checks (and occasionally redirects) the A-stream without slowing it down. This is possible because *checking is inherently parallel* [27][40]. The R-stream is not bound by control-flow and data-flow dependences for which the A-stream has produced correct outcomes. As depicted in Figure 1-1, *the R-stream is a fast checking assist to the A-stream* [40][41][3].

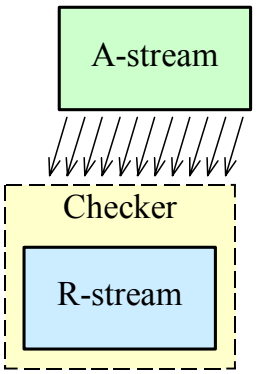

Figure 1-1. A fast checking assist to the A-stream.

1.1.2 A-stream: A program-based predictor

Alternatively, *the A-stream is a program-based predictor for the R-stream* [7][13][43][61]. For example, the A-stream assists the performance of the R-stream by improving its branch prediction accuracy. Dynamic branch predictions are classified into two groups, *confident* and

unconfident [19], as shown in Figure 1-2. Confident branch predictions are more likely to be correct and the corresponding branches and computation feeding the branches are removed from the A-stream. Confident predictions represent the most accurate predictions; therefore, removing the computation needed to verify them is sound, and it allows the A-stream to focus instead on verifying unconfident branch predictions. As a result, *many branch mispredictions are resolved by the A-stream in advance of when the R-stream reaches the same point*.

The A-stream also serves as an accurate value predictor [27] for the R-stream. Although only the results of A-stream-executed instructions are available, the predictions are potentially more accurate than those provided by conventional value predictors: A-stream "predictions" are produced by program computation as opposed to being history-based. Perhaps there is some overlap in what the A-stream provides and what a conventional value predictor could provide. Results in Section 4.2.3 indicate that some benchmarks (e.g., *gcc*) benefit primarily from the short program resolving branch mispredictions in advance; others benefit largely due to value predictions from the A-stream, and the effect is not always reproducible by conventional value prediction tables.

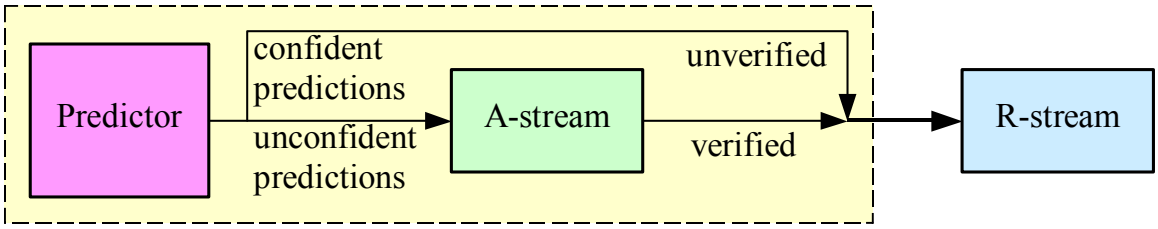

Figure 1-2. A combined predictor/program for improving R-stream branch prediction accuracy.

1.1.3 Slipstream limits and the importance of bypassing instruction fetch

Prior research has shown that, in the absence of any resource constraints, performance is typically limited by mispredicted branches because they serialize execution [25][58]. That is, in an ideal processor with unconstrained fetch and execution bandwidth, mispredicted branches and

their dependence chains tend to dominate the critical path of the program. The A-stream cannot reduce this critical path because the dependence chains of mispredicted branches are not safely removable from the A-stream — only correctly predicted branches are safely removable. The A-stream, like a full version of the program, encounters the same mispredictions and resolves them in program order. Therefore, slipstreaming is not likely to provide performance advantages if fetch and execution bandwidth are unconstrained.

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

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

1.1.4 Other ways of reducing the A-stream

One method for reducing the A-stream is removing branch-predictable computation. Another possibility is removing value-predictable computation. As illustrated in Figure 1-2 in the context of branch prediction, an overall better value predictor may be possible by combining a conventional value predictor with the A-stream. The value predictor identifies and removes

highly value-predictable computation, and the A-stream focuses instead on hard-to-predict values. The R-stream observes a stream of accurate values comprised of both unverified confident values and computed values.

However, this approach complicates the mechanism for reducing the A-stream. For the A-stream to make correct forward progress, the effects of removed, value-predictable computation must be emulated by updating the state of the A-stream with values directly, similar to block/trace/computation reuse [10][15][17] but without the reuse test. This is why we focused initially on the special cases of ineffectual and branch-predictable computation. This computation can be literally removed (i.e., replaced with nothing), and only the program counter needs to be updated to skip instructions.

1.2 Fault tolerance potential

A formal analysis of the fault tolerance of slipstream processors is left for future work. For now, we informally analyze three key scenarios, shown in Figure 1-3, to better understand *potential* fault tolerance. In Figure 1-3, the horizontal lines represent the dynamic instruction streams of the A-stream and R-stream, with older instructions on the left. For this simple analysis, we assume only a single fault occurs and that the fault is ultimately manifested as an erroneous value. A single fault can affect instructions in both streams simultaneously. This is not a problem because the two redundantly-executed copies of an instruction execute at different times (*time redundancy*) [40]; therefore, a single fault that affects both streams will affect different instructions. Since only one copy of an instruction is affected by a fault, we arbitrarily choose the R-stream copy, indicated with X's in Figure 1-3. An X indicates the first erroneous instruction in program order.

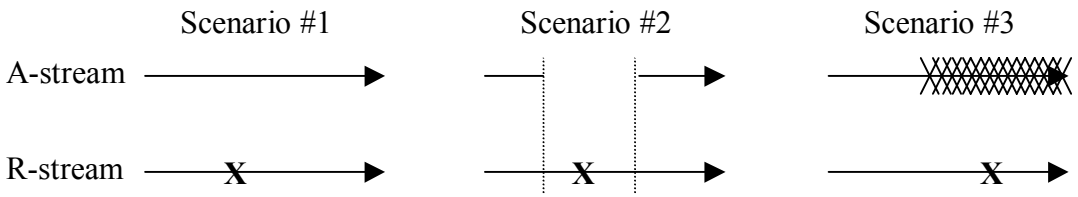

Figure 1-3. Transient fault scenarios.

Scenario #1 in Figure 1-3 shows the A-stream and R-stream executing redundantly, i.e., all instructions overlap and have the same data flow. The fault is detectable because the operands of the first erroneous instruction differ between A-stream and R-stream. Without more information, however, *the fault is indistinguishable from an IR-misprediction*. Under the circumstances, the processor must assume an IR-misprediction since misspeculation is by far the more common case. We point out three successively stronger fault tolerance claims.

1. If we assume a fault cannot flip bits in the R-stream's *architectural* state, then it does not matter that faults and IR-mispredictions are indistinguishable. Recovery succeeds using the R-stream state. Under this model, faults in the pipeline are transparently recoverable. Faults that hit the R-stream register file and data cache are unrecoverable, and worse, undetectable as a fault.
2. If all IR-predictions prior to the first erroneous instruction have been verified, then the source of error is known to be a fault. Software is invoked to diagnose the system and perform recovery operations (e.g., restart). But we default back to (1) if there are prior unresolved IR-predictions.
3. ECC can be used to protect the R-stream register file and data cache, in which case all transient faults within scenario #1 are transparently recoverable.

Scenario #2 in Figure 1-3 shows a region of the program that is not executed redundantly (the A-stream bypassed these instructions). A transient fault in the R-stream is undetectable because there is nothing to compare the erroneous values with. Although an error may be

detected in later, redundantly-executed instructions, the R-stream architectural state is already corrupted and the system is unaware of this fact.

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

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

1.3 Extending capabilities of CMP/SMT processors

Integrating multiple program contexts on a single chip is an important trend. It is difficult to conceive of more effective uses for a billion transistors. The slipstream paradigm extends the capabilities of CMP and SMT substrates. The operating system may flexibly choose among multiple operating modes based on system and user requirements: high job throughput and parallel-program performance (conventional SMT/CMP), improved single-program performance and reliability (slipstream), or highly reliable operation with low performance overhead (AR-SMT[40] / SRT[38]).

1.4 Contributions

This thesis makes six major contributions, outlined below.

1. *First slipstream microarchitecture.* A key contribution of this thesis is the development of the slipstream paradigm and the first slipstream processor. The thesis describes basic components necessary to facilitate slipstream execution.
2. *Understanding slipstream performance and its fundamental limits.* Insight is provided regarding the sources of slipstream performance, and its limitations. This focuses exploration of the architecture and leads to the following key results.

- A 12% average performance improvement is achieved by harnessing an otherwise unused, additional processor in a CMP. Programs with little instruction removal are not sped up at all, whereas programs with upwards of 1/3 instruction removal are sped up by as much as 30%.
- As more execution bandwidth is made available, slipstream execution provides less performance improvement. However, if the A-stream is able to bypass instruction fetching, slipstream retains its edge — because raw instruction fetch bandwidth is not as easily extended as raw execution bandwidth.
- Slipstream execution using two small superscalar cores often achieves similar IPC as one large superscalar core, but slipstream has a potentially faster clock and a more flexible architecture.
- For programs with sufficiently reduced A-streams, slipstream execution on an 8-issue SMT processor improves performance from 10%-20%.
- For some programs, performance improvement is due to the A-stream resolving branch mispredictions in advance. Others benefit largely from A-stream value predictions, and the effect is not always reproducible using conventional value prediction tables.

3. *Method for instruction removal*. A method for instruction removal is proposed, including detecting past-removable instructions in the R-stream and anticipating removable instructions in the A-stream. We also propose a way to do removal prediction at the instruction granularity yet remove dependence chains together. A method for removing instructions before they are fetched is proposed. The thesis focuses on method and not detailed implementation of a slipstream fetch unit.

4. *Efficient CMP memory hierarchy*

    - *Duplication*. Initially, slipstream execution used a complete copy of the program for the A-stream. This copy was forked by the operating system (O/S) and took up the same amount of memory as the R-stream. Using a method called *hardware-based memory duplication*, CMP memory usage is reduced to that of a single program, slipstream execution is transparent to the O/S, and no explicit hardware management is needed (management is implicit via usual block allocations and replacements).
    - *Recovery*. This thesis contributes several much simpler recovery methods enabled by hardware-based memory duplication. These methods include invalidating the L1 cache, partially invalidating the L1 cache, and taking advantage of invalidated cache lines to reduce the penalty of compulsory misses after recovering.

5. *Efficient SMT memory hierarchy*. A highly efficient memory duplication scheme for a single-L1-cache SMT processor is developed. Not only does this scheme significantly reduce L1 cache pressure due to duplication, it also provides a simple zero-penalty method for repairing A-stream memory state and implicitly prefetches for the R-stream.
6. *Management of slipstream execution mode*. Slipstream execution does not accelerate all programs. Even programs that benefit from slipstream may not benefit during all phases. We have developed very simple hardware support for (1) predicting the effectiveness of slipstream execution across and within applications and (2) dynamically enabling and disabling slipstream execution mode. The core mechanisms already exist: The IR-predictor/IR-detector components conveniently provide continuous feedback (even with the A-stream disabled), and enabling/disabling the A-stream is procedurally identical to recovering from an IR-misprediction. The proposed *slipstream management unit* empowers

an operating system to choose between slipstream execution mode and conventional modes of execution. Alternatively, if the operating system selects slipstream execution mode for an application, the slipstream management unit can forfeit a spare processor during intervals for which slipstream is less effective and reclaim it at some later time.

1.5 Thesis organization

Related work is covered in Chapter 2. The slipstream processor (first contribution) and instruction removal (third contribution) are covered in Chapter 3. This introductory chapter and Chapter 4 provide insights regarding slipstream performance (second contribution). Efficient CMP and SMT memory hierarchies (fourth and fifth contributions) are covered in Chapter 5 and Chapter 6, respectively. Management of slipstream execution mode (sixth contribution) is covered in Chapter 7. Chapter 8 summarizes the thesis and proposes future work.

Chapter 2 Related Work

2.1 Basis for reducing the A-stream

Researchers have demonstrated a significant amount of redundancy, repetition, and predictability in general-purpose programs [10][15][17][26][27][29][31][46][47][57]. This prior research forms a basis for creating the shorter program in slipstream processors. A technical report studying the feasibility of reducing programs [41] showed (1) it is possible to ideally construct significantly reduced programs that produce correct final output, and (2) AR-SMT is a convenient execution model to exploit this property (AR-SMT is a fault-tolerant microarchitecture and precursor to slipstream execution, described in Section 2.2).

2.2 Execution models based on hardware multithreading

Advanced-stream/Redundant-stream Simultaneous Multithreading (AR-SMT) [40] is based on the realization that microarchitecture performance trends and fault tolerance are related. Time redundancy — running a program twice to detect transient faults — is cheaper than hardware redundancy, but it doubles execution time. AR-SMT runs the two programs simultaneously [56] but delayed slightly (via the delay buffer), reducing the performance overhead of time redundancy. Results are compared by communicating all retired A-stream results to the R-stream, and the R-stream performs the checks. Here, the R-stream leverages speculation concepts [27] — the A-stream results can be used as ideal predictions. The R-stream

fetches/executes with maximum efficiency, further reducing the performance overhead of time redundancy. And the method for comparing the A-stream and the R-stream is conveniently in place, in the form of misprediction-detection hardware. In summary, AR-SMT leverages the underlying microarchitecture to achieve broad coverage of transient faults with low overhead, both in terms of performance and changes to the existing design.

DIVA [3] and SRT [38] are two other examples of fault-tolerant architectures designed for commodity high-performance microprocessors. DIVA detects a variety of faults, including design faults, by using a verified checker to validate computation of the complex processor core. DIVA leverages an AR-SMT technique — the simple checker is able to keep pace with the core by using the values it is checking as predictions. SRT improves on AR-SMT in a variety of ways, including a formal and systematic treatment of SMT applied to fault tolerance (e.g., spheres of replication).

Tullsen et al. [55][56] and Yamamoto and Nemirovsky [59] proposed simultaneous multithreading for flexibly exploiting thread-level and instruction-level parallelism. Olukotun et. al. [34] motivate using chip multiprocessors.

Farcy et al. [13] proposed resolving branch mispredictions early by extracting the computation leading to branches. Zilles and Sohi [61] similarly studied the computation chains leading to mispredicted branches and loads that miss in the level-two cache. They suggest identifying a difficult subset of the program for pre-execution [43][44], potentially prefetching branch predictions and cache lines that would otherwise be mispredictions and cache misses. Pre-execution typically involves pruning a small kernel from a larger program region and running it as a prefetch engine [44]. Roth and Sohi [43] developed a new paradigm called Speculative Data-Driven Multithreading that implements pre-execution generally. A large body

of work on pre-execution architectures has developed in recent years [2][4][7][9][28][33][49][62]. Rather than spawn many specialized kernels on-the-fly, our approach uses a single, functionally complete, and persistent program (A-stream). Slipstream execution avoids the conceptual and possibly real complexity of forking private contexts, within which the specialized kernels must run.

Dundas and Mudge [11] proposed *run-ahead* to improve first-level data cache performance. Run-ahead enables an in-order pipeline to silently fetch and execute instructions around a stalled load, exploiting the otherwise idle execution core to generate highly accurate data prefetches. The only hardware required is an additional register file. Balasubramonian, Dwarkadas, and Albonesi [4] extend run-ahead to out-of-order pipelines, to reserve a portion of the window for a future thread. The future thread executes when the primary thread is limited by resource availability. The future thread is not bound by in-order retirement, so it can examine a much larger window. Results are communicated to the primary thread via the instruction and data caches (pre-fetching), instruction reuse buffer (pre-computation), and branch predictor (resolving branch mispredictions early). The speculative thread is only initiated when the main thread is stalled, whereas slipstream deploys a redundant program (A-stream).

Speculative multithreading architectures [1][34][48] accelerate a single program by dividing it into speculatively-parallel threads. The speculation model uses one architectural context and future threads are spawned within temporary, private contexts, each inherited from the preceding thread's context. Future thread contexts are merged into the architectural context as threads complete. Our speculation model uses redundant architectural contexts, so no forking or merging is needed. Moreover, there are no dependences between the architecturally-independent threads, rather, outcomes are communicated as predictions via a simple FIFO queue. Register and

memory mechanisms of the underlying processor are relatively unchanged by slipstreaming (particularly if there is an existing interface for consuming value predictions at the rename stage). In contrast, speculative multithreading often requires elaborate inter-thread register/memory dependence mechanisms.

SSMT [7] runs microthreads simultaneously with an application to optimize its performance. Microthreads are small routines designed in conjunction with applications and the processor. For example, microthreads may perform cache prefetching, improve branch prediction accuracy [7], or optimize exception handling [60].

The DataScalar paradigm [6] runs redundant programs on multiple processor-and-memory cores to eliminate memory read requests.

2.3 Thread memory management

In this section, memory management strategies of various multithreaded execution models are described and contrasted with slipstream memory management. References are made to methods proposed in Chapter 5, which the reader may read first to better appreciate the following coverage of related work.

2.3.1 Redundant execution

The AR-SMT [40], DIVA [3], and SRT [38] architectures execute two redundant copies of the program for fault tolerance. AR-SMT uses software-based memory duplication to enable arbitrary slip between the A-stream and R-stream, limited only by the length of the delay buffer. SRT merges redundant stores before they are committed, so memory is not duplicated. However, the SRT design space allows for register duplication, memory duplication, or both. The initial DIVA implementation does not duplicate register or memory state. However, to reduce stalls due to limited slip between the leading and checker threads, enhanced DIVA

implementations include duplicate register files and/or L1 caches (the L1 caches can be different sizes) [8]. State of the leading thread is not written back to the L2, although the issue of replacing modified lines in the L1 cache of the leading thread is not discussed. Preventing stale references may require that the checker thread signal when it has performed all corresponding modifications, at which time the leading thread can safely discard its version of the line. On the other hand, permitting stale references simplifies the hardware, but whole-cache invalidations would become more frequent and the recovery optimizations developed in this thesis can benefit the DIVA microarchitecture as well. To the best of our knowledge, this thesis is the first to propose discarding updates of the leading thread based on the prediction that discarded updates are likely to be re-created in a timely fashion by the trailing thread [37]. This approach eliminates explicit management, the prediction is accurate in practice, and correctness is maintained. Other unique aspects include the invalidate-dirty-line recovery heuristic and the use of invalidated lines as highly-accurate value predictions for reducing the impact of recovery-induced misses, both of which are crucial for maximizing performance without the recovery controller. We also provide insight by characterizing the frequency of references to *stale* data, *self-repair* data, *persistent-stale* data, and *persistent-skipped-write* data. Finally, we implemented hardware-based memory duplication in the context of a POWER4-style memory hierarchy including considerations for cache coherence between the L1 caches and with the rest of the system.

2.3.2 Speculative multithreading

Speculative multithreading architectures (e.g., [1][24][34][35][48][50]) accelerate a sequential program by dividing it into speculative parallel tasks, and concurrently running the tasks on distributed processing elements or a simultaneous multithreading pipeline. Some of

these architectures provide private L1 caches for tasks. Examples include the Multiscalar Processor with SVC [16], TLDS [50], and MDT [24]. The L1 caches are explicitly managed to enforce inter-task dependences. In contrast, there are no dependences between the A-stream and R-stream. Confining the A-stream to its L1 cache occasionally causes A-stream updates to be lost. Whether the A-stream re-references stale data or updated data depends on whether or not the R-stream has performed its corresponding store, but this dependence is not enforced. Therefore, the L1 caches are not explicitly managed in the case of slipstream.

2.3.3 Pre-execution

Speculative Data-Driven Multithreading [43] and other pre-execution architectures [2][4][7][9][11][13][28][33][49][62] fork specialized threads to prefetch cache misses and resolve branch mispredictions in advance. A key difference is the use of multiple, short-lived, specialized threads versus a single, persistent, functionally-complete program (A-stream). This difference results in very different microarchitectures and, specifically, memory renaming has evolved differently. Use of the memory hierarchy (e.g., L1 cache or full duplication) is tailored towards the A-stream's persistence and completeness. Linking stores directly to loads via an explicitly-managed memory cloaking table [32], bypassing the memory system entirely, is tailored towards short-lived dependence-chain-based threads. Many pre-execution architectures omit stores from the specialized threads altogether because stores do not typically affect their specific target.

Chapter 3 Slipstream Microarchitecture

A slipstream processor requires two architectural contexts, one for each of the A-stream and R-stream, and new hardware for directing instruction-removal in the A-stream and communicating state between the threads. A high-level block diagram of a slipstream processor implemented on top of a dual-processor chip multiprocessor (CMP) is shown in Figure 3-1, although an SMT processor might also be used. The shaded boxes show the original processors comprising the multiprocessor. Each is a conventional superscalar/VLIW processor with a branch predictor, instruction and data caches, and an execution engine — including the register file and either an in-order pipeline or out-of-order pipeline with reorder buffer.

Slipstreaming requires four new components.

1. The *instruction-removal predictor*, or IR-predictor, is a modified branch predictor. It generates the program counter (PC) of the next block of instructions to be fetched in the A-stream. Unlike a conventional branch predictor, however, *the predicted next PC may reflect skipping past any number of dynamic instructions* that a conventional processor would otherwise fetch and execute. Also, the IR-predictor indicates which instructions *within* a fetched block can be removed after the instruction fetch stage and before the decode/dispatch stage.
2. The *instruction-removal detector*, or IR-detector, monitors the R-stream and detects instructions that could have been removed from the program, and might possibly be removed

in the future. The IR-detector conveys to the IR-predictor that particular instructions should potentially be skipped by the A-stream when they are next encountered. Repeated indications by the IR-detector build up confidence in the IR-predictor, and the predictor will remove future instances from the A-stream.

3. The *delay buffer* is used to communicate control-flow and data-flow outcomes from the A-stream to the R-stream [40].
4. The *recovery controller* maintains the addresses of memory locations that are potentially corrupted in the A-stream context. A-stream context is corrupted when the IR-predictor removes instructions that should not have been removed. Unique addresses are added to and removed from the recovery controller as stores are processed by the A-stream, the R-stream, and the IR-detector. The current list of memory locations in the recovery controller is sufficient to recover the A-stream memory context from the R-stream's memory context. The register file is repaired by copying all values from the R-stream's register file.

