ABSTRACT

AL-ZAWAWI, AHMED SAMI. Transparent Control Independence (TCI). (Under the direction of Dr. Eric Rotenberg).

Superscalar architectures have been proposed that exploit control independence, reducing the performance penalty of branch mispredictions by preserving the work of future misprediction-independent instructions. The essential goal of exploiting control independence is to completely decouple future misprediction-independent instructions from deferred misprediction-dependent instructions. Current implementations fall short of this goal because they explicitly maintain program order among misprediction-independent and misprediction-dependent instructions. Explicit approaches sacrifice design efficiency and ultimately performance.

We observe it is sufficient to emulate program order. Potential misprediction-dependent instructions are singled out *a priori* and their unchanging source values are checkpointed. These instructions and values are set aside as a “recovery program”. Checkpointed source values break the data dependencies with co-mingled misprediction-independent instructions – now long since gone from the pipeline – achieving the essential decoupling objective. When the mispredicted branch resolves, recovery is achieved by fetching the self-sufficient, condensed recovery program. Recovery is effectively transparent to the pipeline, in that speculative state is not rolled back and recovery appears as a jump to code. A coarse-grain retirement substrate permits the relaxed order between the decoupled programs. Transparent control independence (TCI) yields a highly streamlined pipeline that quickly recycles resources based on conventional speculation, enabling a large window with small cycle-critical resources, and prevents many mispredictions from disrupting this large window.

TCI achieves speedups as high as 64% (16% average) and 88% (22% average) for 4-issue and 8-issue pipelines, respectively, among 15 SPEC integer benchmarks. Factors that limit the performance of explicitly ordered approaches are quantified.

TRANSPARENT CONTROL INDEPENDENCE (TCI)

by

AHMED SAMI AL-ZAWAWI

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, North Carolina

2007

APPROVED BY:

Dr. Eric Rotenberg, Chair of Advisory Committee

Dr. Thomas M. Conte

Dr. Suleyman Sair

Dr. Warren J. Jasper

DEDICATION

To my son, Hamza

To whom I wish a very bright future…

BIOGRAPHY

Ahmed S. Al-Zawawi was born in 1979 in Riyadh, Saudi Arabia. In 1998, he received a full scholarship to continue his higher education in the United States of America. By the end of 2001, Ahmed graduated summa cum laude with a Bachelor of Science degree in Computer Engineering and a minor in Mathematics from North Carolina State University. Ahmed pursued his graduate studies in Computer Engineering at NCSU under the guidance of Dr. Eric Rotenberg. He received the Masters degree in 2002 and the Doctor of Philosophy degree in 2007. During his graduate career, Ahmed joined Intel's prestigious microarchitecture research lab (MRL) to complete his research internship. His research interests include computer architecture, high-performance microarchitecture, and compiler optimization.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Allah (God), the Most Gracious, the Most Merciful for his help and guidance in completing this work.

Next, I must thank my family, especially my parents and my wife. My parents, Zain and Sami, always gave me all the support and encouragement for education that underlies any success I have experienced. Without their unconditional love and confidence in me, I would not be at this stage today. Their strong belief in my abilities was always pushing me forward throughout my life and my graduate career in particular. Whatever I say, I will not be able to thank them enough.

My wife, Najwa, has always been my main source of support. In addition to her being primarily responsible for raising our son, it would have been difficult for me to achieve any success without her understanding, assistance and patience. She has graciously accepted the fact that for three years we would not be able to take a visit to our home country. My debt of gratitude to her cannot be fully paid.

I would like to express my heartfelt gratitude to my advisor Dr. Eric Rotenberg for accepting me as his student. I really respect and appreciate Eric for all the knowledge I learned from him. I would like to thank him for patiently answering all my questions and for his invaluable advice and comments that inspired me in defining my research. Eric was not only my advisor; he was also a big brother for me and a role-model for a respectful professor. He taught me how the relationship between a student and a professor should be and how concerned and devoted a professor must be. I am really indebted to him for all his help and teaching. And I am also grateful to my defense committee members (Dr. Thomas M. Conte, Dr. Suleyman Sair, and Dr. Warren J. Jasper) for their help in improving this thesis.