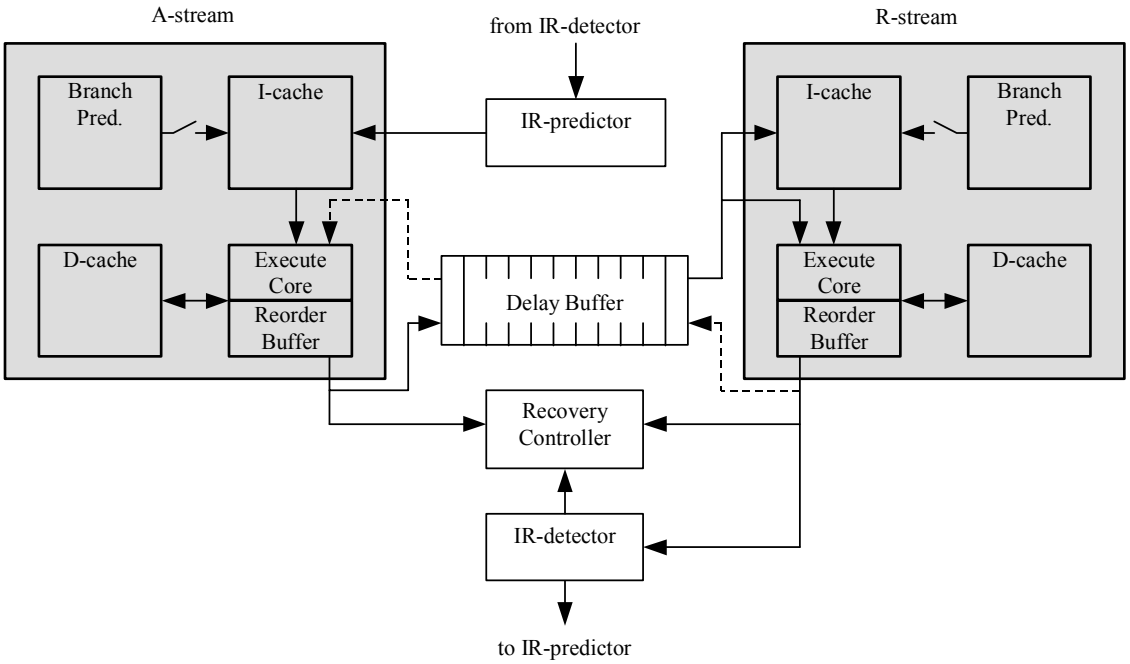

Figure 3-1. Slipstream processor using a dual-processor CMP substrate.

The diagram in Figure 3-1 shows the A-stream on the leftmost core and the R-stream on the rightmost core. This is arbitrary and does not reflect specializing the two cores. A real design would have one core that flexibly supports either the A-stream or R-stream. In any case, there is a clear symmetry that makes designing a single core natural. In both cores, there is an interface to the fetch unit that overrides the conventional branch predictor, indicated symbolically with an open switch and a second interface to the fetch unit. Likewise, both cores show symmetric interfaces to and from the execution pipeline.

3.1 Creating the shorter program

3.1.1 Base IR-predictor

The IR-predictor resembles a conventional branch predictor. In this thesis, the IR-predictor is indexed identically to a *gshare* predictor [30], i.e., an index is formed by XORing the PC and the global branch history bits. Each table entry contains information for a single dynamic basic block.

- *Tag*: This is the start PC of the basic block and is used to determine whether or not the entry contains information for the desired block.
- *2-bit counter*: If the block ends in a conditional branch, the 2-bit counter predicts its direction.
- *Confidence counters*. There is a resetting confidence counter [19] for each instruction in the block. The counters are updated by the IR-detector: a counter is incremented if the corresponding instruction is detected as removable, otherwise the counter is reset to zero. If a counter is saturated, then the corresponding instruction will be removed from the A-stream when it is next encountered.

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

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

In Figure 3-1, the IR-predictor is shown as a new component outside the processor core that overrides the conventional branch predictor. Alternatively, since the IR-predictor is built on top of a conventional branch predictor, the core's predictor and the IR-predictor may be integrated.

3.1.2 Improved IR-predictor: Bypassing instruction fetch

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

The A-stream is more effective if fewer fetch cycles are expended on it than on the full program. In Figure 3-2, we show an example of how the number of fetch cycles can potentially be reduced. Four basic blocks, labeled A through D, are to be predicted and fetched. The corresponding table entries in the IR-predictor are shown; darker shaded entries indicate that all of the confidence counters are saturated and the entire basic block is predicted for removal. The base IR-predictor predicts each block in sequence, requiring four cycles. During two of these

cycles, the instruction cache fetches instructions and then throws them all away (basic blocks B and C). Clearly, only two fetch cycles are required, but it is not known in advance that instruction fetching of blocks B and C can be bypassed.

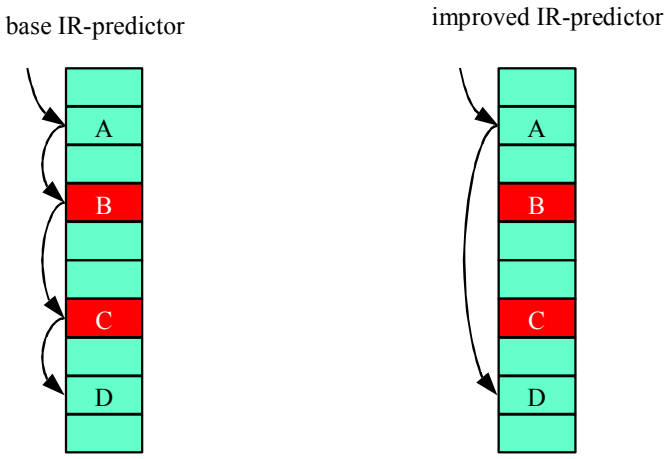

Figure 3-2. Reducing fetch cycles in the A-stream.

Interestingly, the effect we want to produce — bypassing basic blocks — is the same effect produced by taken branches. The improved IR-predictor shown on the right-hand side of Figure 3-2 exploits the analogy. The improved predictor "converts" the branch terminating block A into a taken branch whose target is block D. Below, we consider two possible ways to implement this conversion.

- Two additional pieces of information are stored in block *A*'s table entry. First, the predicted directions of any bypassed branches must be stored, in this case, the predicted directions of the branches in blocks *B* and *C*. The reason is that all control flow information must be pushed onto the delay buffer to be consumed by the R-stream, in spite of partially bypassing instruction fetching in the A-stream. Second, a target address must be stored, in this case, the start PC of block *D*. The target address overrides the next PC computation performed by the fetch unit. The additional information (bypassed predictions and corresponding target

address) is accumulated for block *A*'s entry as the IR-detector sequentially updates the entries of blocks *B*, *C*, and *D*.

- Effectively, the branch terminating block *A* is now a multi-way branch. It has more potential targets than its original taken and fall-through targets because it inherits the targets of skipped blocks. The processor's branch target buffer may be modified to store multiple targets per branch. Now, dynamically-created target addresses do not have to be stored in the IR-predictor. The bypassed predictions still need to be stored and, conveniently, this path information is sufficient to select the appropriate target address from the branch target buffer.

3.1.3 IR-detector

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

As new instructions are merged into the R-DFG, the IR-detector watches for any of three triggering conditions for instruction removal. Triggering conditions are unreferenced writes (a write followed by a write to the same location, with no intervening read), non-modifying writes [26][29][31][57] (writing the same value to a location as already exists at that location), and correctly-predicted branch instructions. When a triggering condition is observed, the corresponding instruction is selected for removal. Then, the circuits forming the R-DFG back-

propagate the selection status to predecessor instructions. Predecessors may also be selected if certain criteria (described later) are met.

The IR-detector is shown in Figure 3-3. A single R-DFG is shown, however, the buffering could be partitioned into multiple smaller R-DFGs. The latter approach reduces the size/complexity of each individual R-DFG but still allows a large analysis scope for killing values (observing another write to the same location).

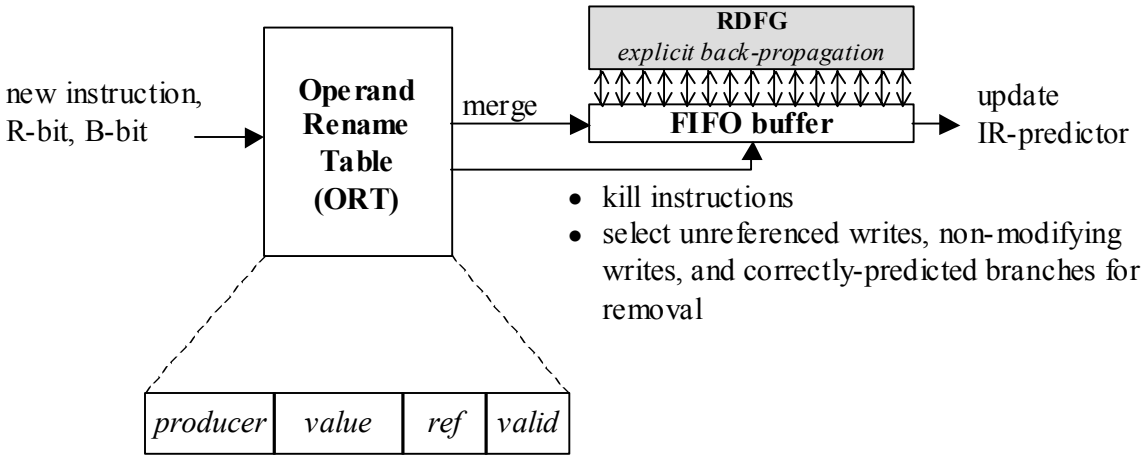

Figure 3-3. IR-detector.

The operand rename table (ORT) in Figure 3-3 is similar to a conventional register rename table but it can track both memory addresses and registers. A single entry of the operand rename table is shown in Figure 3-3. To merge an instruction into the R-DFG, each source operand is checked in the ORT to get the most recent producer of the value (check the valid bit and producer field). The instruction uses this information to establish connections with its producer instructions, i.e., set up the back-propagation logic (if the buffering is partitioned into smaller R-DFGs, connections cannot be made across partition boundaries). The *ref bit* is set for each source operand indicating the values have been used. If the instruction writes a register/memory

location, then the corresponding rename table entry is checked to detect non-modifying/unreferenced writes and to kill values, as follows.

1. If the *valid bit* is set and the current instruction produced the same value as indicated in the *value* field, then the current instruction is a non-modifying write. The current instruction is selected for removal as it is merged into the R-DFG. No fields are updated in the rename table entry since the old producer remains "live" in this case.
2. If the *valid bit* is set and the new and old values do not match, then the old producer indicated by the *producer* field is killed. Furthermore, if the *ref bit* is not set, then the old producer is an unreferenced write and is selected for removal. Finally, all fields in the rename table entry are updated to reflect the new producer.

Correctly predicted branch instructions (indicated by the B-bit in Figure 3-3) are selected for removal when they are merged into the R-DFG.

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

1. All of *x*'s dependent instructions must be known, i.e., *x*'s production(s) must be killed by other production(s).
2. All of *x*'s dependent instructions must be selected for removal.
3. *All of x's dependent instructions must have been removed by the IR-predictor this time around*. (The R-bit in Figure 3-3 indicates whether or not an instruction was removed by the IR-predictor this time around.)

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

The third back-propagation condition, highlighted above and called the *R-bit criterion*, is a major innovation. The R-bit criterion ensures a producer's counter saturates only after all consumers' counters saturate. This prevents recurring IR-mispredictions caused by (1) multiple consumers on different control-flow paths and (2) IR-predictor aliasing that causes a consumer's counter to reset.

The first case is shown in Figure 3-4. Nodes are instructions, solid arrows are control-flow edges, and dashed arrows are data dependences. A single producer instruction has consumers on both paths after the branch instruction. Thus, the producer is back-propagated to along two separate control-flow paths. If the intervening branch alternates fairly often, then the back-propagated instruction's confidence can saturate (shown as a wholly-shaded node) before either of the consumer instructions is removable (shown as the three-quarter-shaded nodes). This results in multiple IR-mispredictions until the confidence counters of all consumers become saturated, or until one of the consumer instructions becomes effectual (resetting the confidence counter of the back-propagated instruction). By requiring an instruction to be actually removed before back-propagation (R-bit criterion), the back-propagated instruction doesn't even begin to get confident until its consumers are removed.

A similar situation occurs when a consumer's basic block is displaced from the IR-predictor by aliasing, resetting its removal information. If the producer's block is not aliased, it is still removable, resulting in the same scenario as before (producer's counter saturated and consumer's counter not saturated). Back-propagation based on the R-bit criterion will likely generate a single IR-misprediction from the aliasing but will correctly reset the back-propagated instruction's counter after the IR-misprediction.

The significance of the R-bit method is that it manages confidence counters individually, yet implicitly synchronizes the counters of arbitrarily distant producers and consumers.

Recently, Koppanalil and Rotenberg [22][23] reduced the complexity of the IR-detector substantially with the realization that back-propagated instructions are no different from unreferenced writes, therefore back-propagation can be achieved implicitly via the ORT. This eliminates the R-DFG component in Figure 3-3. The method described above (the R-bit criterion in particular) is a key enabler for implicit back-propagation.

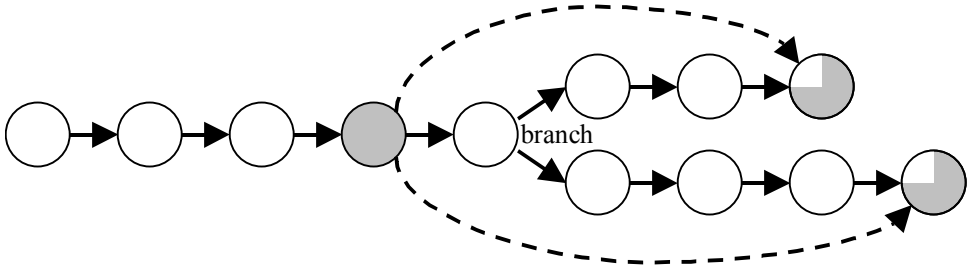

Figure 3-4. Example showing the need for synchronizing counters.

3.2 Delay buffer

The delay buffer is a simple FIFO queue that allows the A-stream to communicate control-flow and data-flow outcomes to the R-stream. The A-stream pushes both a complete history of branch outcomes and a partial history of operand values onto the delay buffer. This is shown in Figure 3-1 with a solid arrow from the reorder buffer of the A-stream (left-most processor) to the delay buffer. Value history is partial because only a subset of the program is executed by the A-stream. Complete control history is available, however, because the IR-predictor predicts all branches even though the A-stream may not fetch all instructions (Section 3.1.2).

The R-stream pops control-flow and data-flow information from the delay buffer. This is shown in Figure 3-1 with solid arrows from delay buffer to the instruction cache and execution core of the R-stream (right-most processor). Branch outcomes from the delay buffer are routed

to the instruction cache to direct instruction fetching. Source operand values and load/store addresses from the delay buffer are merged with their respective instructions after the instructions have been fetched/renamed and before they enter the execution engine. To know which values/addresses go with which instructions, the delay buffer also includes a single bit per dynamic instruction indicating which instructions were skipped by the A-stream (for which there is no data-flow information available).

3.3 IR-misprediction recovery

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

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

Some IR-mispredictions take awhile to cause any visible symptoms in the A-stream. For example, a store may be removed incorrectly and the next load to the same location may not occur for a very long time. The IR-detector can detect these IR-mispredictions much sooner by comparing its computed removal information against the corresponding predicted removal information — if they differ, computation was removed that should not have been. Thus, the IR-detector may serve dual roles of updating the IR-predictor and checking for IR-mispredictions.

However, we found that checking for IR-mispredictions in the IR-detector provides little benefit and that waiting for IR-mispredictions to be detected by the R-stream yields the best performance.

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

All that remains is restoring the corrupted register and memory state of the A-stream so it is consistent with the R-stream. Because register state is finite, the entire register file of the R-stream is copied to the A-stream register file. The movement of data (both register and memory values) occurs via the delay buffer, in the reverse direction, as shown with dashed arrows in Figure 3-1.

The recovery controller receives control signals and the addresses of store instructions from the A-stream, the R-stream, and the IR-detector, as shown in Figure 3-1. The control signals indicate when to start or stop tracking a memory address (only unique addresses need to be tracked). After detecting an IR-misprediction, stores may either have to be "undone" or "done" in the A-stream.

- The recovery controller tracks addresses of stores retired in the A-stream but not yet retired in the R-stream. After detecting an IR-misprediction, these A-stream stores must be "undone" since the R-stream has not yet performed the companion, redundant store.
- The recovery controller tracks addresses of stores retired in the R-stream and skipped in the A-stream, only until the IR-detector verifies that the stores are truly ineffectual. When an IR-

misprediction is detected, all unverified, predicted-ineffectual stores are “done” in the A-stream by copying data from the redundant locations in the R-stream.

Chapter 4 Evaluation

For the experiments in this chapter, the memory system is somewhat idealized to isolate the performance of slipstream execution disregarding the effects of full memory duplication. L1 instruction and data caches are modeled but the L2 cache is infinite. The memory hierarchy is modeled in significant detail in Chapter 5, which investigates various slipstream memory management alternatives.

4.1 Simulation environment

We developed a detailed execution-driven simulator of a slipstream processor. The simulator faithfully models the architecture depicted in Figure 3-1 and outlined in Chapter 3: the A-stream produces real, possibly incorrect values/addresses and branch outcomes, the R-stream checks the A-stream and initiates recovery actions, A-stream state is recovered from the R-stream state, etc. The simulator itself is validated via a functional simulator run independently and in parallel with the detailed timing simulator [48][42]. The functional simulator checks retired R-stream control-flow and data-flow outcomes.

The Simplescalar [5] compiler and ISA are used. We use the SPEC95 integer benchmarks (-O3 optimization) run to completion (Table 4-1).

Table 4-1. Benchmarks.

benchmark	input dataset	dynamic instruction count
compress	40000 e 2231	124 million
gcc	cccp.i -o cccp.s	265 million
go	9 9	133 million
jpeg	vigo.ppm	166 million
li	test.lsp (queens 7)	202 million
m88ksim	-c < ctl.in (dcrand.big)	121 million
perl	scrabble.pl < scrabble.in	108 million
vortex	vortex.in (persons.250)	101 million

Microarchitecture parameters are listed in Table 4-2. The top half of the table lists parameters for individual processors within a CMP or, alternatively, a single SMT processor. The bottom half describes the four slipstream components. A large IR-predictor is used for accurate instruction removal. The removal confidence threshold is 32. The IR-detector has a scope of 256 instructions and the R-DFG is unpartitioned. The delay buffer stores 256 instructions (data flow buffer) and 4K branch predictions (control flow buffer). The recovery controller tracks any number of store addresses, although we observe not too many outstanding addresses in practice. The recovery latency (*after* the IR-misprediction is detected) is 5 cycles to startup the recovery pipeline, followed by 4 register restores per cycle, and lastly 4 memory restores per cycle.

Table 4-2. Microarchitecture configuration.

single processor core	
instruction cache	size/assoc/repl = 64KB/4-way/LRU
	line size = 16 instructions
	2-way interleaved
	miss penalty = 12 cycles
data cache	size/assoc/repl = 64KB/4-way/LRU
	line size = 64 bytes
	miss penalty = 14 cycles
superscalar core	reorder buffer: 64, 128, or 256 entries
	dispatch/issue/retire bandwidth: 4-/8-/16-way
	n fully-symmetric function units (*n* = issue b/w)
	n loads/stores per cycle (*n* = issue b/w)
execution latencies	address generation = 1 cycle
	memory access = 2 cycles (hit)
	integer ALU ops = 1 cycle
	complex ops = MIPS R10000 latencies
new components for slipstreaming	
IR-predictor	2^{20} entries *gshare*-indexed (16 bits of global branch history)
	block size = 16 16 confidence counters per entry
	confidence threshold = 32
IR-detector	R-DFG = 256 instructions, unpartitioned
delay buffer	data flow buffer: 256 instruction entries
	control flow buffer: 4K branch predictions
recovery controller	# of outstanding store addr. = unconstrained
	recovery latency (*after* IR-misp. detected): • 5 cycles to start up recovery pipeline • 4 reg. restores/cycle (64 regs performed 1st) • 4 mem. restores/cycle (mem performed 2nd) • ∴ min. latency (no memory) = 21 cycles

4.2 Results

4.2.1 Slipstream performance results

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

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

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

IPC performance of the eight models is shown in Figure 4-1. The first conclusion is a slipstream processor can exploit a second, otherwise unused processor to significantly improve single-program performance. From Figure 4-2, CMP(2x64x4) performs on average 12% better

than using only a single SS(64x4) processor. And CMP(2x128x8) performs on average 7% better than using only a single SS(128x8) processor. Slipstreaming degrades performance in jpeg, by 1% and 5% for CMP(2x64x4) and CMP(2x128x8), respectively. Jpeg's A-stream is not reduced much and jpeg is already quite parallel; IR-mispredictions cause an overall degradation.

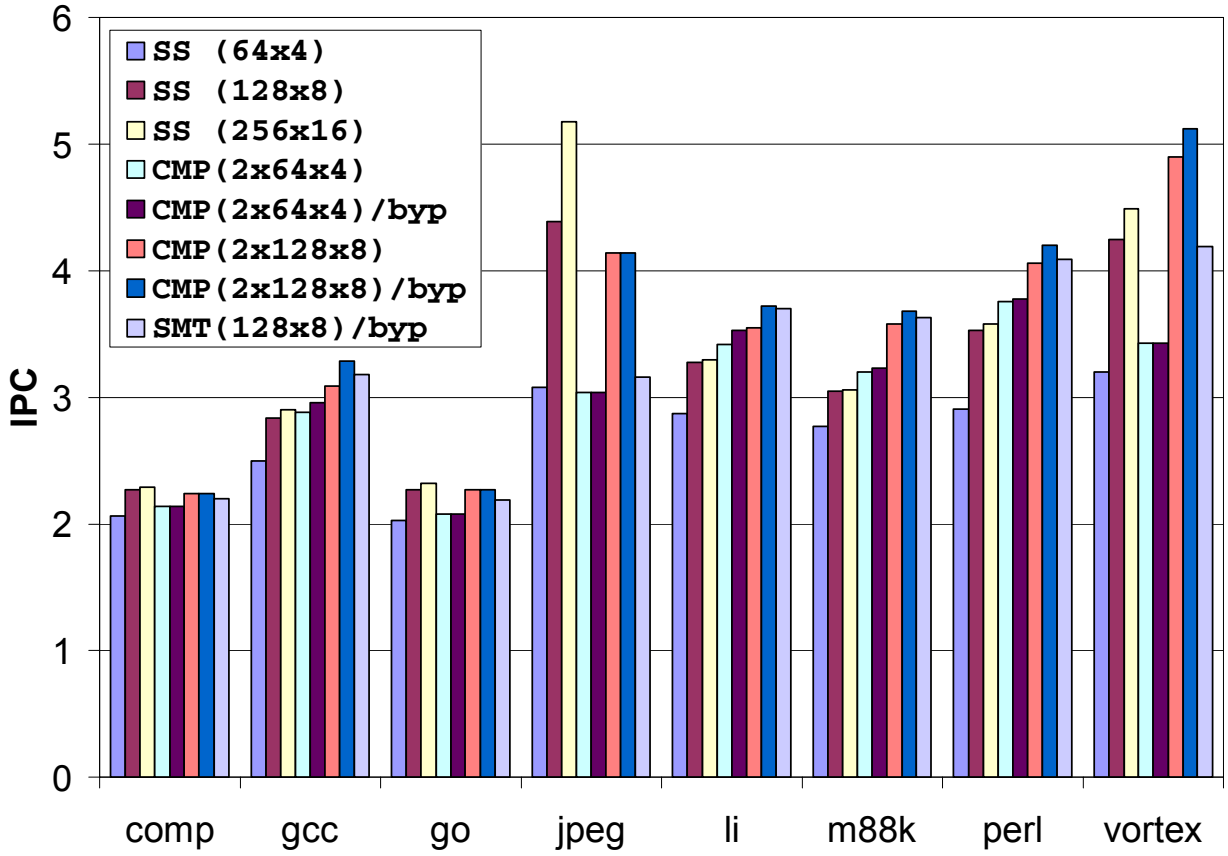

Figure 4-1. IPC results. (Slipstream processors are rightmost five bars.)

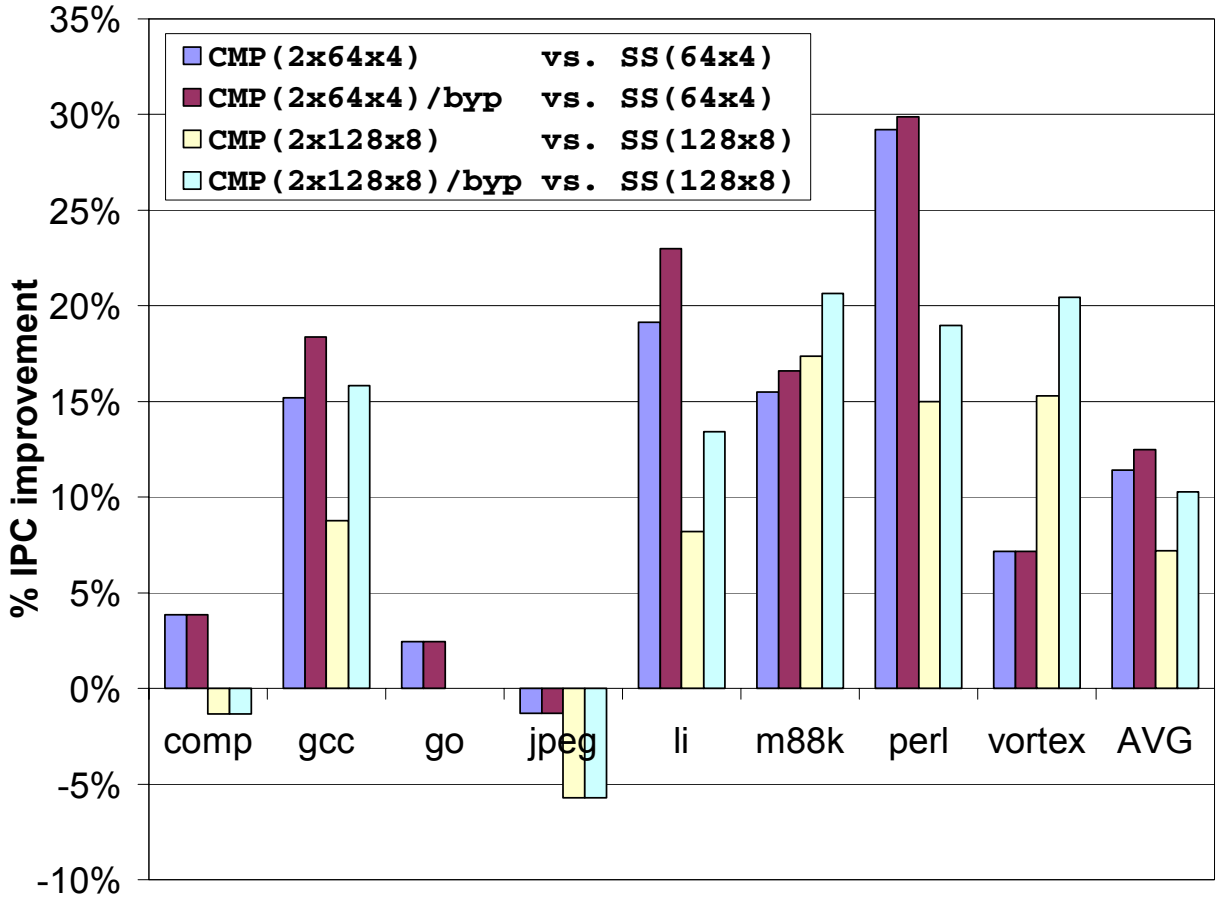

Figure 4-2. Performance improvement using a second processor for slipstreaming.

The second conclusion is the benefit of slipstreaming decreases as more execution bandwidth is made available. This is evident from the first and third bars of Figure 4-2. For all except m88ksim and vortex, the performance improvement of CMP(2x128x8) over SS(128x8) is less than the improvement of CMP(2x64x4) over SS(64x4). For example, perl drops from a 30% improvement down to a 15% improvement as the window size and issue bandwidth of the processor core is doubled. This is evidence for the arguments made in Section 1.1.3.

The above result motivates reducing the number of instructions fetched in the A-stream, using the improved IR-predictor (Section 3.1.2). From Figure 4-2, CMP(2x64x4)/byp on average performs 13% better than SS(64x4), a modest change from CMP(2x64x4). As expected,

it is more important to bypass instruction fetching for larger processor cores. CMP(2x128x8)/byp on average performs 10% better than SS(128x8), whereas CMP(2x128x8) performs 7% better. With the improved IR-predictor, slipstream performance improvement increases from 8% to 16% for gcc, from 8% to 14% for li, from 17% to 21% for m88ksim, from 15% to 19% for perl, and from 15% to 20% for vortex.

In Figure 4-3, we compare the performance of slipstreaming on two small processors to the performance of a larger processor. The larger processor has the same total number of ROB entries and issue bandwidth as the two smaller processors combined. For half of the benchmarks (perl, gcc, li, m88ksim), CMP(2x64x4)/byp actually performs from 4% to 8% better than SS(128x8). Overall, CMP(2x64x4)/byp performs comparably to the more complex, less flexible SS(128x8) processor — within 5% on average. The results are more pronounced for CMP(2x128x8)/byp, which on average performs 7% better than SS(256x16).

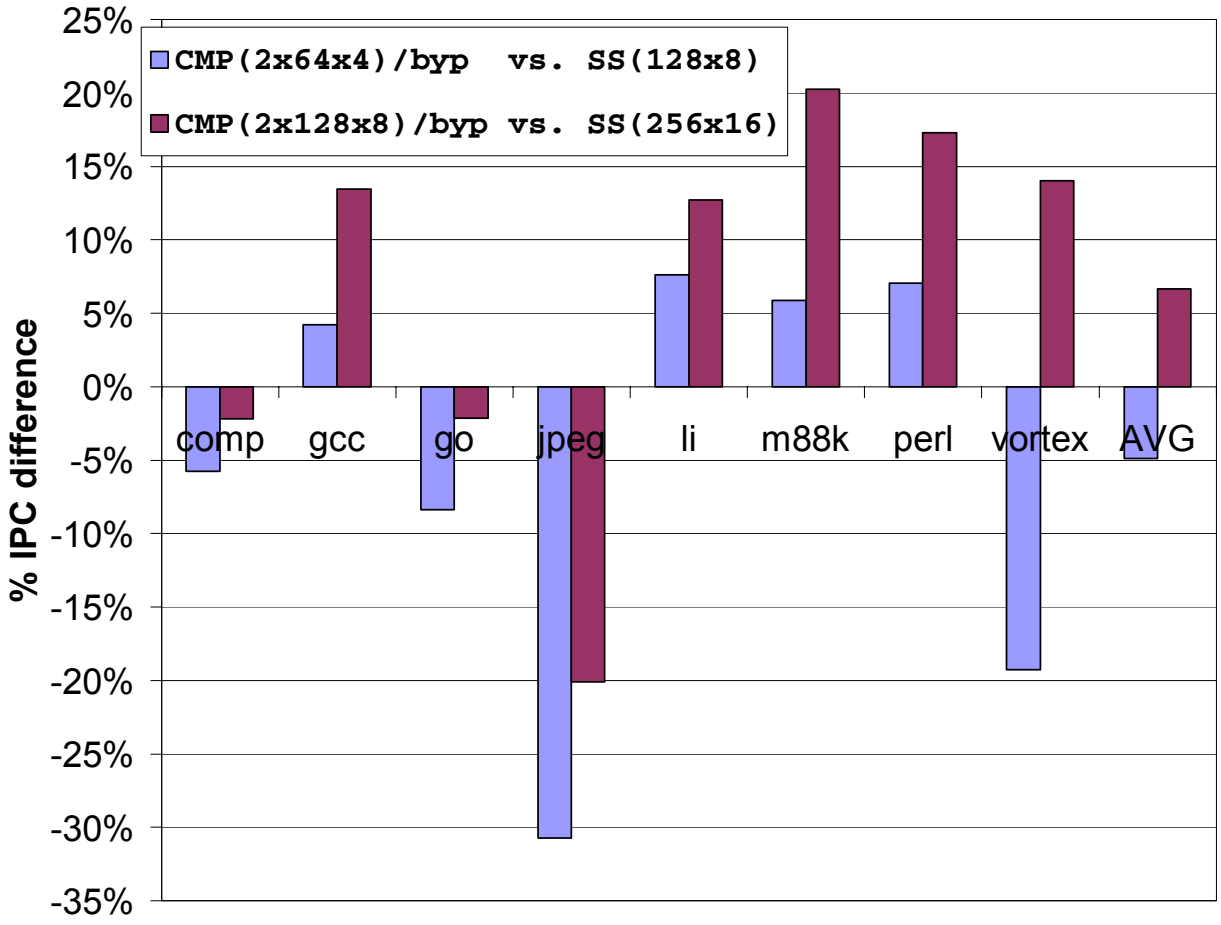

Figure 4-3. Performance of slipstream execution on two small processors vs. performance of conventional non-redundant execution on a single large processor.

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

exceed half of the peak IPC — and the A-stream steals useful processor bandwidth from the R-stream. The effect is more pronounced for jpeg than for vortex because jpeg exhibits little reduction in its A-stream (Figure 4-5).

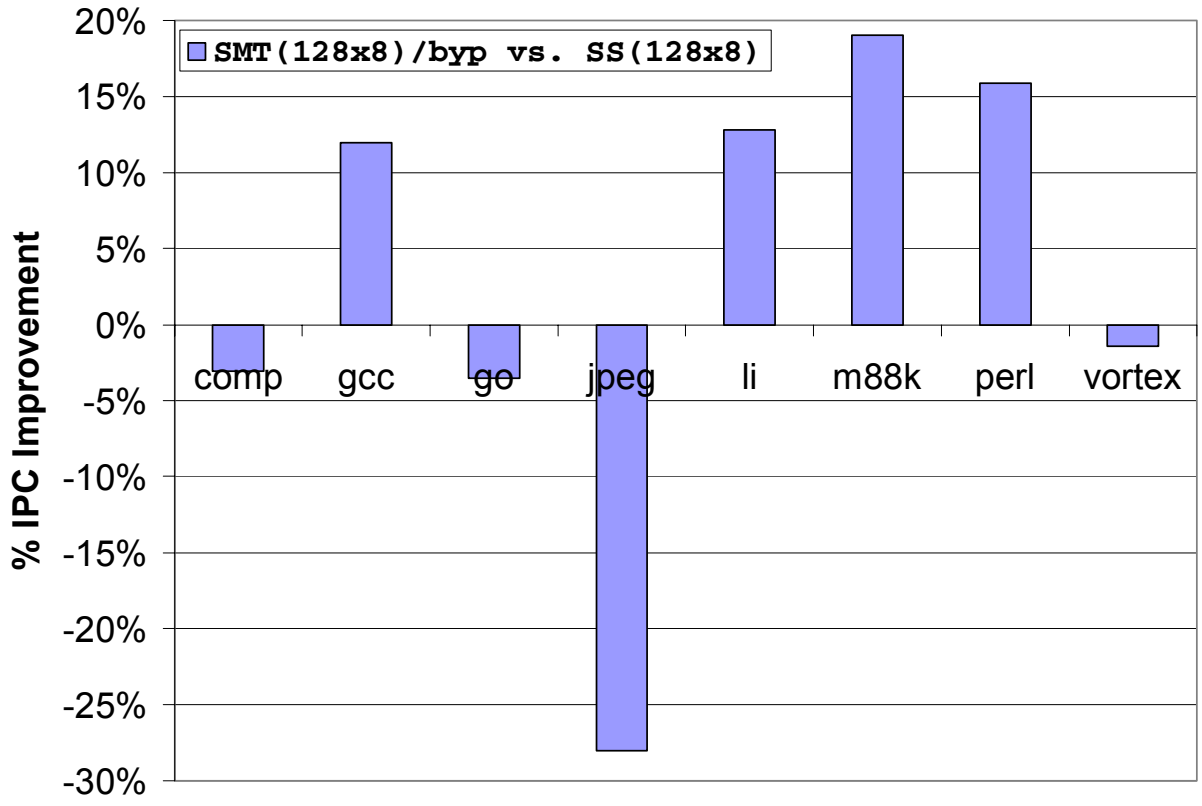

Figure 4-4. Performance improvement of SMT(128x8)/byp over SS(128x8).

4.2.2 Instruction removal

Figure 4-5 shows the fraction of original dynamic instructions removed from the A-stream. Nearly half of the program is removed for gcc, li, perl, and vortex, and about two-thirds of m88ksim is removed. About 20% of compress is removed, and only 10% for go and jpeg.

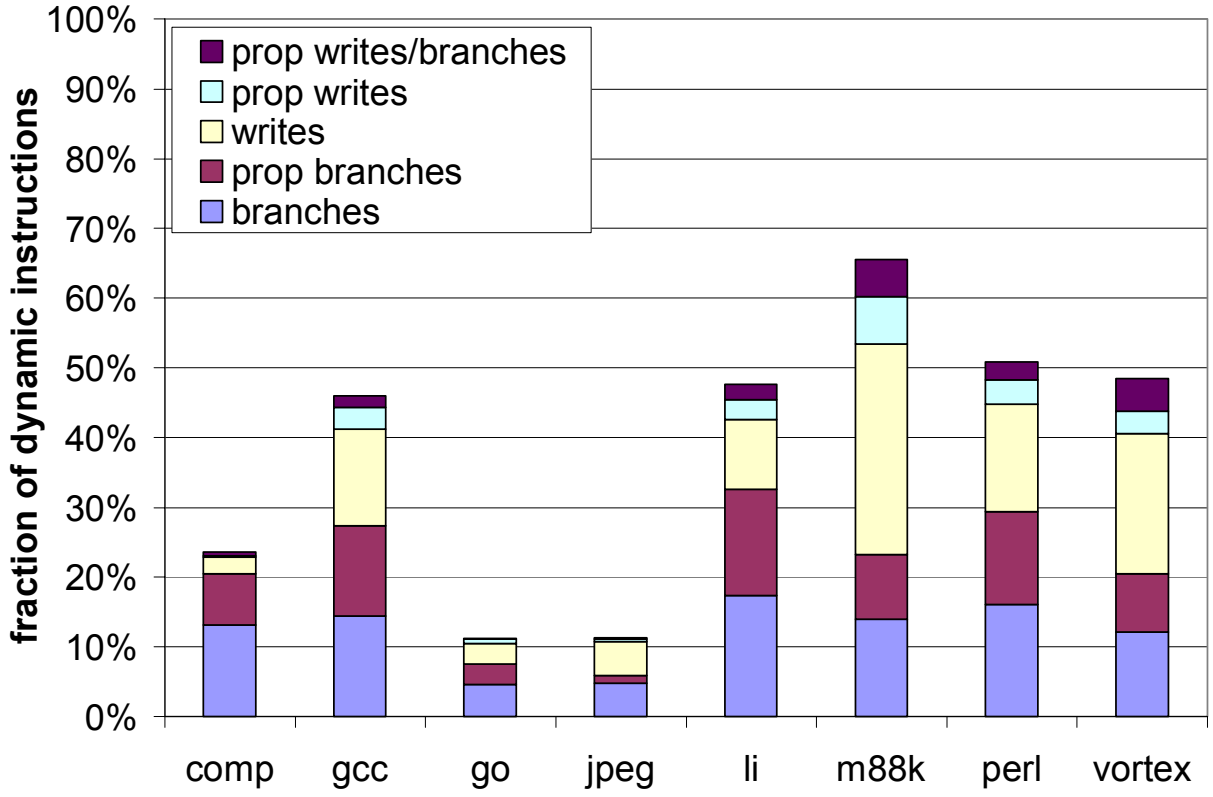

Figure 4-5. Breakdown of instruction removal.

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

Figure 4-5 also breaks down the reasons for instruction removal. On average, branches are the primary source, at just over a third of the removed instructions ("branches"). Ineffectual writes are about a third of removed instructions ("writes"). Among instructions removed due to

back-propagation ("prop —"), most are in dependence chains of removed branches ("prop branches").

4.2.3 Prediction

In Figure 4-6, we show the performance improvement of three models with respect to SS(64x4). The first is SS(64x4) with conventional value prediction added. A large context-based value predictor (CVP) [46] is used (2^{18} and 2^{20} entries in the first and second levels, respectively). The second is CMP(2x64x4)/byp, but the R-stream does not use A-stream values speculatively ("no value prediction"). The third is CMP(2x64x4)/byp.

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

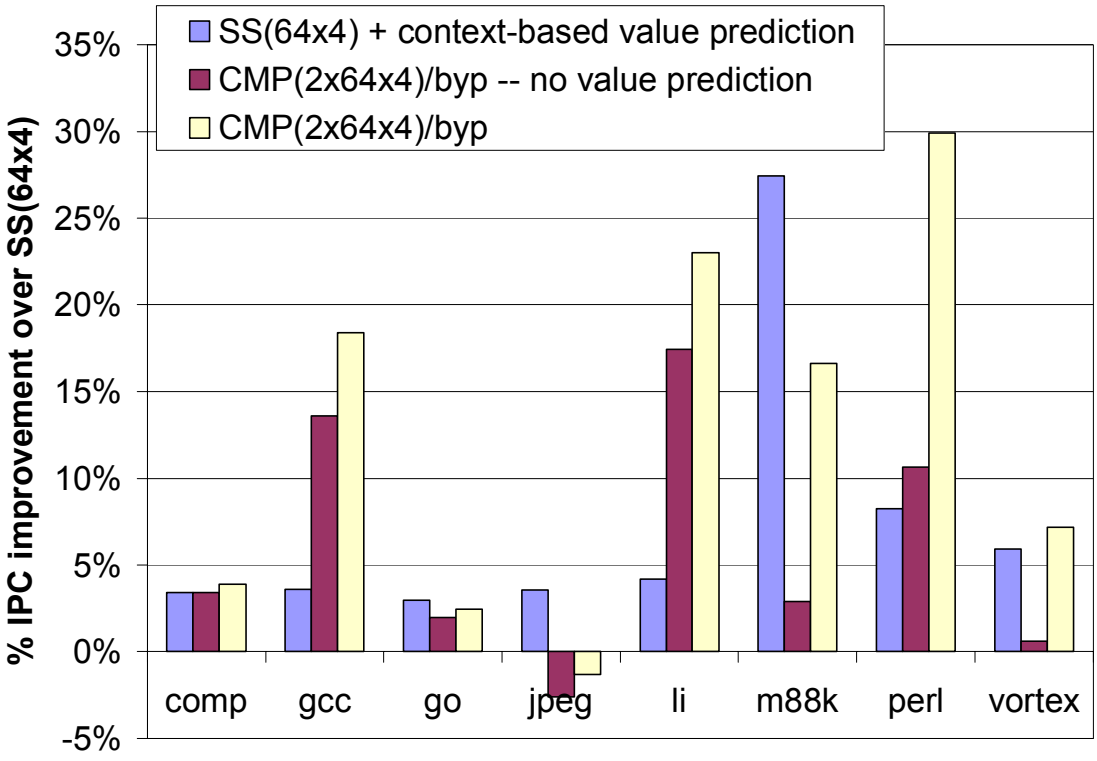

Figure 4-6. Measuring the relative importance of branch and value prediction benefits.

4.2.4 Other measurements

Branch mispredictions per 1000 instructions for each benchmark are provided in Table 4-3. A key observation is that instruction removal is most successful for benchmarks with low (m88k, vortex) to medium (gcc, li, perl) branch misprediction rates. In these benchmarks, it is easier for the confidence mechanism to distinguish between predictable and unpredictable dynamic branches. This is not true for benchmarks like compress and go, which have relatively high branch misprediction rates.

Slipstream execution extends the IR-predictor's update latency due to large slack between the A-stream and R-stream. Comparing branch mispredictions per 1000 instructions for SS(64x4) (conventional execution) and CMP(2x64x4) (slipstream execution) in Table 4-3, it can

be seen that extending the update latency does not significantly degrade branch prediction accuracy for most benchmarks, although it is noticeable. Jpeg is an exception – branch mispredictions per 1000 instructions increase from 4.31 to 5.46. According to Table 4-4, jpeg's average slack is low compared to other benchmarks, but the standard deviation is high, suggesting that there are regions of high slack.

Table 4-3. Branch mispredictions per 1000 instructions.

	comp	**gcc**	**go**	**jpeg**	**li**	**m88k**	**perl**	**vortex**
SS(64x4)	9.16	5.95	12.46	4.31	5.34	2.34	3.40	1.13
CMP(2x64x4)	9.16	6.26	12.54	5.46	5.45	2.55	3.52	1.26

The first row in Table 4-4 shows that a confidence threshold of 32 results in 0.02 (perl) to 0.38 (jpeg) IR-mispredictions per 1000 instructions. Thus, as expected, IR-mispredictions are relatively rare (e.g., compared to branch mispredictions). The second and third rows show how many of the IR-mispredictions per 1000 instructions are detected as branch mispredictions (control faults) and value mispredictions (data faults) in the R-stream, respectively. Except for vortex, most IR-mispredictions are manifested as control faults in the R-stream. Vortex has the lowest branch misprediction rate among all the benchmarks. With so few branch mispredictions to begin with, it is less likely for a mispredicted branch to be removed from the A-stream. Therefore, it is not surprising that there are few control faults in vortex.

The fourth row in Table 4-4 shows that the average recovery latency (after detecting an IR-misprediction) is at most 29 cycles, close to the minimum of 21 cycles, which implies only a handful of memory locations need to be restored after an IR-misprediction.

The last two rows in Table 4-4 show the average slack and average delay buffer length (standard deviations are shown in parentheses). Slack is the total number of instructions, both

skipped and non-skipped, separating the A-stream and R-stream. Delay buffer length is the number of non-skipped instructions (i.e., A-stream-retired instructions) in the delay buffer. The first observation is that the slack is large, in general. The second observation is that, not surprisingly, benchmarks with significant instruction removal also have relatively large slack and delay buffer lengths, whereas benchmarks with only 10-20% instruction removal do not have large slack.