My sincere thanks also go to so many friends and colleagues in the CESR group. I want to thank Vimal, Ali, Aravindh, Ravi, Muawya, Hashem, Saurabh, Chad, Paul, and Balaji for being good friends and for providing me with valuable feedback whenever I needed it. My time with you guys in the office and our conversations together will never be forgotten.

I am very thankful to my friends who made the past five years possible. I must thank my best friends Hatem, Abdul-Satir, Hani, Joud and Wail for always being there whenever I needed them in time of happiness and sadness. They were my family in Raleigh and I could always count on them. Our gatherings and arguments will always bring to me a nice memory of great people I was fortunate to be surrounded by during my stay in the U.S.

Finally, I would like to thank my little son Hamza who always put a smile on my face in spite of all the stress and tension I got through during my graduate career. His presence was a landmark in my life and looking at him is always encouraging me to be the best I can be.

I would like to express my sincere thanks to the Kingdom of Saudi Arabia for funding me throughout my graduate studies and for their continued support through the Saudi Arabian Cultural Mission. This project was also supported by NSF CAREER grant No. CCF-0092832, NSF grant No. CCF-0429843, and Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ILP – Instruction-Level Parallelism

IPC – Instructions Per Cycle

CI – Control-Independent

CD – Control-Dependent

CIDD – Control-Independent Data-Dependent

CIDI – Control-Independent Data-Independent

CFS – Control-Flow Stack

TCI – Transparent Control Impendence

RXB – Re-Execution Buffer

RP – Reconvergent Point

Chapter 1

Introduction

The performance of microprocessors has shown remarkable improvement in the past two decades. This improvement can be attributed to two factors: faster transistors, through technology advancements, and higher levels of instruction-level parallelism (ILP), through microarchitecture advancements. Technology trends remain strong for the foreseeable future. However, continuing to increase ILP is jeopardized by control-flow limits and ever-increasing memory latency. This dissertation is concerned with control-flow limits.

Modern superscalar processors extract ILP from a reservoir of instructions called the *instruction window*. The larger the instruction window, the more likely the processor can find independent instructions to execute in parallel. Because branches occur frequently (one branch every 5-10 instructions), processors must speculate past many branches to form a deep instruction window. Unfortunately, a single branch misprediction causes the processor to discard 100's of speculative instructions from the instruction window. Because the penalty is so high, even a seemingly mild misprediction rate (e.g., 5%-10%) profoundly limits ILP. Figure 1 shows the utilization gap between real and perfect branch prediction across varying issue widths, using a detailed cycle-level simulator of superscalar processor with a state-of-the-art perceptron branch predictor (Jimenez, et al., 2001) and a pipeline depth and memory hierarchy modeled after the Pentium-4. With perfect branch prediction, a large window is able to expose sufficient instruction-level parallelism in many benchmarks. However, with real branch prediction, a misprediction rate of only 5%-10% can significantly limit performance.