Table 4-4. IR-misprediction rate, recovery latency, slack, and delay buffer length.

	comp	**gcc**	**go**	**jpeg**	**li**	**m88k**	**perl**	**vortex**
IR-misp./1000 instr.	0.1041	0.1714	0.2348	0.3808	0.2139	0.2250	0.0231	0.0865
control faults/1000 instr.	0.0960	0.1380	0.2106	0.3258	0.2064	0.1996	0.0177	0.0269
data faults/1000 instr.	0.0081	0.0334	0.0242	0.0550	0.0075	0.0254	0.0054	0.0596
recovery latency (cyc.)	21.92	22.93	21.31	21.28	24.79	23.86	26.45	28.92
avg. slack (instr.)	17.84	246.37	22.46	43.7	147.59	1483.62	342.05	269.83
(standard deviation)	(32.83)	(300.29)	(35.61)	(82.78)	(130.58)	(1562.89)	(202.75)	(204.36)
avg. delay buffer length	14.15	105.74	19.23	33.55	71.00	176.03	158.44	204.36
(standard deviation)	(23.02)	(96.40)	(22.14)	(48.67)	(63.13)	(96.96)	(86.7)	(78.79)

The distinction between benchmarks with significant removal and those with less removal is also evident from the delay buffer occupancy graph shown in Figure 4-7. The graph shows the fraction of cycles that the delay buffer contains 0 instructions, 1 instruction, and so on, up to the maximum occupancy of 256 instructions. For compress, go, and jpeg, delay buffer occupancy is highly clustered between 0 and 32 instructions. On the other hand, gcc, m88k, perl, and vortex have large clusters beyond 200 instructions. Li has the most uniform distribution among the benchmarks.

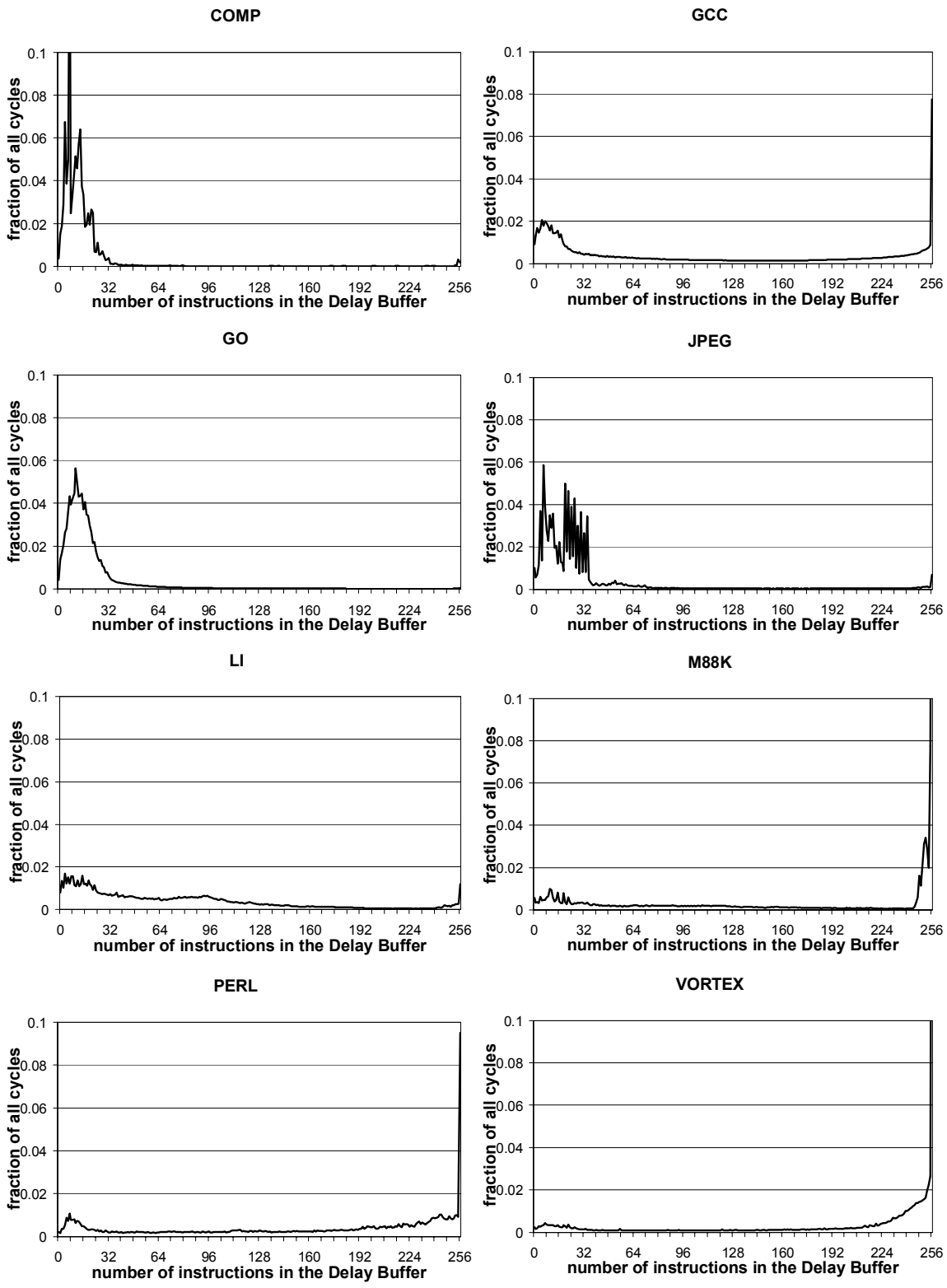

Figure 4-7. Delay buffer occupancy.

4.3 Summary of key results

- A 12% average performance improvement is achieved by harnessing an otherwise unused, additional processor in a CMP. Programs with little instruction removal are not sped up at all (and IR-mispredictions result in minor slowdowns, e.g., 1%), whereas programs with upwards of 1/3 instruction removal are sped up by as much as 30%.
- As more execution bandwidth is made available, slipstream execution provides less performance improvement. However, if the A-stream is able to bypass instruction fetching, slipstream retains its edge — because raw instruction fetch bandwidth is not as easily extended as raw execution bandwidth.
- Slipstream execution using two small superscalar cores often achieves similar IPC as one large superscalar core, but with a potentially faster clock and a more flexible architecture.
- For programs with sufficiently reduced A-streams, slipstream execution on an 8-issue SMT processor improves performance from 10%-20%.
- For some programs, performance improvement is due to the A-stream resolving branch mispredictions in advance. Others benefit largely from A-stream value predictions, and the effect is not always reproducible using conventional value prediction tables.

Chapter 5 Simple and Efficient Memory Management for Slipstream Execution on a CMP Substrate

A favorable attribute of slipstream is that its redundant programs are architecturally independent, leading to a simple execution model. Physical memory pages are duplicated by the operating system. As such, the processor does not need to explicitly manage a fixed amount of transparent rename storage for the leading program. However, this benefit is outweighed by the two-fold increase in memory usage.

In this chapter, we show it is possible to efficiently duplicate memory without explicit management. The leading program's state is confined to its private L1 cache and evicted lines are simply discarded whether or not they contain modified data. Discarding modified lines implies the leading program may later reference stale data. This possibility does not undermine correctness or performance. In terms of correctness, discarding modifications is as safe as removing predicted-ineffectual instructions, since correct operation is always assured by the trailing program. In terms of performance, the trailing program is close behind so discarded modifications are likely to be re-created and available in the shared L2 cache before the leading program re-references evicted lines.

The new duplication approach also enables much simpler state recovery when the speculative program diverges. Simple cache invalidation eliminates a previously-required slipstream recovery component. The performance impact of recovery-induced compulsory misses is

reduced by invalidating only dirty lines (trading thoroughness for efficiency) and/or exploiting preserved data within invalidated cache lines as highly-accurate value predictions.

5.1 Overview

The appeal of slipstream is that the A-stream and R-stream are executed by their respective processors as if they were unrelated, like different programs in a multiprogrammed workload. Conventional register and memory dependence mechanisms remain intact because the programs are architecturally independent. However, independence comes at the cost of doubling memory usage. That is, the easiest way to ensure A-stream loads/stores do not interfere with R-stream loads/stores is to have the operating system allocate separate physical memory pages for each program. In this way, the processor does not have to rename memory locations itself. Hardware memory renaming is typically complex because the processor must provide the illusion of unlimited memory to one of the program copies, within a fixed amount of hardware-managed renaming storage.

Therein lies the challenge in the design of memory systems for slipstream processors. On the one hand, software-based memory duplication leads to a simple execution model. On the other hand, full duplication increases pressure in the memory system and may degrade memory system performance.

In this chapter, we develop a hardware-based solution that duplicates memory efficiently, yet retains the simplicity of software-based memory duplication. We exploit a memory system like the one implemented in the IBM POWER4 [21][53], a commercial CMP composed of two processors. Each processor has a private level-1 (L1) cache, the L1 caches are backed by a shared level-2 (L2) cache, and the L1 caches implement a write-through policy. Our approach works as follows. First, the private L1 caches implicitly provide unique storage for the A-stream

and R-stream. Therefore, extra storage for renaming memory locations is not explicitly provided. Second, the A-stream L1 cache does not write-through to the L2 cache. If the A-stream writes to a line in its L1 cache, and later that line is replaced, *the update is simply lost*. Dropping A-stream L1 cache updates works due to the nature of slipstream execution. By the time the A-stream re-references the evicted line, the R-stream is likely to have performed the corresponding redundant store to its L1 cache and the shared L2 cache. When the A-stream re-references the line from the L2 cache, the line most likely reflects the previously-lost A-stream update. Moreover, even if the R-stream has not performed the corresponding redundant store before the A-stream re-references the line, the A-stream is speculative in any case and occasional references to stale data just adds another source of A-stream misspeculation.

From the perspective of hardware, the above approach is virtually identical to software-based memory duplication. As before, no special hardware mechanisms are required, in that (1) rename storage is not explicitly provided due to already-replicated L1 caches, and (2) the fixed A-stream storage is not explicitly managed because *renaming does not have to be 100% accurate*. The second point is a key departure from conventional renaming approaches. Conventionally, hardware determines when storage can be safely freed. In the context of slipstream execution, it is safe to "free" an A-stream L1 line as soon as the corresponding R-stream L1/L2 line becomes redundant with it. But this condition does not have to be explicitly identified because we can afford to be incorrect. *A general mechanism is in place to detect A-stream deviations, regardless of the cause*. Instead, A-stream storage is freed when an L1 line is replaced, whether or not it is safe to do so at the time, essentially making a *prediction* at replacement time that either the L2 line is redundant with the A-stream L1 line or will be before re-referencing it.

The second contribution of this chapter is simplifying restoring of memory locations when the A-stream diverges, including eliminating a previously required slipstream component (the memory recovery controller). Occasionally, the A-stream does not make correct forward progress and its state becomes corrupted. A recovery sequence restores A-stream state so it matches the R-stream. The entire register file can be restored quickly because it is small. However, we cannot quickly restore the entire memory image. Therefore, special hardware pin-points memory locations that actually need to be restored [52]. With software-based memory duplication, sophisticated recovery is unavoidable because A-stream and R-stream physical memory pages are distinct. However, with our new duplication approach, A-stream memory can be restored simply by invalidating the A-stream L1 cache. Then, A-stream and R-stream memory match exactly since the A-stream must re-reference all data in the L2 cache, which is R-stream-only data. We also develop a novel technique for reducing the performance impact of recovery-induced compulsory misses in the A-stream. A line is invalidated by resetting its valid bit, but both the tag and data are preserved. Preserved values are usually correct and may be consumed as accurate value predictions, allowing load-miss-dependent instructions to execute while the invalidated lines are re-filled from the L2 cache.

The two alternative memory duplication models are described in Section 5.2. Three recovery models are described in Section 5.3, including the original recovery controller and two simpler invalidation models enabled by hardware-based memory duplication. Section 5.4 summarizes the advantages and disadvantages of the various duplication and recovery methods, including a table for quick reference. Section 5.5 describes the simulator and benchmarks. Simulation results comparing duplication and recovery methods are presented in Section 5.6. Finally, the chapter is summarized in Section 5.7. (Related work can be found in Section 2.3.)

5.2 Memory Duplication Models

Slipstream execution requires memory duplication so that A-stream loads/stores and R-stream loads/stores do not interfere with each other. We examine both software-based duplication and our new hardware-based duplication approach, in Sections 5.2.1 and 5.2.2, respectively.

5.2.1 Software-based memory duplication

In previous slipstream implementations [36][40][41][52], the operating system (O/S) duplicated physical memory pages to separate the two redundant programs. Software-based memory duplication is shown in Figure 5-1. Light shading and dark shading indicate data from A-stream pages and R-stream pages, respectively. As the figure indicates, a major drawback of software-based duplication is increased pressure in the memory system. Some or all of the performance improvement due to slipstream execution may be negated due to additional capacity/conflict misses throughout the memory hierarchy.

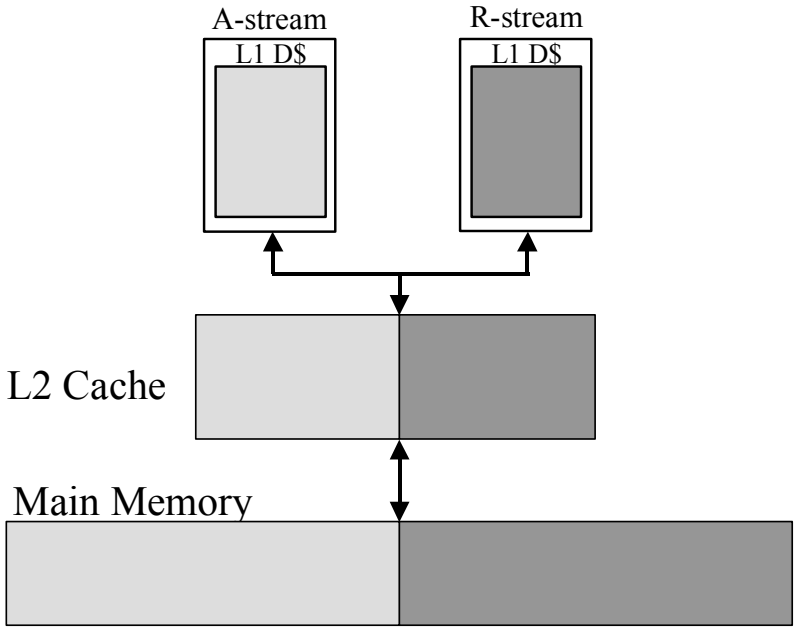

Figure 5-1. Software-based memory duplication.

Another major drawback of software-based memory duplication is that it requires O/S support. The O/S must duplicate the original program's physical memory image and manage separate page tables for the A-stream and R-stream. Also, exception handling requires unique O/S support (exceptions include program exceptions, system calls or traps, and external interrupts). Slipstream processors synchronize the A-stream and R-stream at exceptions [40]. After that, the O/S has two options. In the first option, the O/S preserves both programs, i.e., both programs are swapped out and later swapped in. The O/S handles the exception as it would for one program, but duplicates the results as needed for both suspended contexts. For example, file input/output (I/O) is not duplicated but the memory state of both contexts should reflect the results of file I/O. In the second option, one of the program copies is terminated by the O/S before servicing the exception. The remaining program is swapped out and the exception is serviced in the usual manner. To restart, the suspended program is duplicated and both copies are swapped in. The first option complicates servicing of exceptions, whereas the second option incurs high performance overhead because all pages must be copied after every exception. The fact that the O/S is involved at all is undesirable.

5.2.2 Hardware-based memory duplication

We will demonstrate hardware-based memory duplication in the context of the dual-processor POWER4 memory system [53]. The POWER4 on-chip memory system consists of two private L1 data caches (one per processor) and a shared L2 cache. The L1 caches are write-through (a write-through policy enhances fault tolerance and simplifies L1 cache coherence [53]). The shared L2 cache maintains coherency of the L1 caches, both with respect to each other and the rest of the system. A store by one of the processors causes the L2 cache to issue an

invalidation request to the other processor's L1 cache if it has a copy of the line (this is called "back-invalidation" [53]).

We now turn to support for slipstream execution. The A-stream reaches store instructions before the R-stream, so R-stream memory state lags slightly behind A-stream memory state — but not by much. The redundant programs are typically no more than a few thousand instructions apart, and often less. This means the vast majority of A-stream and R-stream physical pages in Figure 5-1 are identical, and only a small amount of duplication is required. Therefore, the private L1 caches already provide enough memory replication. Two minor changes are needed regarding the handling of processor stores.

1. A-stream stores to the L1 cache are *not* also performed in the L2 cache (the write-through policy is disabled for the A-stream). If a "dirty" L1 cache line (a line that has been written to) needs to be evicted to make room for another cache line, the line is *not* written back to the L2 cache. The evicted cache line, and the updated data it contains, is simply lost. In short, the A-stream can read from the L2 cache but not write to it.
2. The L2 cache does *not* issue a back-invalidation request to the A-stream L1 cache if the R-stream stores to a line that is cached by the A-stream L1 cache. (Back-invalidation requests to the R-stream L1 cache are implicitly prevented because the A-stream does not propagate stores to the L2 cache, according to item 1 above.) The A-stream and R-stream copies of the line are not truly the same line, so coherence should not be maintained.

Figure 5-2 shows hardware-based memory duplication. As before, A-stream state is indicated with light shading and R-stream state with dark shading. A thin black rectangle in the A-stream L1 cache represents a cache line to which the A-stream has performed a store, and the

R-stream has not yet performed its corresponding redundant store. The figure shows how one such cache line written by the A-stream is evicted from the cache and the update it contains is lost, because the A-stream does not have its own renamed state beyond the L1 cache.

Evicting and losing A-stream updates is rarely a problem because the R-stream usually reproduces the data before the A-stream re-references it. The R-stream is generally not far behind and, because its L1 cache is write-through, the evicted-and-lost A-stream data is re-created in the L2 cache before the A-stream needs it again. Occasionally, the A-stream re-references the line in the L2 cache before the R-stream has performed its corresponding update. The A-stream gets stale data, but the A-stream is speculative in any case and A-stream mispredictions are recoverable. In Section 5.6, we measure how many stale bytes are consumed by the A-stream; we label this measurement *stale* in Section 5.6.

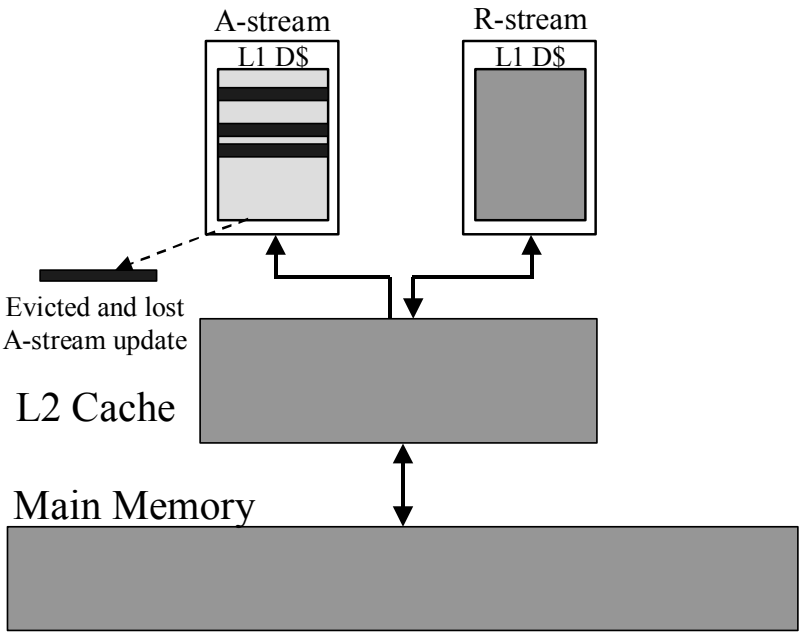

Figure 5-2. Hardware-based memory duplication.

Losing A-stream data can occasionally aid the A-stream. The A-stream sometimes incorrectly skips a store or incorrectly performs a store. In either case, the corrupt cache line

may be evicted before being referenced and, later, the A-stream references a correct version of the cache line from the L2 cache. In Section 5.6, we measure how many inadvertently-repaired bytes are referenced by the A-stream; we label this measurement *self-repair* in Section 5.6.

In summary, our new approach is as simple as before because the processor does nothing explicit to manage A-stream storage. Instead, inherent cache actions perform the desired operations: the A-stream *implicitly* duplicates memory by allocating lines in its L1 cache and *implicitly* frees memory by replacing lines in its L1 cache. The approach is also efficient: memory system performance is not impacted because duplication is limited to the already-replicated L1 cache.

5.2.3 Regarding system interaction

A CMP used to execute a sequential program in slipstream mode may be part of a larger multiprocessor system or a system with DMA devices. The key point is that the speculative A-stream is safely quarantined from the rest of the system, whereas the non-speculative R-stream interacts with the rest of the system as usual. A-stream L1 cache misses only have the external effect of generating prefetches and allocating lines in the L2 cache. Subsequent writes by the A-stream are confined to its L1 cache. If the A-stream has a memory block in its L1 cache for which an invalidation request is received from another node or DMA device, the block is discarded like usual for a write-through L1 cache. Finally, there are various situations which may cause the A-stream and R-stream to diverge (critical sections are an example). It is not necessary to describe any case in particular because under no circumstances can the quarantined A-stream modify memory.

So far, there have been no additional changes beyond the modifications already described in Section 5.2.2 to safely quarantine the speculative A-stream from the rest of the system,

specifically the point that A-stream writes are not propagated to the L2 cache. The only other change is that loads and stores to memory-mapped devices are discarded in the A-stream.

Although we framed this discussion in terms of sequential programs, it also applies to parallel programs. However, a modified form of slipstream execution should be employed in the case of parallel programs in order to effectively enhance their performance. For parallel programs, we refer the reader to related work that applies redundant execution for enhancing multiprocessor performance [18].

5.3 Recovery Models

This section describes three memory recovery models, including the recovery controller and two much simpler cache-invalidation approaches enabled by hardware-based duplication. Also, we propose an optimization that reduces the performance impact of recovery-induced compulsory misses for the two cache-invalidation approaches.

5.3.1 Recovery controller

The *recovery controller* [52] monitors store activity in the A-stream, R-stream, and IR-detector, pin-pointing memory locations that need to be restored from the R-stream if the A-stream diverges. It maintains a list of memory locations, identified by address, that are known to differ between the A-stream and R-stream. Actually, there are many locations that differ but do not affect program correctness. The recovery controller only tracks memory locations that differ and have not yet been verified as "OK" to differ. Our implementation keeps track of individual doublewords, although a word or cache line granularity could also be used.

Figure 5-3 demonstrates how the recovery controller tracks memory locations. The figure shows the progression in time (from left to right) of two different types of stores as they pass first through the A-stream, then through the R-stream, and finally through the IR-detector. The contents of the recovery controller is shown evolving over time, at the bottom of the figure. The first store is to address *A* and is not skipped by the A-stream. It is shown with a solid circle to indicate it was not skipped. The second store is to address *B* and is skipped by the A-stream. It is shown with a hollow circle to indicate it was skipped.

When store(*A*) is committed by the A-stream, address *A* is added to the recovery controller because that location now differs between the A-stream and R-stream. When the R-stream commits the corresponding redundant store(*A*), address *A* is removed from the recovery

controller because it no longer differs. If the processor initiates a recovery sequence after store(*A*) is committed by the A-stream and before it is committed by the R-stream, we know to "undo" the A-stream store because address *A* is in the recovery controller's list.

Store(*B*) is skipped by the A-stream so no signal is sent by it to the recovery controller. The R-stream knows which instructions were skipped by the A-stream (as described in Section 3.2, the delay buffer contains removal information for matching up results of A-stream-executed instructions with R-stream instructions). When store(*B*) is committed by the R-stream, address *B* is added to the recovery controller because the A-stream perhaps should have performed the store. By the time store(*B*) has reached the end of the IR-detector's analysis scope, we will have determined whether or not store(*B*) was necessary. If store(*B*) is deemed ineffectual, address *B* is removed from the recovery controller. Otherwise, it remains in the recovery controller until the next recovery sequence since we do not know for certain that it was alright to skip store(*B*).

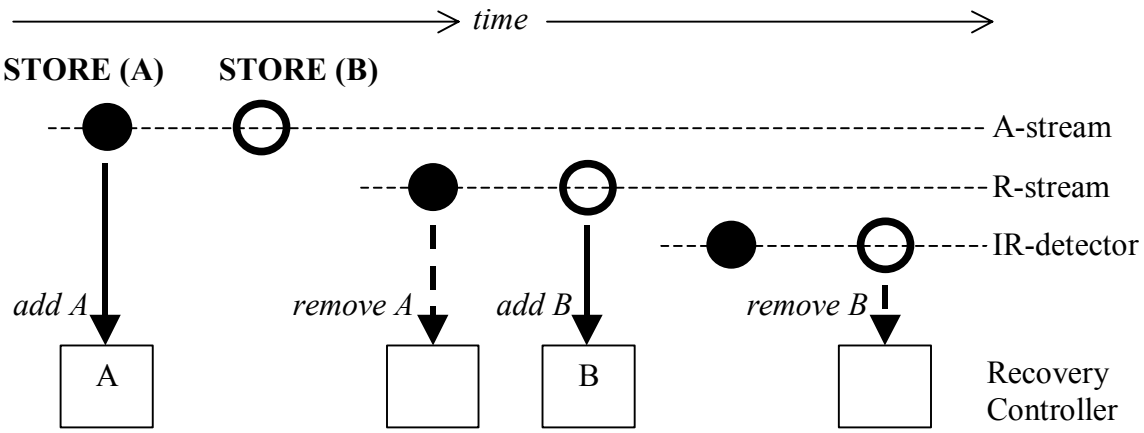

Figure 5-3. Recovery controller operation.

The recovery controller is organized as a set-associative or fully-associative buffer. Each entry contains a memory address and two counters, the *store-undo* counter and the *store-do* counter. Counters are used because there can be multiple unverified stores to the same location, all of which must be tracked. Referring back to Figure 5-3, the *store-undo* counter for address *A*

is incremented when store(*A*) is committed by the A-stream and decremented when store(*A*) is committed by the R-stream. The *store-do* counter for address *B* is incremented when store(*B*) is committed by the R-stream and decremented when store(*B*) is detected as removable by the IR-detector. An entry in the recovery controller can be replaced when both the *store-undo* and *store-do* counts are zero. If there is no room for a new address in the recovery controller, a recovery sequence is initiated to clear out the recovery controller.

The recovery controller is required if software-based duplication is used. It also works with hardware-based duplication but is optional in this case, because simpler recovery models are enabled by efficient duplication. Actually, the recovery controller does not recover state perfectly when used in conjunction with hardware-based duplication. Namely, *the recovery controller has no knowledge of stale lines brought into the A-stream L1 cache* (A-stream evicts and loses an updated line, and re-references a stale version of the line from the L2 cache). After a recovery, stale data may persist in the cache and potentially cause problems for the A-stream in the future. In Section 5.6, we measure how many stale bytes introduced before recovery are referenced by the A-stream after recovery; we label this measurement *persistent-stale* in Section 5.6.

5.3.2 Invalidate cache

The recovery controller adds complexity to slipstream processors. The new hardware-based memory duplication approach can be exploited to eliminate the recovery controller. A-stream memory state can be restored, i.e., re-synchronized with the memory state of the R-stream, simply by invalidating *en masse* the A-stream's L1 cache lines. Recovery is gradual as lines are re-accessed from the L2 cache, which contains correct and up-to-date R-stream memory state.

The only hardware support for this recovery method is a global invalidation control signal that resets the valid bit of all cache lines. Also, whereas the recovery controller is slightly imperfect due to the *persistent-stale* case, invalidating the cache is completely effective.

5.3.3 Invalidate dirty lines

Invalidating all lines in the A-stream L1 cache is inefficient because only a handful of memory locations are typically corrupted. There are undue compulsory misses after recovery. To reduce A-stream compulsory misses, we propose invalidating only *dirty* lines. Any data written to the cache after the A-stream diverges may be incorrect. And even if data is written correctly, our recovery strategy requires stores not yet performed in the R-stream to be "undone" in the A-stream at the time of recovery (to re-synchronize state). Thus, lines that become dirty after the A-stream diverges are good candidates for invalidating. Unfortunately, lines that were dirty before the A-stream diverged and not subsequently written to are needlessly invalidated.

As with the recovery controller, invalidating dirty lines leads to imperfect recovery. First, stale lines brought in from the L2 cache that are not subsequently written to (clean) will persist after recovery. So, invalidating dirty lines also suffers the *persistent-stale* problem. Second, clean lines that are corrupt due to incorrectly-skipped stores persist after recovery (the recovery controller, on the other hand, tracks addresses of correctly- and incorrectly-skipped stores). In Section 5.6, we measure how often bytes corrupted by a skipped-store before recovery are referenced by the A-stream after recovery; we label this measurement *persistent-skipped-write* in Section 5.6.

The hardware support for this recovery method is (1) dirty bits in the L1 cache and (2) a global invalidation control signal gated by the dirty bit that resets the valid bit of dirty cache lines.

5.3.4 Reducing impact of recovery-induced misses: Value prediction using invalidated cache data

For either recovery method in Sections 5.3.2 and 5.3.3, invalidating a cache line resets its valid bit but preserves its tag and data. The preserved tag and data can be exploited to reduce the impact of recovery-induced compulsory misses in the A-stream. When the A-stream accesses an invalid line, the cache miss is serviced like usual. However, the preserved cache tag(s) are still checked and, if there is a match, the A-stream retrieves a value from the cache and uses it as a value prediction. The value prediction is eventually validated when the cache miss completes.

Value predicting a load miss in this way gives some performance benefit, even if the load reaches the head of the reorder buffer and stalls A-stream retirement while waiting for the cache miss to complete. First, execution of dependent instructions is not delayed and this results in a faster retirement rate when retirement eventually resumes. Second, the load is unlikely to stall retirement for too long, if at all, because the invalidated line is likely to be in the L2 cache if it was found in the L1 cache. Third, other recovery-induced misses potentially initiate earlier. Finally, value predictions are nearly 100% accurate because, typically, only a handful of lines are corrupted when the A-stream diverges.

5.4 Qualitative Comparisons of Duplication and Recovery Methods

Table 5-1 summarizes the advantages, disadvantages, and required hardware support of the two memory duplication methods (top-half) and three memory recovery methods (bottom-half). Notice the four useful measurements introduced in Sections 5.2 and 5.3 are highlighted in bold italics: *stale* and *self-repair* relate to memory duplication, and *persistent-stale* and *persistent-skipped-write* relate to recovery. Results in Section 5.6 quantify much of the information summarized in Table 5-1.

Note that the cache-based value prediction technique is not listed in Table 5-1, but is used in conjunction with either invalidation-based recovery model to reduce the performance impact of recovery-induced misses.

Figure 5-4 shows the original slipstream microarchitecture with software-based memory duplication. Figure 5-5 shows the new slipstream microarchitecture using hardware-based memory duplication and either of the invalidation-based recovery models. The new microarchitecture does not need a memory recovery controller, so now there are only three slipstream components — the IR-predictor, IR-detector, and delay buffer. Notice the A-stream only reads from the L2 cache and the R-stream reads and writes the L2 cache.

Table 5-1. Qualitative comparisons of duplication and recovery methods.

		POSITIVES	NEGATIVES	HARDWARE SUPPORT
memory duplication method	software-based	+ simple state renaming	- double memory usage - requires recovery controller - hard system-level issues	none
	hardware-based	+ as simple as s/w-based + efficient memory usage + enables simpler recovery + system-transparent + ***self-repair***	- ***stale***	No explicit hardware mechanisms, only assumes IBM-POWER4-like CMP memory hierarchy
memory recovery method	recovery controller	+ restore data without invalidating line from cache	- adds h/w complexity - explicitly increases recovery latency - force recovery when full - imperfect recovery: ***persistent-stale***	recovery controller mechanism
	invalidate	+ simple + 100% recovery	- many compulsory misses	invalidate wire
	invalidate dirty lines	+ simple + invalidate fewer lines	- some compulsory misses - imperfect recovery: ***persistent-stale*** ***persistent-skipped-write***	invalidate wire, dirty bits

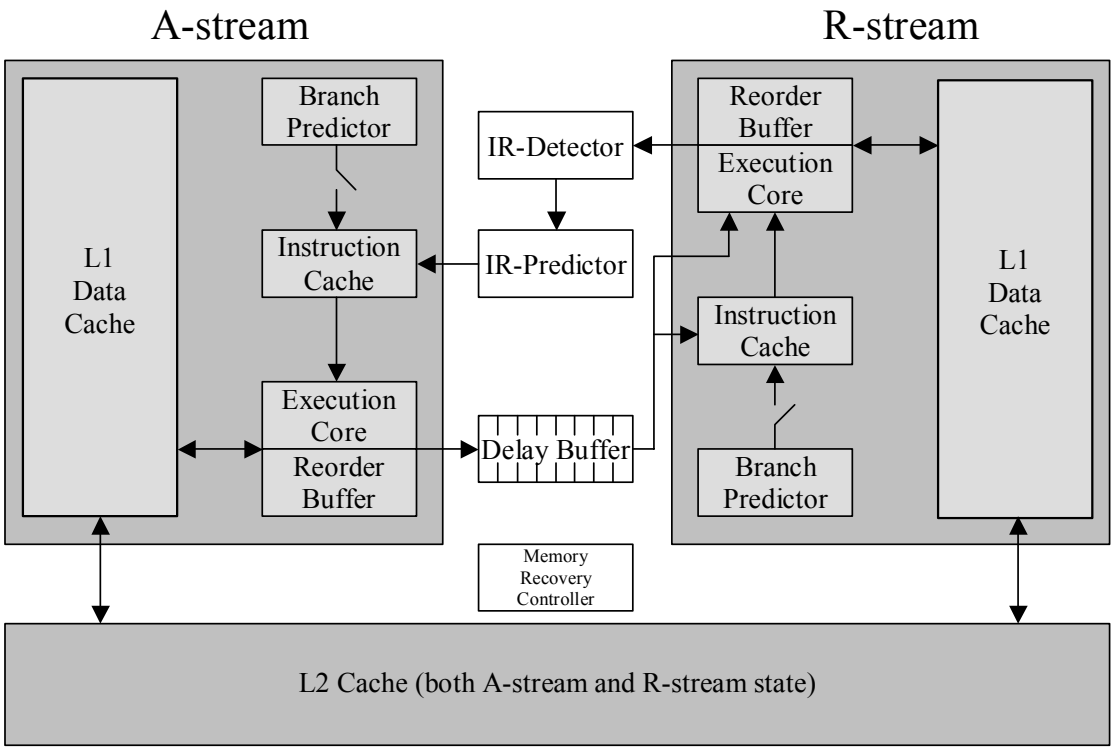

Figure 5-4. Original slipstream microarchitecture.

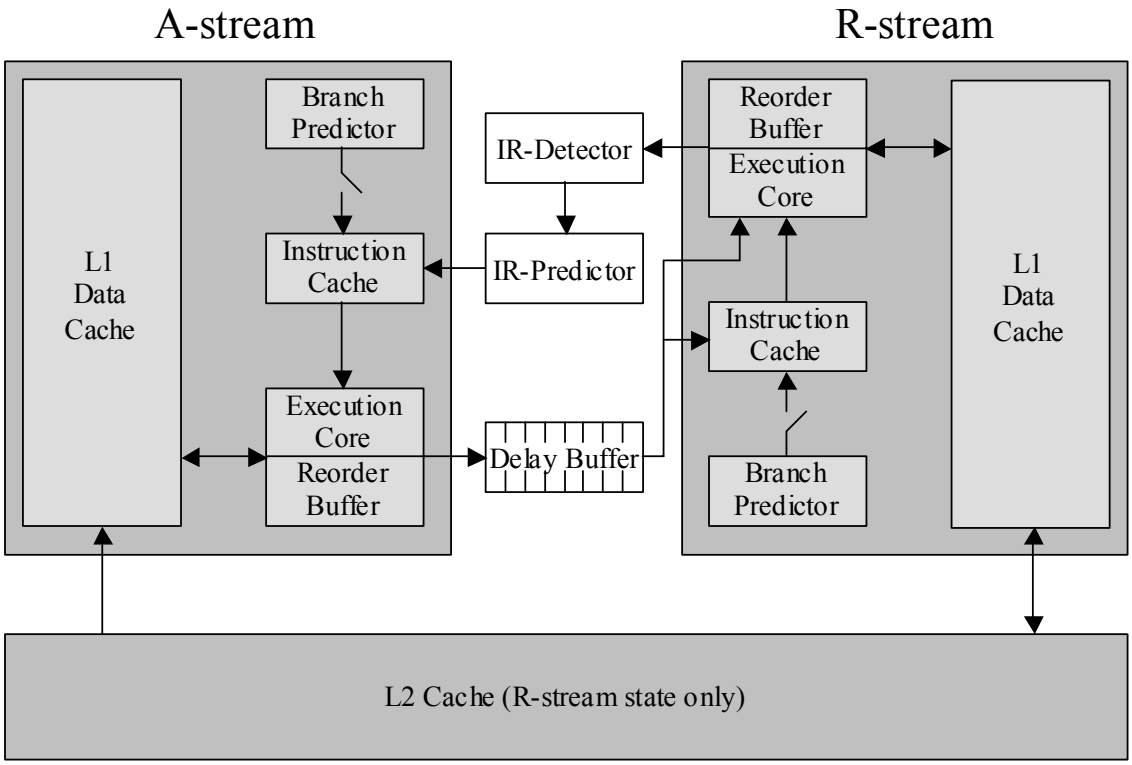

Figure 5-5. New slipstream microarchitecture with hardware-based memory duplication and invalidation-based recovery.

5.5 Simulation methodology

We use a detailed execution-driven simulator of a slipstream processor. The simulator faithfully models the microarchitectures depicted in Figure 5-4 (software-based duplication) and Figure 5-5 (hardware-based duplication): the A-stream produces real, possibly incorrect values/addresses and branch outcomes, the R-stream checks the A-stream and initiates recovery actions, A-stream state is recovered from the R-stream state, etc. The simulator itself is validated via a functional simulator run independently and in parallel with the detailed timing simulator [48][42]. The functional simulator checks retired R-stream control flow and data flow outcomes.

Microarchitecture parameters are listed in Table 5-2. The top-left portion of the table lists parameters for individual processors within a CMP. The bottom-left portion describes the four slipstream components. We use the same parameters as in previous work [36], to which the reader is referred for more details, and focus on the slipstream memory system (right-hand side).

The per-processor L1 data cache size/associativity is varied. The shared L2 cache is 256KB, 4-way set-associative, and holds both instructions and data. Instruction pages are read-only so they are not duplicated even if software-based duplication is used. We also reduce A-stream/R-stream conflicts in the L2 cache for software-based duplication by inverting the high index bit for R-stream accesses (otherwise, the two address streams are too alike). An L1 data cache hit is 2 cycles, an L1 miss/L2 hit takes 12 cycles, and round-trip time to main memory is a minimum of 70 cycles.

Finally, we vary the memory duplication method and recovery method. The recovery controller (if present) holds 128 addresses and is fully-associative. Independent of the memory recovery method, recovery latency (*after* the IR-misprediction is detected) is 5 cycles to startup

the recovery pipeline followed by 4 register restores per cycle (a total of 21 cycles). An additional latency of 4 memory restores per cycle is incurred *if the recovery controller is used*. For invalidation-based recovery, we assume the global invalidation signal can invalidate the cache in a cycle. However, cache invalidation only needs to be as quick as register file recovery (21 cycles, as described above) since the two can be overlapped.

Table 5-2. Microarchitecture configuration.

single processor core (PE)		**slipstream memory hierarchy**	
caches	private L1 instr. cache (*see memory hier. column*)	**L1 instruction cache (per PE)**	size = 64 KB
	private L1 data cache (*see memory hier. column*)		assoc. = 4-way
superscalar core	reorder buffer: 64 entries		replacement = LRU
	dispatch/issue/retire bandwidth: 4 instr/cycle		line size = 64 bytes
	4 universal function units	**L1 data cache (per PE)**	size = 8 KB / 32 KB / 64 KB
	4 loads/stores per cycle		assoc. = 1-way / 4-way
execution latencies	address generation = 1 cycle		replacement = LRU
	load access = 2 cycles (hit)		line size = 64 bytes
	integer ALU operations = 1 cycle	**L2 cache**	unified instr./data
	complex operations = MIPS R10000 latencies		shared among PEs
slipstream components			size = 256 KB
IR-predictor	2^{20} entries, *gshare*-indexed (16 bits branch history)		assoc. = 4-way
	block size = 16, 16 confidence counters per entry		replacement = LRU
	confidence threshold = 32		line size = 64 bytes
IR-detector	R-DFG = 256 instructions, unpartitioned		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 controller	128 entries, fully-associative		L1 miss/L2 hit = 12 cycles
	recovery latency (*after* IR-misprediction detected): • 5 cycles to start up recovery pipeline • 4 reg. restores/cycle (64 regs performed 1st) • 4 mem. restores/cycle (mem performed 2nd) • ∴ min. latency (no memory) = 21 cycles		L1 miss/L2 miss = 70 cycles
		# out. misses	unlimited for all caches
		DUPLICATION	software- or hardware-based
		RECOVERY	recovery controller, inv., inv.-dirty, or inv./inv.-dirty with value prediction

The Simplescalar [5] compiler and ISA are used. We use eight SPEC2000 integer benchmarks compiled with -O3 optimization and run with *ref* input datasets (Table 5-3). The first billion instructions are skipped and then 100 million instructions are simulated. Of the four other integer benchmarks not used, two did not compile with the Simplescalar compiler (*crafty*, *eon*) and the other two exceeded virtual memory of our largest machines (*bzip*, *mcf*) because we

have to maintain several full memory images to measure the number of *stale*, *self-repair*, *persistent-stale*, and *persistent-skipped-write* references (this is a statistics-gathering issue).

Table 5-3. Benchmarks.

benchmark	ref input dataset
gap	-l ./ -q -m 8M ref.in
gcc	cccp.i -o cccp.s (note: SPEC2K version of cc1 is hardwired to -O3 optimization)
gzip	input.program 16
parser	2.1.dict -batch < ref.in
perl	-I./lib splitmail.pl 850 5 19 18 1500
twolf	ref
vortex	bendian1.raw
vpr	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

5.6 Results

Slipstream execution provides a means to use a second processor in a CMP to accelerate a single program. Accordingly, all results are reported as the speedup of slipstream execution using two processors of the CMP with respect to conventional execution on one of the processors. Except for L1 data cache size and set-associativity, the processor is fixed and is called the BASE configuration. BASE is a 4-issue dynamically scheduled superscalar processor with a 64-instruction reorder buffer, as described in Section 5.5.

We first compare software-based and hardware-based memory duplication (Section 5.6.1), demonstrating that the hardware-based approach is required for materializing slipstream performance. We then investigate five recovery models within the context of hardware-based memory duplication (Section 5.6.2). In all, we simulate six slipstream configurations, labeled with the memory duplication method — SD for (s)oftware-based (d)uplication and HD for (h)ardware-based (d)uplication — followed by the recovery model in parentheses — "rc" = recovery controller, "inv" = invalidate entire cache, "inv-vp" = invalidate entire cache and use as value predictions, "invd" = invalidate dirty lines in cache, "invd-vp" = invalidate dirty lines in cache and use as value predictions. The six configurations are SD(rc), HD(rc), HD(inv), HD(inv-vp), HD(invd), and HD(invd-vp). Recall that the recovery controller is the only valid recovery method for software-based memory duplication.

5.6.1 Software-based vs. hardware-based memory duplication

The instructions-per-cycle (IPC) performance improvement of SD(rc) and HD(rc) with respect to BASE are shown in Figure 5-6. There is one graph per benchmark and L1 data cache configuration is varied along the x-axis (for example, 8k-1 is an 8KB direct-mapped cache, 32k-4 is a 32KB 4-way set-associative cache).

Based on the results in Figure 5-6, efficient memory duplication is required to materialize slipstream performance. HD(rc) almost always outperforms SD(rc), and by large margins in all benchmarks except *gap* and *perl*. The SPEC2000 benchmarks place moderate stress on the memory hierarchy. Consequently, SD(rc) takes a large performance hit because it doubles the number of physical pages competing for the already highly-utilized memory hierarchy.

The performance impact of full duplication in *gzip*, *parser*, *twolf*, and *vpr* is large enough to degrade performance with respect to BASE by up to 14%. On the other hand, HD(rc) improves performance by about 17% for *parser*, 7% for *vpr,* and 5% for *gzip*, with *twolf* performance neither increasing nor decreasing due to slipstream execution. SD(rc) improves performance by about 8% and 5% for *gcc* and *vortex*, respectively. However, HD(rc) is able to increase those speedups to as high as 13% and 22%, respectively.

The *gcc*, *gzip*, *perl*, and *vortex* benchmarks, and to a lesser extent *gap*, *twolf*, and *vpr*, show interesting trends for HD(rc) as cache size and set-associativity are increased. First, performance improvement increases steadily with progressively larger direct-mapped caches. Yet, performance improvement is relatively constant with cache size if the cache is 4-way set-associative. Second, there is a jump in performance improvement when associativity is increased from direct-mapped to 4-way set-associative.

These trends can be explained by examining the number of *stale* bytes referenced per 1000 instructions using HD(rc), shown in Figure 5-8 averaged across all benchmarks. *Stale* byte references are extremely rare (fewer than 0.001 stale bytes per 1000 instructions) for set-associative caches. In contrast, direct-mapped caches result in a comparatively high rate of *stale* byte references (e.g., 0.67 stale bytes per 1000 instructions for the 8 KB direct-mapped cache). A direct-mapped cache typically has more conflict misses than a set-associative cache. Conflict

misses result in many evicted-and-lost line updates that are re-accessed in the L2 cache too soon, before the R-stream has a chance to re-create the lost data (resulting in additional, costly A-stream mispredictions). Evictions in set-associative caches are more likely to be caused by capacity misses than conflict misses, in which case the evicted-and-lost line update is less likely to be re-accessed soon. We conjecture that an A-stream victim cache [20] will improve HD(rc) performance improvement with direct-mapped caches. The same analysis explains why HD(rc) performance improvement is sensitive to L1 cache size for direct-mapped caches and not for 4-way set-associative caches. The *stale* problem always exists, but decreases, as direct-mapped cache size is increased.