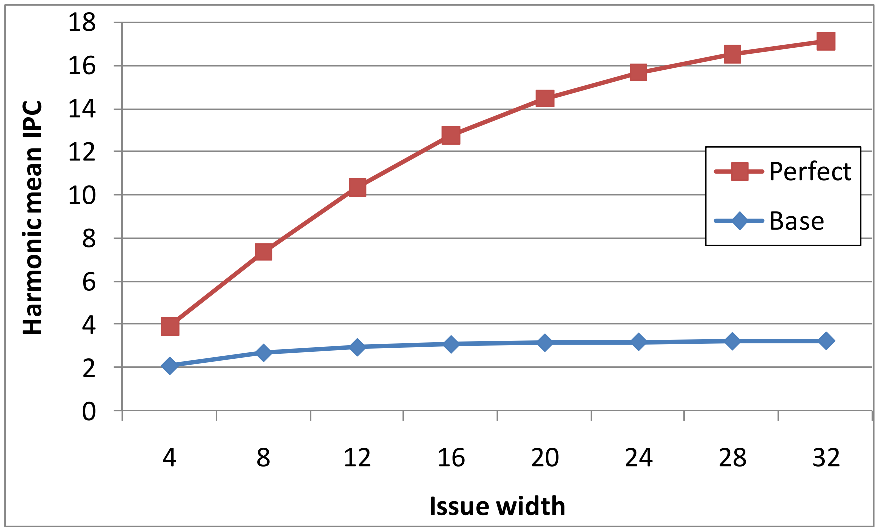


Figure 1. Harmonic mean IPC with perfect vs. real branch prediction, for SPEC95/SPEC2k integer benchmarks.

Because of the crucial role that branch prediction plays in extracting ILP, it has received much attention in past decades. Current branch predictors are able to achieve high degrees of accuracy (higher than 90% on most benchmarks). However, completely eliminating branch mispredictions remains an open challenge. Therefore, techniques for tolerating and reducing the penalty of branch mispredictions are increasingly important.

1.1 Branch misprediction tolerance techniques

Figure 2 shows a branch and the instructions following it. The branch can take one of two control-flow paths. The point where control-flow merges is called the reconvergent point (RP). Instructions between the branch and its reconvergent point are control-dependent (CD) on the branch – whether or not they are fetched depends on the branch outcome. Instructions after the reconvergent point are control-independent (CI) of the branch and will be fetched regardless of the branch outcome. However, some of the CI instructions are dependent on the branch outcome indirectly through data dependences (register or memory) and are labeled as control-independent data-dependent (CIDD) instructions. For example, in the figure, the consumer of R5 may get a different version of R5 depending on which path the branch takes. All other instructions in the CI region are control-independent data-independent (CIDI) instructions. CIDI instructions are truly independent of the branch and preserving them is the key to tolerating branch mispredictions.

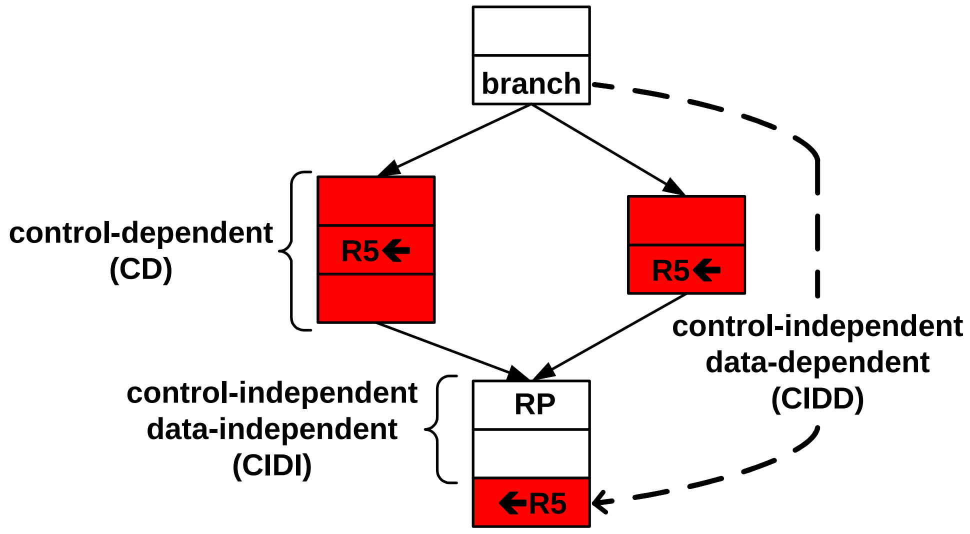


Figure 2. Example control-flow region.

Branch misprediction tolerance implementations can be divided into four techniques: multipath, predication, control independence with skipping (CI-skip), and control independence with speculation (CI-speculate).