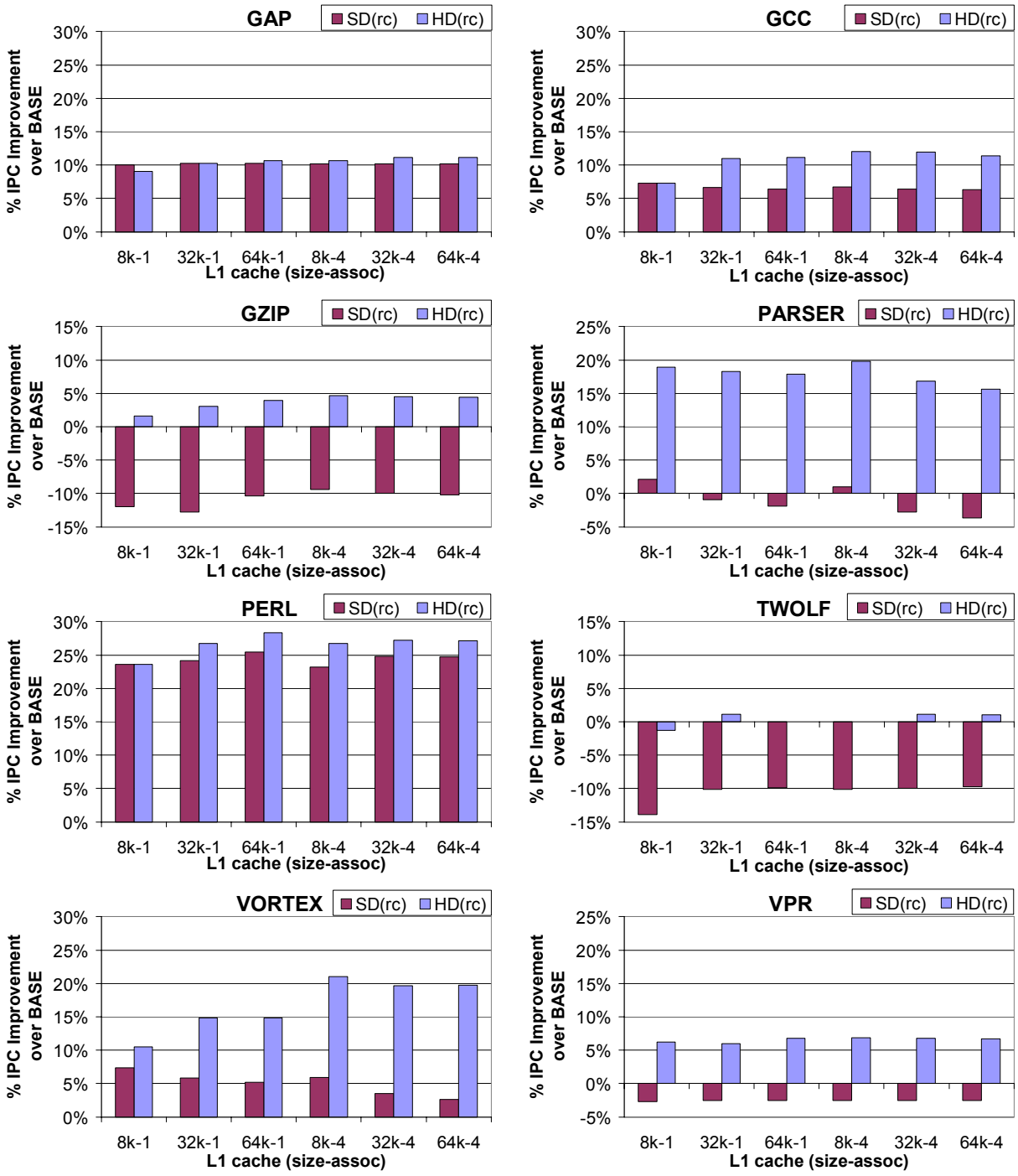

Figure 5-6. Comparison of duplication methods: performance of SD(rc) and HD(rc) with respect to BASE.

In Figure 5-8, the number of *self-repair* bytes referenced per 1000 instructions shows exactly the same trend as *stale* bytes referenced, i.e., direct-mapped caches exhibit a lot of *self-repair*.

This is to be expected, since conflict misses can actually be beneficial in terms of evicting corrupt lines before they are referenced.

5.6.2 Recovery model results

The IPC performance improvement of HD(inv), HD(inv-vp), HD(invd), HD(invd-vp), and HD(rc) with respect to BASE are shown in Figure 5-7, averaged across all the benchmarks. As before, L1 data cache configuration is varied along the x-axis. We only discuss results averaged across all benchmarks, however, per-benchmark results are shown in Figure 5-9.

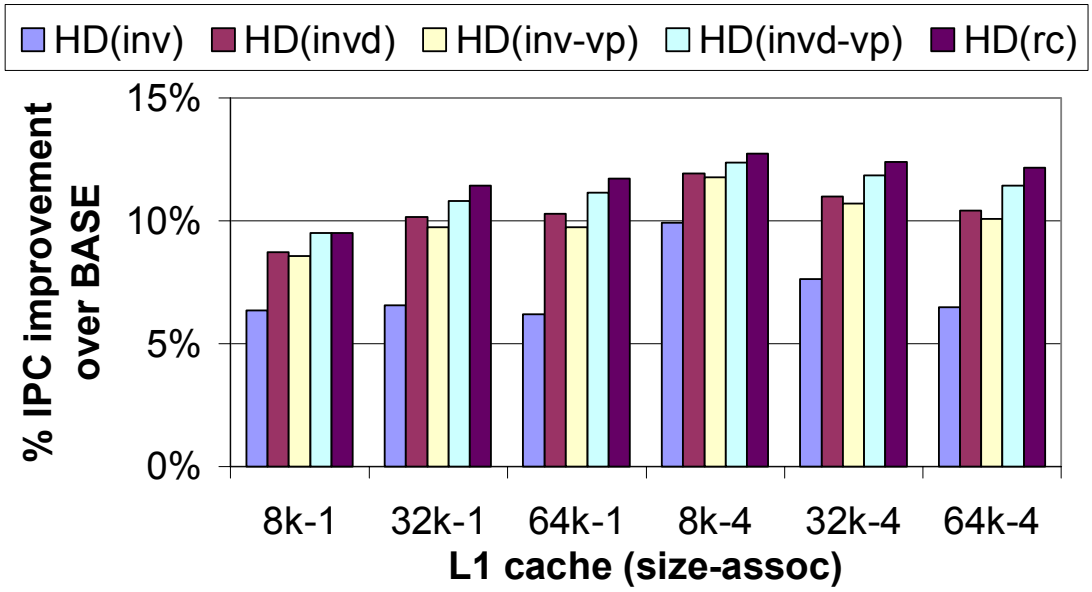

Figure 5-7. Comparison of recovery methods for hardware-based memory duplication, averaged across all benchmarks.

From Figure 5-7, on average, HD(rc) almost always performs best. This is to be expected because the recovery controller is both accurate and efficient, by virtue of pin-pointing corrupt data words.

The reason HD(rc) does not outperform HD(invd-vp) for the 8KB direct-mapped cache has to do with references to *persistent-stale* bytes. HD(rc) and HD(invd) do not recover perfectly.

Neither explicitly identifies stale cache lines, ultimately leading to *persistent-stale* data that remains after recovery. From Figure 5-8, the number of *persistent-stale* bytes referenced per 1000 instructions is significantly higher for HD(rc) than for HD(invd). This is to be expected, because the less efficient invalidation method inadvertently invalidates some *stale* data, preventing *persistent-stale* data. This factor, combined with the value prediction enhancement to reduce invalidation's cache miss penalty, pushes HD(invd-vp) up to HD(rc). This is only true for the 8KB direct-mapped cache (also supported by data in Figure 5-8).

HD(inv) significantly underperforms the other recovery models, because of too many compulsory misses after recovery. HD(invd) performs significantly better than HD(inv) because it invalidates fewer cache lines. For a 32KB 4-way set-associative cache, HD(inv) drops slipstream performance improvement from 13% to 8%, whereas HD(invd) only drops it down to 11%.

Using invalidated data as value predictions significantly reduces the impact of recovery-induced misses. HD(inv-vp) performs close to HD(invd) — 10.7% versus 11.0%, respectively, for the 32KB 4-way set-associative cache. And HD(invd-vp) nearly closes the gap between HD(invd) and HD(rc). For all cache configurations, HD(invd-vp) is within a single percentage point of HD(rc). The significant result is that HD(invd-vp) renders the recovery controller obsolete. In fact, all of the invalidation-based models except HD(inv) are effective alternatives to the recovery controller.

As discussed in the previous section, performance improvement of all HD(*) models is sensitive to direct-mapped cache size due to stale data. This trend is observed again in Figure 5-7.

Performance improvement of HD(rc) is relatively insensitive to cache size for the 4-way set-associative caches. Interestingly, the performance improvement of all of the invalidation-based models with respect to BASE decreases as the size of the 4-way set-associative cache is increased. The reason is the BASE processor benefits fully from the increased cache capacity, whereas the invalidation-based models do not benefit fully because lines are invalidated during recovery. And the reason this trend was not visible for direct-mapped caches is the stale problem dominates in that context.

As mentioned earlier, HD(rc) and HD(invd)/HD(invd-vp) are imperfect recovery models. Figure 5-8 shows the *persistent-stale* problem is minor for both models, an order of magnitude smaller than the number of *stale* bytes referenced per 1000 instructions. HD(invd)/HD(invd-vp) have the *persistent-skipped-write* problem as well. The *persistent-skipped-write* problem is also minor, fewer than 0.01 *persistent-skipped-write* bytes referenced per 1000 instructions, for all cache configurations.

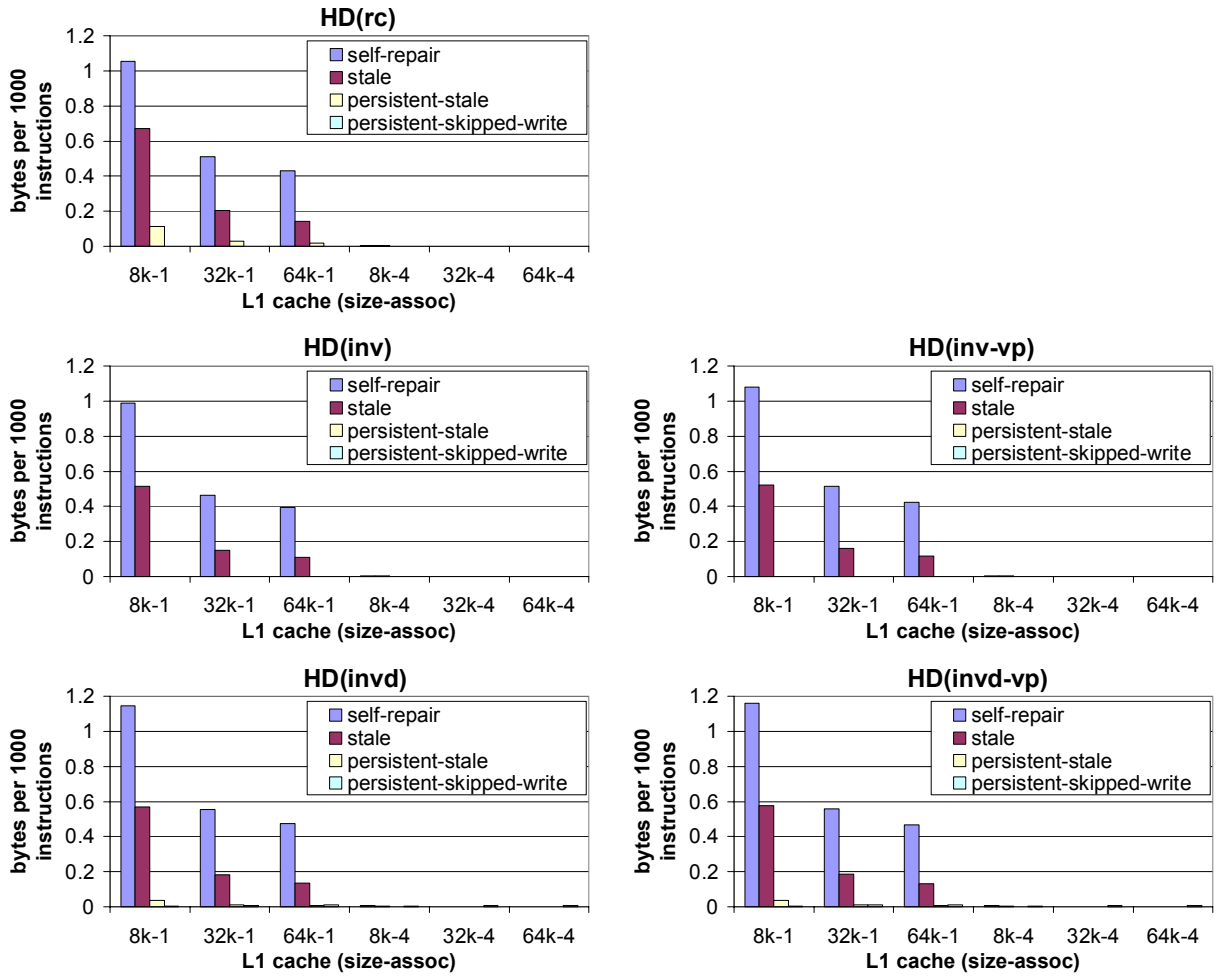

Figure 5-8. Number of referenced *stale*, *self-repair*, *persistent-stale*, and *persistent-skipped-write* bytes per 1000 instructions.

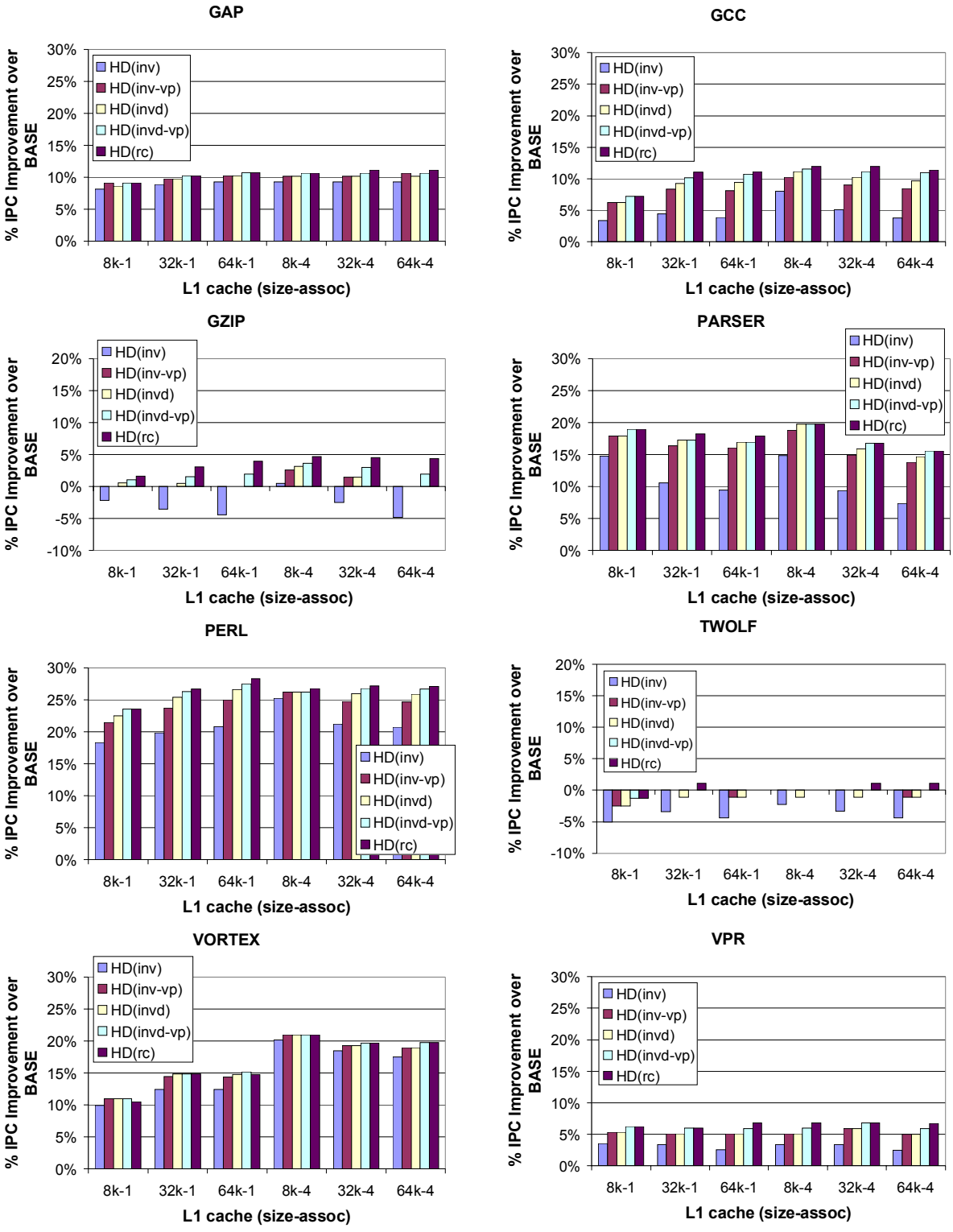

Figure 5-9. Comparison of recovery methods for hardware-based memory duplication, per-benchmark results.

5.7 Summary of memory management for slipstream execution

Through the use of dual redundant execution, slipstream provides a means to use a second processor in a CMP to accelerate a sequential program. Full duplication of physical memory pages in software leads to a simple execution model due to program independence. However, memory usage is doubled, and we showed this partially or fully negates slipstream performance benefits when a realistic memory hierarchy is simulated.

We showed it is possible to duplicate memory efficiently in hardware, without the complications normally associated with managing a fixed amount of rename storage in hardware. Representative CMP hierarchies (private L1 caches that write-through to a shared L2 cache) and the unique nature of slipstream are the sources of simplification. First, the already-replicated L1 caches in a CMP provide enough implicit rename storage. Second, this storage does not need to be explicitly managed because slipstream execution is tolerant of slightly inaccurate memory renaming. Evicted L1 cache lines containing A-stream updates are simply lost, but the R-stream usually reproduces the lost data (which reaches the L2 cache via write-though) before the A-stream re-references the line in the L2 cache. Occasional references to stale data are not a problem because the A-stream is speculative in any case. Moreover, no targeted checking is required because the R-stream already checks the A-stream more generally. In summary, the new hardware-based memory duplication requires no explicit rename storage nor explicit management, and significantly outperforms software-based memory duplication.

Another nice feature of hardware-based duplication is it enables much simpler state recovery. The A-stream can be re-synchronized with the R-stream by invalidating the A-stream L1 cache. We showed compulsory misses after recovery limit performance, and that invalidating dirty lines (while not 100% effective at restoring state) performs much better. And, using preserved data

within invalidated cache lines as value predictions allows the invalidate-dirty-line recovery model to perform within a few percent of the recovery controller, rendering that slipstream component obsolete.

Chapter 6 Simple and Efficient Memory Management for Slipstream Execution on an SMT Substrate

Many of the ideas from the previous chapter, regarding simple and efficient memory management in the context of CMP substrates, also apply to SMT substrates. However, SMT processors typically have a single L1 cache. In this chapter, we develop novel memory management for slipstream execution on single-L1-cache SMT processors. Also, SMT processors typically have a single physical register file, which can be exploited for single-cycle recovery of A-stream registers.

6.1 Quick Recovery of Register Values

For recovery on a CMP substrate, register values must be passed from the R-stream processor to the A-stream processor. This process takes 21 cycles in our experiments (5 cycles to initiate, 4 values/cycle thereafter), making the penalty for IR-mispredictions fairly high. However, on an SMT substrate, values themselves do not need to be copied. Instead, the R-stream's rename map table can be copied to the A-stream's rename map table. This causes the A-stream to initially reference correct values directly from R-stream registers. Support already exists in most superscalar processors for single-cycle copying of map tables (for checkpointing the rename map table at branch instructions and recovering the rename map table when a branch misprediction is resolved). Therefore, we assume it takes only one cycle to recover A-stream register values via this approach.

Although it is safe for the A-stream to initially reference R-stream registers after resolving an IR-misprediction, the A-stream must not be allowed to free R-stream registers. This can be achieved by marking the *previous_mapping* field in the active list as invalid, so that physical registers corresponding to previous mappings inherited from the R-stream are not freed.

Moreover, to prevent potential IR-mispredictions, it is also best if the R-stream does not free one of its registers if the A-stream still needs it. For example, the A-stream may defer freeing a register due to one or more non-modifying writes to the register. A queue can be added before the freelist that merges redundant freeing directives from the A-stream and R-stream, ensuring that both programs indicate it is safe to free a physical register before it is actually returned to the freelist.

6.2 Memory Duplication and Recovery with a Single L1 Cache

To implement hardware-based memory duplication with only a single L1 cache, each L1 cache line is tagged with a thread id indicating whether it belongs to the A-stream or R-stream (indicated by the light and dark shading in Figure 6-1, respectively). As before,

- R-stream stores are performed in both the L1 cache and L2 cache, and
- A-stream stores are performed only in the L1 cache and evicted dirty lines are not written back to the L2 cache.

While this policy achieves the storage efficiency of hardware-based memory duplication for the L2 cache and main memory, there is now pressure on the limited storage capacity of the L1 cache. Once again, We take advantage of the proximity of the A-stream and R-stream to reduce L1 cache pressure.

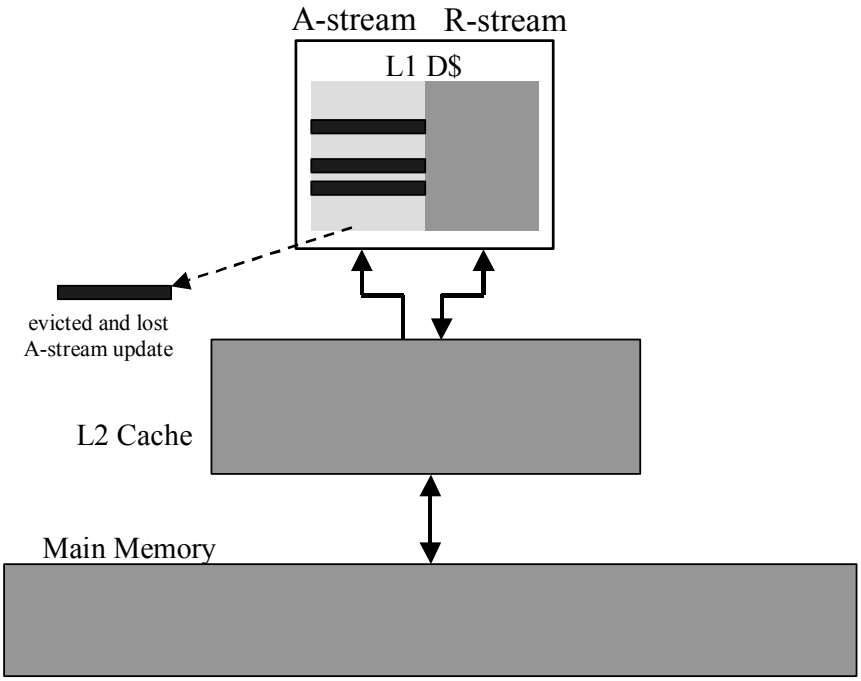

Figure 6-1. Unoptimized hardware-based memory duplication for single L1 cache.

The majority of A-stream and R-stream lines in the L1 cache are redundant, most obviously A-stream lines that have not been stored to (clean lines) are redundant with corresponding R-stream lines. To reduce this redundancy, clean lines are not duplicated and are tagged as R-stream data. When the A-stream performs a load, the cache is searched for both A-stream and R-stream copies of the line, giving preference to the A-stream copy if it exists. A miss (neither the A-stream nor R-stream copy of the line is found) is handled in exactly the same way as an R-stream cache miss – the new line is brought in and tagged as R-stream data. An added benefit of reducing the cache pressure in this way is that an A-stream load that misses in the L1 cache prefetches data for the R-stream.

When the A-stream performs a store and there is not an A-stream copy of the line, a new line is created in the cache and tagged as A-stream data. If the R-stream copy of the line exists in the L1 cache, the new A-stream line is initialized by copying data from the R-stream line, otherwise the data is retrieved from the L2 cache. The A-stream copy of the line is used for all successive

A-stream loads and stores, until it is evicted. This optimized hardware-based memory duplication approach is called *dirty-line duplication*, shown in Figure 6-2.

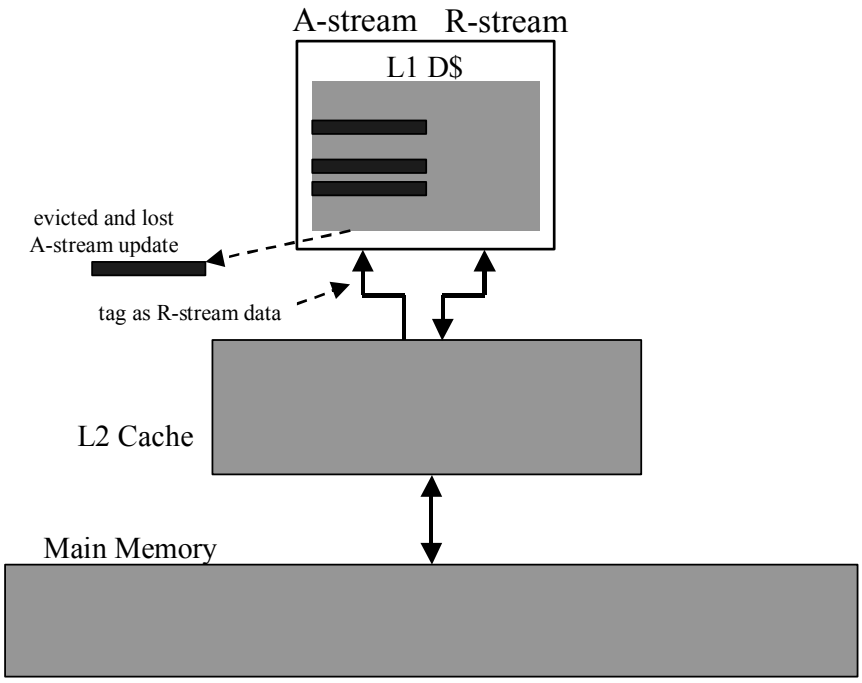

Figure 6-2. Optimized hardware-based memory duplication for single L1 cache: *Dirty-Line Duplication*.

Dirty-line duplication can also be exploited for fast and accurate recovery. Previously, we described two new methods to recover memory state after IR-mispredictions, invalidate and invalidate-dirty, both of which can be augmented with value prediction to reduce the performance impact of compulsory misses after IR-mispredictions. Two points can be made regarding recovery with dirty-line duplication. First, there is no longer a distinction between invalidate and invalidate-dirty, and recovery is total in either case. Second, the A-stream does not have to retrieve correct R-stream data from the L2 cache after recovery, because the data already exists in the L1 cache, resulting in very few compulsory misses after recovery. Fast and accurate recovery of memory state combined with single-cycle recovery of register state yield a very low recovery penalty overall.

Figure 6-3 shows the performance of slipstream execution using two different memory duplication models.

- *basic*: This is unoptimized hardware-based memory duplication, i.e., clean and dirty lines are duplicated in the L1 cache.
- *optimized*: This is optimized hardware-based memory duplication, i.e., dirty-line duplication.

The performance metric is IPC improvement of slipstream execution on an 8-issue SMT processor with respect to conventional non-redundant execution on the same 8-issue SMT processor. We consider 2-way and 4-way set-associative caches of sizes 16, 32, and 64 KB. For both the *basic* and *optimized* caches, all A-stream lines are invalidated when an IR-misprediction is resolved. However, the *basic* cache must subsequently retrieve lines from the L2 cache whereas the *optimized* cache can use R-stream lines in the L1 cache directly.

In every case, the *optimized* cache outperforms the *basic* cache by large margins, due to (1) reduced L1 cache pressure and (2) faster recovery. Most notably, for *perl* and a 32 KB 2-way set-associative cache, a 27% slowdown is converted into a 5% speedup. Except for *gap* and *parser*, the *basic* cache significantly degrades performance. The *optimized* cache performs well for *gap*, *gcc, parser*, and *perl*, and breaks even for other benchmarks except *twolf*, which suffers a small degradation.

Generally speaking, slipstream execution on an SMT substrate performs better with larger caches and higher set-associativity. Larger caches and higher set-associativity accommodate the additional A-stream memory more easily.

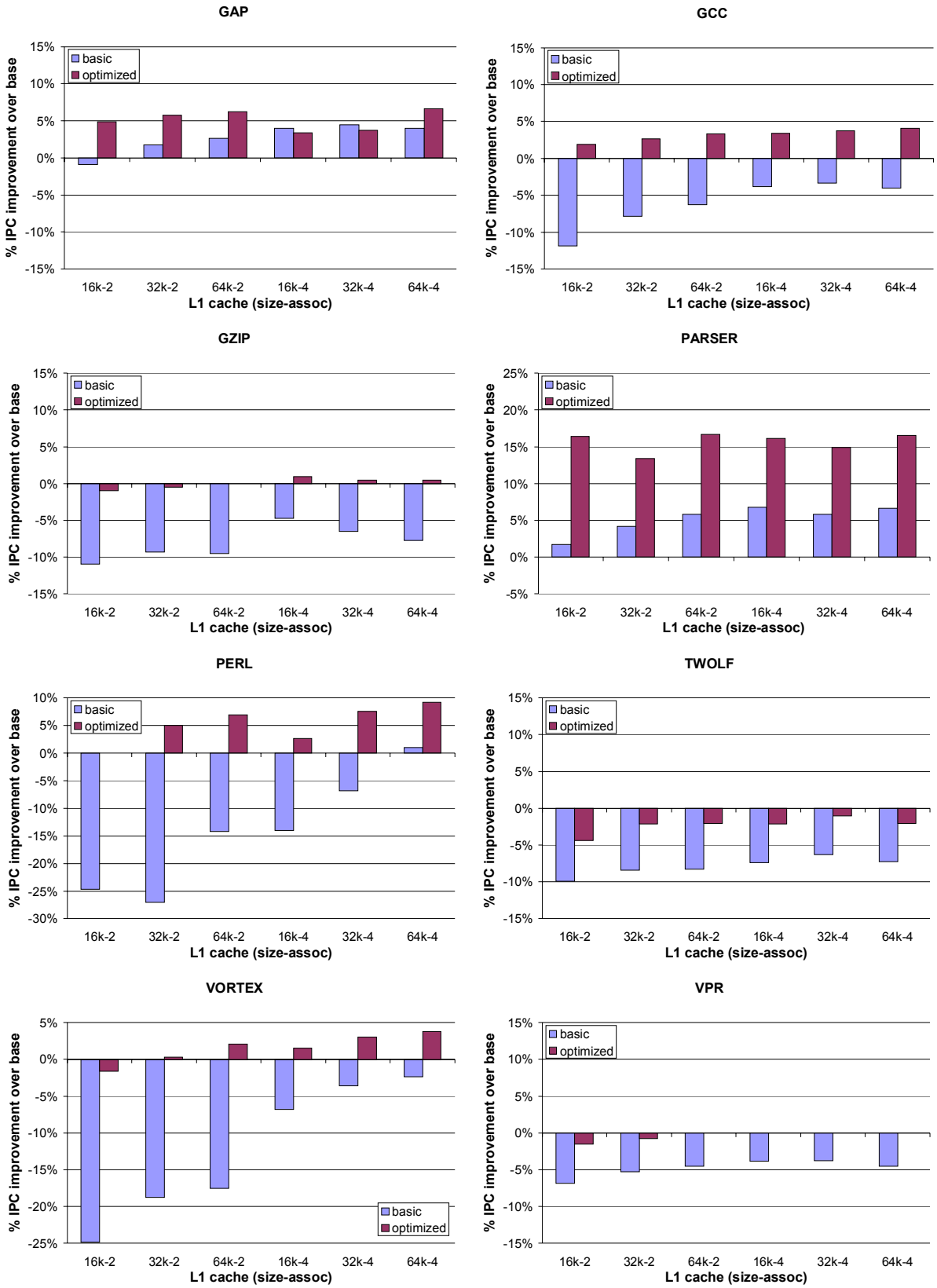

Figure 6-3. Speedup of slipstream execution on SMT substrate with two dup. models.

Figure 6-4 shows the IPC improvement of slipstream execution with respect to conventional execution on an 8-issue SMT processor, using a 64 KB 4-way set-associative cache with dirty-line duplication. Three different confidence thresholds are used: the default threshold of 32 (used for all previous simulations) and two lower thresholds of 24 and 16. Lowering the confidence threshold allows for more aggressive instruction removal but results in many more IR-mispredictions. The relatively flat IPC improvement with decreasing confidence threshold suggests that the quick recovery capability of the SMT-based slipstream processor allows it to tolerate more IR-mispredictions.

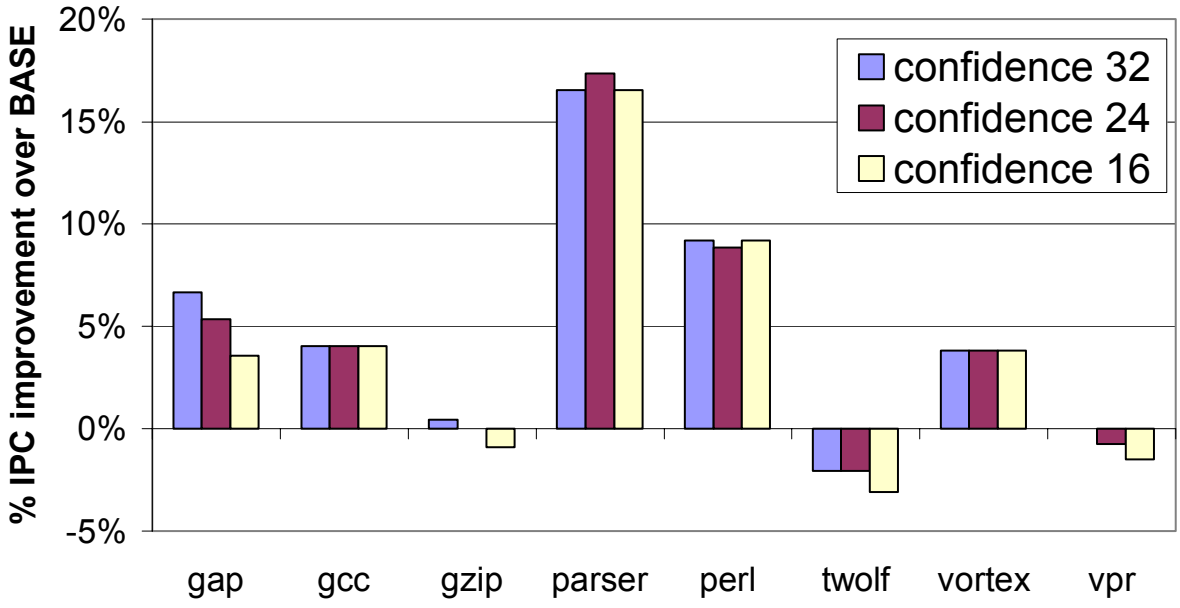

Figure 6-4. SMT-based slipstream processor with dirty-line duplication is tolerant of IR-mispredictions due to quick recovery.

Chapter 7 Managing Slipstream Execution Mode on a CMP

Slipstream execution does not speed up all applications. Moreover, within an application, there may be periods for which slipstream execution is beneficial and other periods for which it is not. In this chapter, we explore simple hardware support for (1) predicting the effectiveness of slipstream execution across and within applications and (2) dynamically enabling and disabling slipstream execution mode. This kind of support may enable an operating system (O/S) to choose between slipstream execution mode and conventional modes of execution. The O/S can base its policy decisions on slipstream performance predictions, job requirements, and constraints such as power consumption. Several scenarios are described below in terms of job requirements.

- *One foreground job* (common in PC environment [14]): Slipstream is enabled if it is likely to enhance performance. Otherwise, it is disabled to conserve power.
- *Multiple foreground jobs* (e.g., multithreaded implementation of MPEG player or parallel make [14]) or *multiple high-priority batch jobs* (servers): Even if slipstream is predicted to do well on one or more of the jobs, slipstream may be disabled in favor of job throughput. However, if two different jobs compete for memory resources, then enabling slipstream and context-switching between the jobs may perform better than executing the jobs simultaneously.

We propose using the percentage of predicted-ineffectual instructions as a key indicator of whether or not slipstream should be enabled. The key idea is that the IR-detector and IR-predictor can be run continuously, *whether or not slipstream execution mode is enabled*. (That is, the IR-detector and IR-predictor can operate with or without an A-stream.) These components will be used to measure the percentage of predicted-ineffectual instructions over an interval. Slipstream will be enabled during the next interval if the percentage of predicted-ineffectual instructions exceeds a threshold during the previous interval.

7.1 Preliminary analysis

We did preliminary analysis by dividing the retired dynamic instruction stream into 4K-instruction intervals, for slipstream execution on two cores (the R-stream is used) and conventional execution on a single core. For the slipstream processor, we recorded IPC and the percentage of predicted-ineffectual instructions for each interval. For the superscalar processor, we recorded IPC for each interval. Using this information, we can deduce overall speedup if slipstream is enabled only during intervals that exceed a certain threshold of predicted-ineffectual instructions. Speedup as a function of threshold is shown in Figure 7-1 for bzip, gap, gcc, and jpeg, and in Figure 7-2 for parser, perl, and vortex. This method is somewhat ideal because slipstream is enabled/disabled during an interval based on that interval's ineffectual percentage. It is labeled "use_current_interval" in the graphs. A more realistic approach is to enable slipstream during an interval if the previous interval exceeds a certain threshold of predicted-ineffectual instructions. This method is labeled "use_previous_interval" in the graphs. To the right of each speedup graph is another graph, indicating the percentage of intervals for which slipstream is enabled, as a function of threshold.

For gcc, overall speedup is 12.6% if slipstream is always enabled. Setting the threshold to 50% predicted-ineffectual instructions enables slipstream for 65% of the intervals for an overall speedup of 10%. This means the second core can be used by another job 1/3 of the time while still extracting most of the benefits of slipstream for gcc.

For vortex and bzip, more than 98% of the intervals have over 50% predicted-ineffectual instructions. This results in slipstream being enabled almost all of the time even with a threshold as high as 50% and almost no change in speedup as the threshold increases from 0% to 50%. For vortex, these are desirable results since slipstream provides significant speedup (20%). However, bzip speedup is only moderate (6.5%) in spite of significant A-stream reduction. The reason is the base IPC of bzip is 3.7. For a 4-issue superscalar core, the maximum possible speedup is only 8%. The conclusion is that base IPC should also be considered before enabling slipstream. Base IPC can be sampled by occasionally disabling slipstream. If base IPC is close to peak IPC, slipstream is disabled until base IPC drops and the predicted-ineffectual threshold is exceeded. In the case of bzip, it is a close call regarding whether or not to enable slipstream because 8% speedup potential is respectable.

Parser and perl are similar to bzip and vortex, except there is a significant drop-off in speedup for thresholds above 40%. However, both parser and perl still achieve significant speedup with a 50% threshold. A 50% threshold enables slipstream 45% (perl) to 60% (parser) of the time. The second core can be used for other jobs during intervals that slipstream is disabled, at the price of decreasing slipstream speedup by 1/3 (parser) to 1/2 (perl).

For gap, speedup decreases more or less linearly with increasing threshold. For example, setting the threshold at 20% causes slipstream to be enabled about 50% of the time for an overall

speedup of 6.5%, just over half the speedup with slipstream always enabled. For gap, slipstream should be enabled most of the time otherwise speedup declines significantly.

Finally, for jpeg, slipstream should always be disabled due to marginal speedup (2%). Using a threshold to disable slipstream yields mixed results for jpeg. A 20% threshold disables slipstream about 2/3 of the time. However, a 50% threshold fails to disable jpeg entirely and also causes an overall slowdown. The slowdown stems from too many A-stream deviations. This can be mitigated by also considering potential A-stream deviations. The slipstream components can measure potential A-stream deviations. Predicted-ineffectual instructions (from the IR-predictor) and past-ineffectual instructions (from the IR-detector) can be cross-checked to detect potential *instruction-removal mispredictions* (IR-mispredictions).

For all benchmarks except gap, using measurements from the previous interval ("use_previous_interval") to enable/disable slipstream in the next interval is as effective as using measurements from the current interval ("use_current_interval").

Preliminary observations are summarized below. Although only a very minimal implementation is presented in the next section, the observations below may guide future research in this area.

- Existing slipstream components can provide continuous feedback regarding the potential effectiveness of slipstream, whether or not slipstream execution mode is enabled.
- Setting a threshold for the percentage of predicted-ineffectual instructions is an effective means for controlling the use of slipstream execution mode.
- The percentage of predicted-ineffectual instructions in the previous interval is a remarkably good indicator of whether or not slipstream should be enabled in the next interval.

- A single threshold is not evident from preliminary experiments. A 50% threshold does not diminish performance for bzip and vortex, and only moderately impacts performance for gcc while freeing up the second core 1/3 of the time. A 40% threshold is preferred for parser and perl, although a 50% threshold strikes a good balance between slipstream speedup and freeing the second core for other jobs. A 10% or lower threshold is preferred for gap, although a 20% threshold also strikes a good balance between slipstream speedup and job throughput. A 50% threshold manages to disable jpeg 90% of the time, as desired, but A-stream deviations cause a minor slowdown.
- A 50% threshold appears to give desirable results on the whole, but a single threshold is non-optimal on a case-by-case basis. Other indicators are clearly required, including slack between the A-stream and R-stream, IR-mispredictions, and base IPC. (1) *Slack*. Gap achieves significant speedup with an atypically low percentage of instruction removal. This is why a 10-20% threshold is preferred for gap instead of 50%. Clearly, quality of instruction removal is equally important as quantity. One possible approach is to begin with a high threshold (50%) and decrease the threshold until there is a positive impact on A-stream/R-stream slack. Slack reflects both the quality and quantity of instruction removal. (2) *IR-mispredictions*. A 50% threshold is preferred for jpeg in terms of minimizing slipstream usage, but results in an overall slowdown due to IR-mispredictions. Therefore, the potential for a high IR-misprediction rate should override the decision to enable slipstream. (3) *Base IPC*. Bzip has a high base IPC so the speedup potential is limited even though there is significant instruction removal in this benchmark. As discussed earlier, slipstream can be occasionally disabled to check the base IPC and only re-enabled if there is room for improvement.

- Interval size may be a crucial parameter. An interval that is too small results in large overheads for enabling and disabling slipstream. Activating the A-stream is the same as recovering after an IR-misprediction. Thus, the lower bound on interval size should be based on information regarding acceptable IR-misprediction rates. An interval that is too large results in less timely enabling/disabling of slipstream.