- **Multipath:** Multipath fetches and executes both paths after the branch, duplicating the CI instructions. The wrong CD and CI instructions are discarded when the branch executes.
- **Predication:** Predication fetches and executes both CD paths up to the reconvergent point. Then, the CI instructions are fetched like usual. However, because the CI

instructions are not duplicated, the execution of CIDD instructions is delayed until the branch outcome is known.

- **CI-skip:** CI-skip does not predict the branch direction and instead jumps immediately to the reconvergent point and fetches the CI instructions. Only when the branch outcome is known does CI-skip fetch the correct CD instructions. Like predication, CI-skip must delay CIDD instructions until the branch outcome is known.
- **CI-speculate:** CI-speculate fetches and executes the predicted CD path and the CI instructions, like conventional speculation. If the branch was mispredicted, then, selective recovery is attempted upon resolving the branch outcome. Selective recovery requires fetching and executing the correct CD instructions, and selectively re-executing the CIDD instructions.

The way CD and CI instructions are handled affects (1) the degree of branch misprediction tolerance and (2) the penalty imposed on correctly predicted branches.

First thesis claim:

For a processor with realistic resources, the CI-speculate technique outperforms other branch misprediction tolerance techniques, because it does not penalize correctly predicted branches.

Multipath, predication, and CI-skip incur a penalty on correctly predicted branches. Multipath wastes fetch and execution bandwidth on the alternate path of a correctly predicted branch, including duplicating CI instructions after the reconvergent point. Predication fetches both CD paths of a correctly predicted branch, thereby wasting fetch and execution bandwidth on the alternate CD path, and also needlessly delays the execution of the CIDD instructions. CI-skip needlessly delays the execution of correctly predicted CD instructions and the CIDD instructions that depend on them. In practice, these penalties on correctly predicted branches partially or fully offset any gains from tolerating mispredicted branches. In some cases, performance is degraded with respect to conventional speculation.

In this dissertation, the four branch misprediction tolerance techniques are qualitatively and quantitatively compared. The comparison reveals that CI-speculate is the best performer.

1.2 Transparent Control Independence (TCI)

Since CI-speculate is the best performer, we next focus on analyzing CI-speculate implementations to understand their limitations. Based on this analysis, we develop a new CI-speculate implementation.

Second thesis claim:

To achieve the full potential of CI-speculate, the microarchitecture must

truly decouple misprediction-independent (CIDI) instructions

from misprediction-dependent (CD & CIDD) instructions.

Prior CI-speculate implementations do not truly decouple CIDI instructions from CD and CIDD instructions. The root cause is that they explicitly maintain program order. The microarchitecture contribution of this dissertation is Transparent Control Independence (TCI). TCI decouples CIDI instructions from CD and CIDD instructions, by focusing on repairing program state instead of program order. TCI fully capitalizes on the work performed by CIDI instructions, by not wasting bandwidth on CIDI instructions during selective branch misprediction recovery and not delaying the freeing of CIDI instructions' resources.

Explicit order is maintained by prior CI-speculate implementations for two main reasons:

1) Previous implementations evolved from reorder buffer (ROB) based designs. The ROB buffers all instructions in program order to implement in-order retirement. Hence, the late-fetched correct CD instructions need to be reordered with respect to the early-fetched CI instructions.
2) When CIDD instructions re-execute with changed values from the repaired CD region, they may also need to re-reference *unchanged* values from CIDI instructions. Ultimately, this means dependencies need to be maintained or recreated among co-mingled CIDI and CIDD instructions.

Implementations that explicitly maintain program order sacrifice design efficiency and performance.

We propose that it is sufficient to mimic the effect of program order between misprediction-independent and misprediction-dependent instructions. First, we depart from the traditional ROB-based substrate, in favor of a more resource-efficient checkpoint-based substrate (Akkary, et al., 2003) (Cristal, et al., 2004) (Hwu, et al., 1987) (Moudgill, et al., 1993). Leveraging coarse-grain retirement of a checkpoint-based substrate frees us from the fine-grain ordering constraint imposed by the ROB. Now, late-fetched correct CD instructions do not need to be reordered with respect to early-fetched CI instructions. Second, CIDD instructions are identified as they are fetched and their CIDI-supplied source values are checkpointed, breaking any dependencies on CIDI instructions. The CIDD instructions along with their checkpointed source values are set aside in a FIFO re-execution buffer (RXB) in preparation for recovery. This is the first implementation that truly decouples the CIDI instructions from the CIDD instructions.