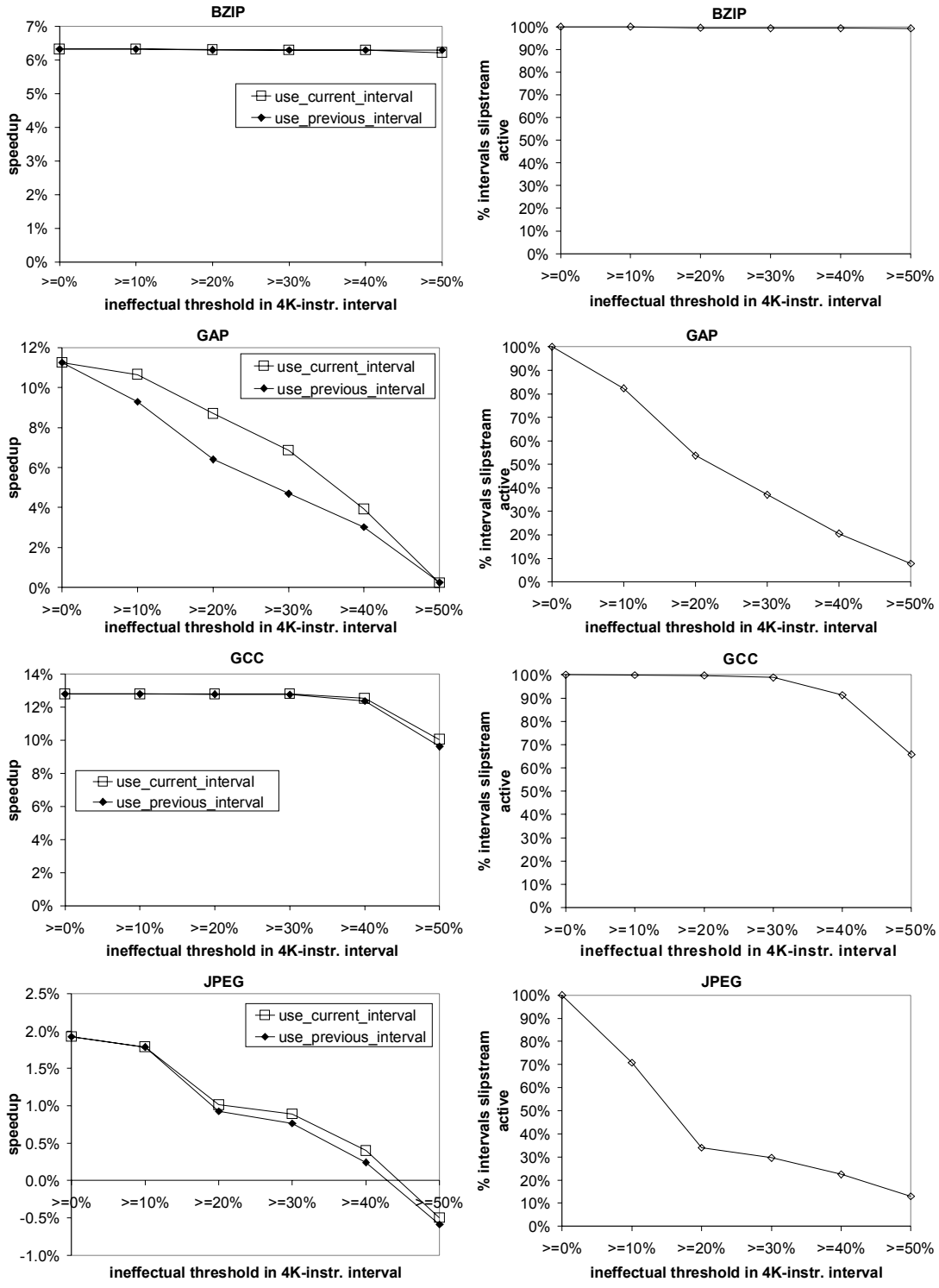

Figure 7-1. ***Speedup*** **(left) and** ***percentage of intervals that slipstream is active*** **(right) as a function of predicted-ineffectual threshold. (*bzip*, *gap*, *gcc*, *jpeg*)**

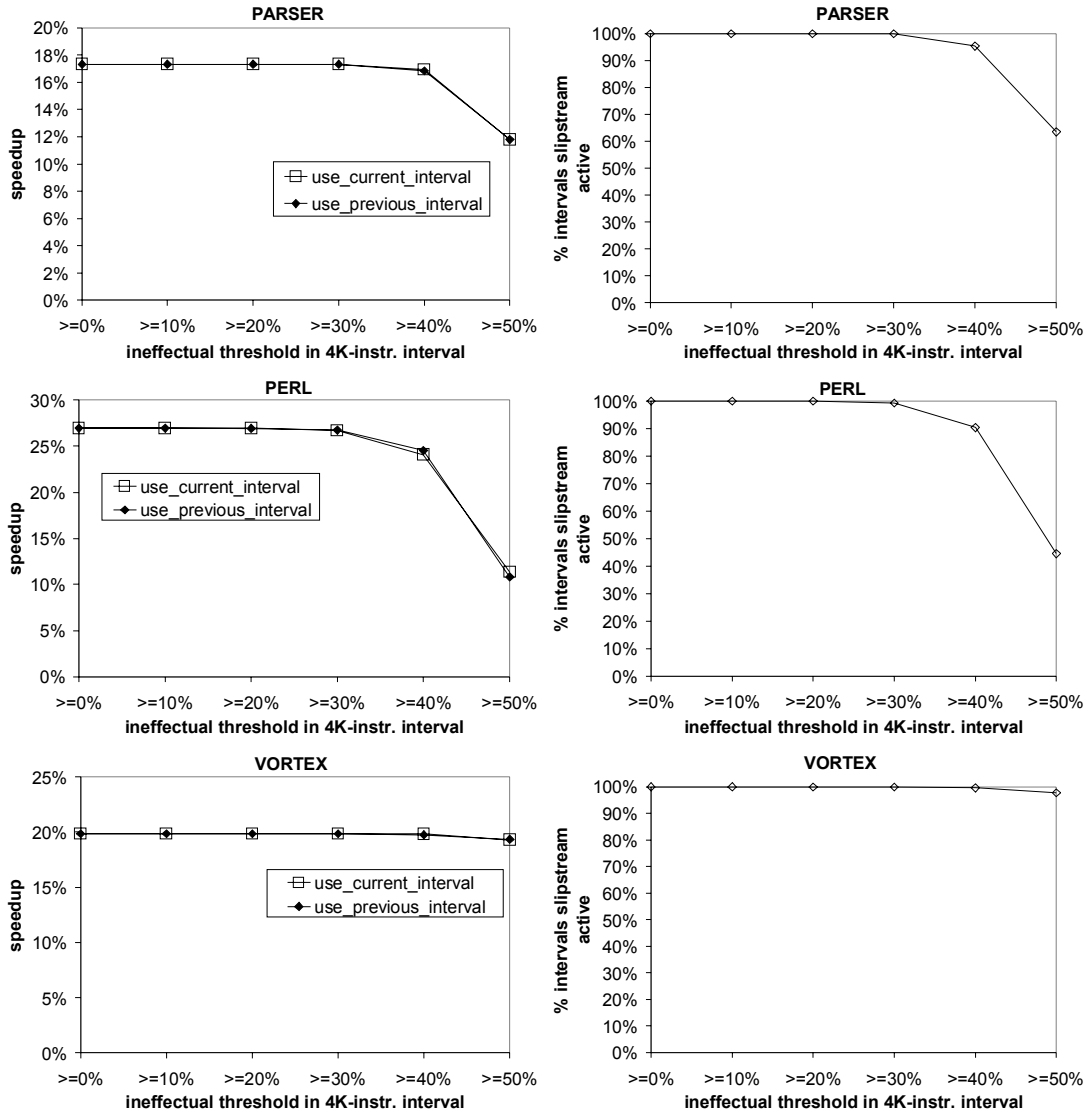

Figure 7-2. ***Speedup*** **(left) and** ***percentage of intervals that slipstream is active*** **(right) as a function of predicted-ineffectual threshold. (*parser*, *perl*, *vortex*)**

7.2 Implementation

7.2.1 Enabling slipstream execution mode

The O/S is involved in enabling slipstream execution mode only so far as ensuring that an idle processor is available. The slipstream management unit does not enable slipstream if a spare processor is not available.

The mechanisms for enabling slipstream execution mode are already in place, since initiating the A-stream is the same as repairing the A-stream after an IR-misprediction. To initiate the A-stream, an IR-misprediction is forced in the single program copy that is currently executing (which becomes the R-stream). This causes the reorder buffer of the R-stream to be squashed, the IR-predictor to be backed up to the precise program counter, and the A-stream reorder buffer to be initialized (head and tail pointers set equal). Then, the R-stream register file is copied to the A-stream register file and the A-stream L1 cache is invalidated. Note that the delay buffer should already be empty at this point, although it is safe to explicitly squash it. In summary, the procedure for initiating the A-stream is identical to IR-misprediction recovery.

7.2.2 Disabling slipstream execution mode

Slipstream execution can be disabled at any time by terminating A-stream instruction fetching and draining residual control-flow and data-flow outcomes from the delay buffer. The IR-predictor is suspended from generating new predictions until the delay buffer is drained by the R-stream. After the delay buffer is drained, the IR-predictor continues making predictions from where it left off, only now its branch predictions are supplied to the R-stream directly (instruction removal predictions are ignored). The IR-predictor and IR-detector continue to run as in normal slipstream execution mode, so that predicted-ineffectual percentages can be

measured continuously and used to guide re-enabling slipstream execution mode at some later time.

7.2.3 Slipstream management unit

The slipstream management unit maintains two counters. The first counter is incremented by one for each retired R-stream instruction. The second counter is incremented by one for each retired R-stream instruction whose R-bit is set, indicating a predicted-ineffectual instruction. When the first counter reaches the interval size (4K instructions in the experiments that follow), a decision is made to enable/disable slipstream execution mode based on the magnitude of the second counter. If the number of predicted-ineffectual instructions in the past interval exceeds a certain threshold, slipstream is enabled if it is currently disabled. If the threshold is not exceeded, then slipstream is disabled if it is currently enabled. After making a decision, both counters are reset in order to take a new measurement for the next interval.

7.2.4 Results

Figure 7-3 shows IPC improvement of slipstream execution on two processors of a CMP with respect to conventional non-redundant execution on only one of the processors. The first bar (“regular”) shows IPC improvement when slipstream execution mode is always enabled. The second bar (“duet”) shows IPC improvement when slipstream execution mode is dynamically managed. For duet, slipstream execution mode is enabled during a 4K-instruction interval if the predicted-ineffectual percentage in the previous interval is 30% or higher. Figure 7-4 shows the percentage of 4K-instruction intervals that slipstream execution mode is enabled, for the duet model.

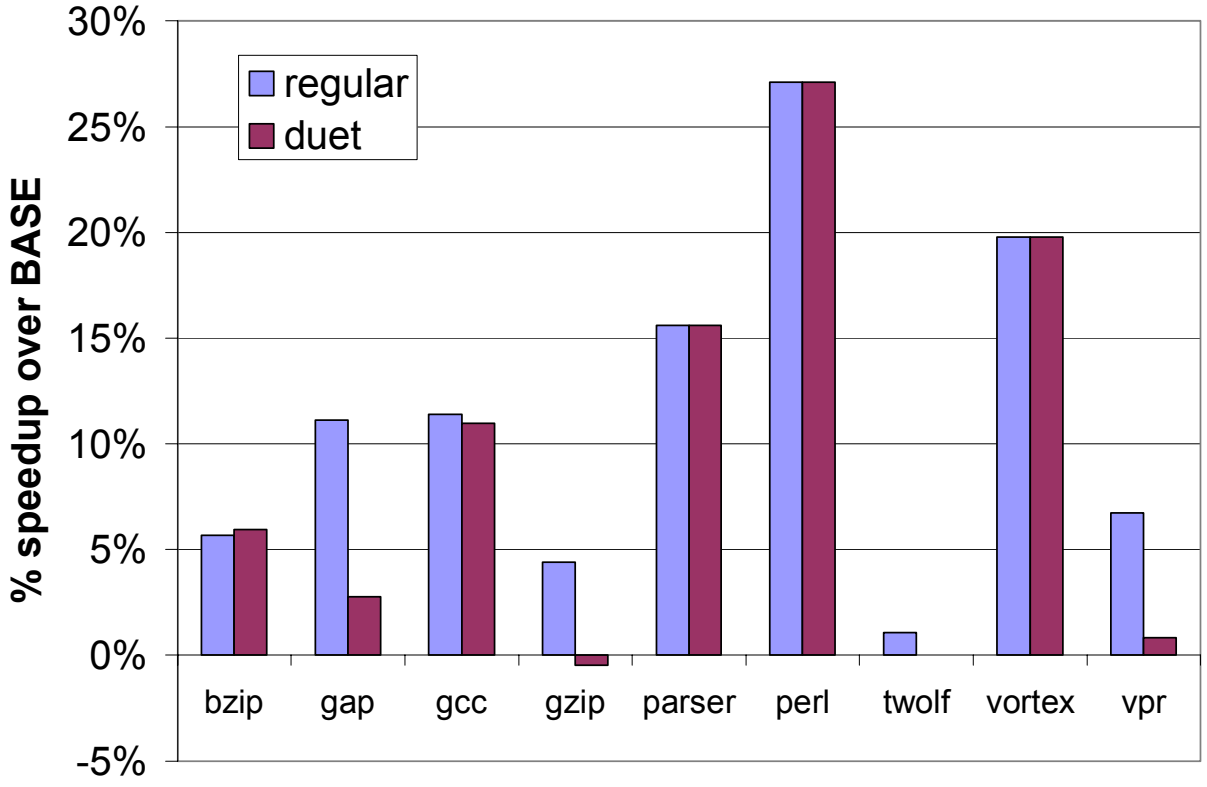

Figure 7-3. Slipstream performance with and without management of slipstream execution mode.

Based on the preliminary results in Section 7.1, we would expect bzip, gcc, parser, perl, and vortex to be enabled for almost 100% of the intervals and for duet performance to match regular performance. These observations are confirmed by the results in Figure 7-3 and Figure 7-4. The preliminary results projected that gap should be enabled for about 35% of the intervals for a speedup of about 4%. The results in this section are close to the projections: gap is actually enabled for about 32% of the intervals for a speedup of nearly 3%.

Thus, an implementable slipstream management unit produces results that are remarkably close to projections, even with the overheads for enabling and disabling slipstream execution mode accounted for. Using a 4K-instruction interval and a 30% threshold correctly distinguishes between programs for which slipstream execution mode is beneficial and not beneficial.

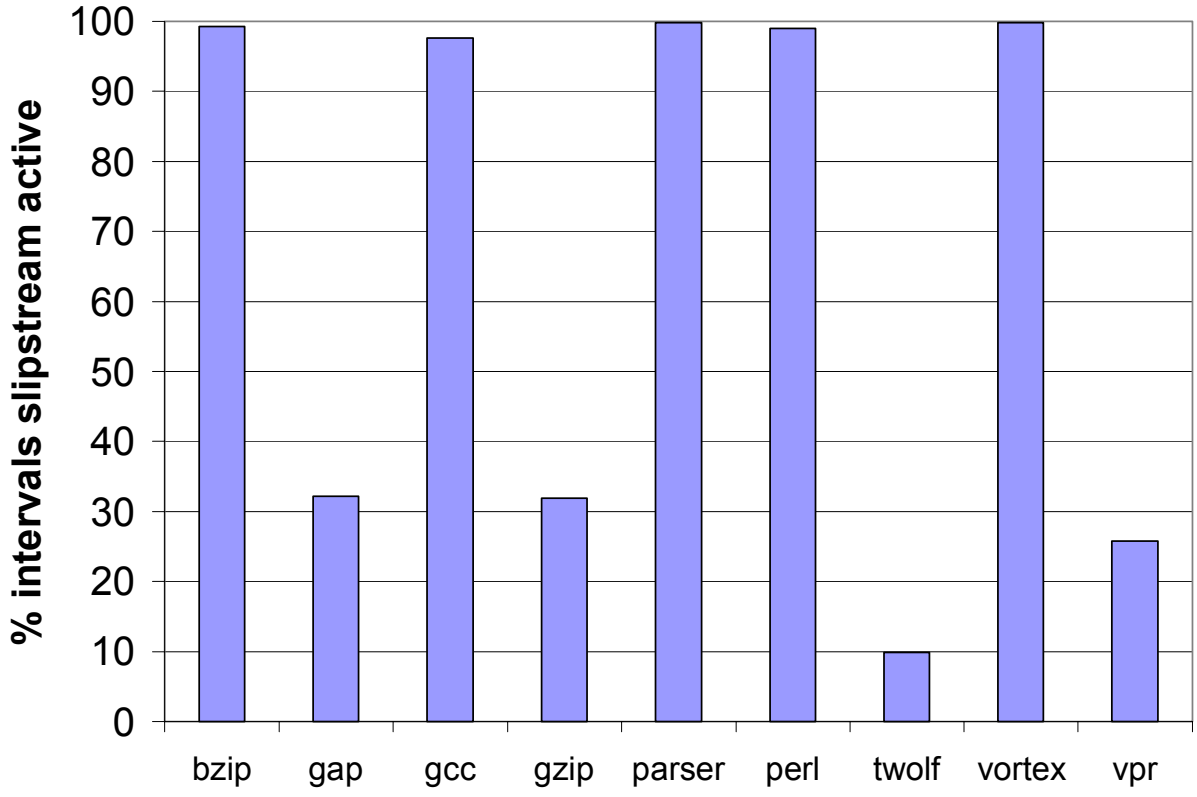

Figure 7-4. Percentage of intervals that slipstream execution mode is enabled.

Chapter 8 Summary and Future Work

One of the chief trends in high-performance microprocessors is hardware support for simultaneous execution of multiple independent programs. This trend is no longer academic. For example, the IBM POWER4 is a dual-processor CMP and the Intel Pentium4 is a dual-context SMT processor. The reason for this trend is that integrating multiple contexts on a single chip provides high pay-off with only evolutionary extensions. The CMP and SMT paradigms effectively and efficiently leverage 100 million to 1 billion transistors on a chip.

The broad rationale for slipstream processors is extending, not replacing, the capabilities of CMP/SMT processors. This thesis demonstrates that the addition of three components (the IR-predictor, IR-detector, and delay buffer), interfaced with conventional pathways in a contemporary pipeline, can facilitate slipstream execution, accelerating conventional sequential programs and at the same time transparently enhancing their resistance to single-event upsets. Specific milestones in the development of an effective and feasible slipstream microarchitecture include (1) a means for shortening the leading program, that includes such innovations as managing removal-confidence counters individually while synchronizing counters of producer/consumer pairs, (2) an efficient means for duplicating memory, that does not require any explicit software or hardware memory management, simplifies recovery, and facilitates maximum slipstream performance without increasing memory system capacity, and (3) a means for predicting the effectiveness of slipstream execution across and within applications and

dynamically enabling/disabling slipstream execution mode accordingly, thereby empowering the operating system to flexibly manage execution modes of the processor.

The contributions of this thesis have significantly advanced the slipstream paradigm and microarchitecture, yet many interesting areas remain for future research, some of which are highlighted below.

- *Efficient IR-predictor.* The IR-predictor is the one slipstream component that has not been engineered for practical implementation yet. Koppanalil implemented a highly efficient IR-detector [22] and Purser implemented efficient memory duplication and recovery [37]. The main concern with the current IR-predictor is its impractical size. Preliminary experiments indicate that a single confidence counter per entry, time-shared among instructions in the fetch block, is sufficient. Preliminary experiments also indicate that only a small fraction of all dynamic fetch blocks contribute most of the instruction removal in a program. A table of 2-bit or 3-bit confidence counters can filter out fetch blocks that are likely to make large contributions, and only these fetch blocks are assigned semi-permanent entries in a small cache of tagged removal-confidence counters. Preliminary experiments with the simplest design of all – associating a confidence counter with each instruction in the instruction cache – yield instruction removal rates of 30% or higher on benchmarks that normally achieve 40% removal with unbounded predictors. These latter experiments suggest that deep global branch history is not so much needed for the confidence counters themselves (as previously thought), as for maximizing the prediction accuracy and hence removability of branches.
- *Slipstream fetch unit designs and alternative A-stream representations*. This thesis proposed ways to interface the IR-predictor to the A-stream fetch unit, in particular with respect to

bypassing instruction fetch and collapsing predicted-ineffectual instructions within a fetch block. However, specific designs are needed, including BTB support for bypassing instruction fetch, etc. Alternative A-stream representations (e.g., compressed program images), derived from a compiler, binary translator, or dynamic optimization framework, could be cached in the instruction cache or trace cache of the A-stream processor to assist program sequencing.

- *Interfacing slipstream components with pipeline*. Specific pipeline designs are needed with respect to injecting A-stream branch and value predictions, verifying predictions, etc.
- *Implementable slipstream processor*. Now is the time to reflect on what has been learned about slipstream processors and which factors affect performance the most, and design a slipstream processor from the ground-up based on accumulated experiences. Ideally, the end result of this exercise is a lean slipstream processor design that is complete, effective, and implementable.
- *Improving slipstream performance*. Further research is needed to affect benchmarks that do not currently benefit from slipstream and increase speedups for benchmarks that do. In particular, the A-stream needs to be reduced more, either through additional ineffectual criteria, on-the-fly removal of long-latency operations that stall A-stream retirement, or both. The slipstream execution model may also provide a framework for efficiently exploiting control independence [42].
- *Slipstream fault tolerance*. This thesis only briefly touched upon the intrinsic fault tolerance of slipstream processors. A more rigorous study of reliability is needed, taking into consideration partial redundancy, pipeline coverage, etc.

- *Slipstream on deep pipelines*. This thesis did not evaluate slipstream performance in the context of a deep pipeline, a strategic design point.
- *Deploying slipstream management in full-system simulation*. This thesis provides a slipstream management unit for fluidly enabling/disabling slipstream execution mode. An interesting area for future research is deploying fluid slipstream management in full-system simulation that includes the O/S, multiprogramming, priorities, and power/reliability goals. O/S policies need to be developed that leverage and work with the slipstream management unit. These policies manage processors and/or thread contexts under multiple constraints: single-program performance, job throughput, reliability, and power.

References

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

[2] M. Annavaram, J. Patel, and E. Davidson. Data Prefetching by Dependence Graph Precomputation. *28th Int'l Symp. on Computer Architecture*, July 2001.

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

[4] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically Allocating Processor Resources Between Nearby and Distant ILP. *28th Int'l Symp. on Computer Architecture*, July 2001.

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

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

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

[8] S. Chatterjee, C. Weaver, and T. Austin. Efficient Checker Processor Design. *33rd Int'l Symp. on Microarchitecture*, Dec. 2000.

[9] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. *28th Int'l Symp. on Computer Architecture*, July 2001.

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

[11] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss. *Proceedings of ICS*, 1997.

[12] J. Emer. Simultaneous Multithreading: Multiplying Alpha Performance. *Microprocessor Forum*, October 1999.

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

[14] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread Level Parallelism of Desktop Applications. *Workshop on Multi-threaded Execution, Architecture, and Compilation (MTEAC2000)*, Jan. 2000.

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

[16] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. *4th Int'l Symposium on High-Performance Computer Architecture*, February 1998.

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

[18] K. Ibrahim, G. Byrd, and E. Rotenberg. Slipstream Execution Mode for CMP-Based Multiprocessors. *9th Int'l Symp. on High-Performance Computer Architecture*, Feb. 2003.

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

[20] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *17th Int'l Symp. on Computer Architecture*, May 1990.

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

[22] J. J. Koppanalil. A Simple Mechanism for Detecting Ineffectual Instructions in Slipstream Processors. M.S. Thesis, North Carolina State University, May 2002.

[23] J. J. Koppanalil and E. Rotenberg. A Simple Mechanism for Detecting Ineffectual Instructions in Slipstream Processors. *Submitted to IEEE Transactions on Computers (Jan. 2003)*.

[24] V. Krishnan and J. Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor. *1998 Int'l Conference on Supercomputing*, July 1998.

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

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

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

[28] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. *28th Int'l Symp. on Computer Architecture*, July 2001.

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

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

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

[32] A. Moshovos and G. S. Sohi. Streamlining Inter-Operation Memory Communication via Data Dependence Prediction. *30th Int'l Symp. on Microarchitecture*, Dec. 1997.

[33] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice Processors: An Implementation of Operation-Based Prediction. *15th Int'l Conf. on Supercomputing*, June 2001.

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

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

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

[37] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. Slipstream Memory Hierarchies. Technical Report, North Carolina State University, Feb. 2002.

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

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

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

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

[42] E. Rotenberg. Trace Processors: Exploiting Hierarchy and Speculation. Ph.D. Thesis, University of Wisconsin – Madison, Aug. 1999.

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

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

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

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

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

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

[49] Y. H. Song and M. Dubois. Assisted Execution. Technical Report CENG-98-25, Department of EE-Systems, University of Southern California, October 1998.

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

[51] S. Storino and J. Borkenhagen. A Multi-Threaded 64-bit PowerPC Commercial RISC Processor Design. *11th Hot Chips Symposium*, August 1999.

[52] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. *9th Int'l Conf. on Arch. Support for Programming Languages and Operating Systems*, Nov. 2000.

[53] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1):5-25, Jan. 2002.

[54] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. *11th Hot Chips Symposium*, Aug. 1999.

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

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

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

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

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

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

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

[62] C. Zilles and G. Sohi. Execution-based Prediction Using Speculative Slices. *28th Int'l Symp. on Computer Architecture*, July 2001.