When a branch is mispredicted, its incorrect CD instructions are fetched followed by CI instructions. All instructions – correct and incorrect – complete and speculatively release cycle-critical resources as they drain from the pipeline (physical registers, issue queue entries, etc.). When the mispredicted branch resolves, recovery is achieved by fetching a self-sufficient condensed "recovery program": the correct CD instructions (fetched from the instruction cache), the CIDD instructions (fetched from the RXB), and all input values needed to launch the correct CD and CIDD instructions (the branch's checkpoint and the checkpointed CIDI-supplied source values of CIDD instructions). Recovery is effectively transparent to the pipeline, in that speculative state is not rolled back and recovery appears as a jump to code. TCI yields a highly streamlined pipeline that quickly recycles resources based on conventional speculation, enabling a large window with small cycle-critical resources, and prevents many mispredictions from disrupting this large window.

Figure 3 shows a high-level view of TCI. Dynamic instructions are shown from left to right in the order in which they are fetched (fetch time). Correctly fetched and executed instructions are shown in white and incorrectly fetched or executed instructions are shown in gray. Correctly fetched instructions are labeled with their order in sequential program order (incorrect CD instructions are labeled with x's instead). A branch is mispredicted at the beginning of the fetch timeline. Thus, incorrect CD instructions are fetched first followed by

CIDI and CIDD instructions. The first correctly fetched instruction is instruction 4. Sometime later, after fetching instruction 14, the misprediction is finally detected. At this point, the independent (thanks to input values from the branch's checkpoint and RXB) recovery program is fetched. Notice the relaxed order: the recovery program's instructions 1, 2, 3, 6', 10', and 12' come after the speculative program's instruction 14 in the timeline. The pipeline does not differentiate between the speculative and recovery programs, as shown. The speculative state is not rolled back. Instead, the recovery program transparently repairs the speculative state.

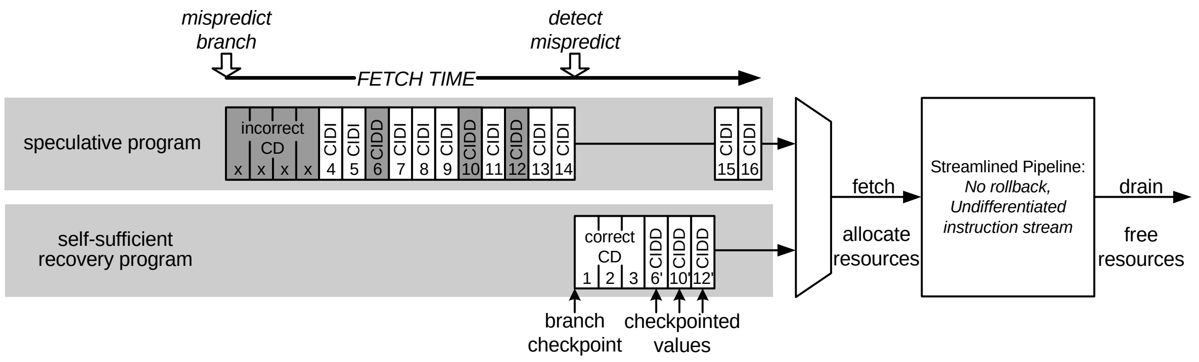


Figure 3. Transparent Control Independence (TCI).

1.3 Thesis contributions

This thesis makes the following chief contributions:

1) Comparison of branch tolerance techniques (Chapter 3):

- *Comparison of bandwidth overheads of branch misprediction tolerance techniques*. Various techniques (delay slots, multipath, static/dynamic predication, CI-skip, and CI-speculate) are analyzed based on branch coverage, branch misprediction penalty reduction, and overhead incurred by correctly predicted branches. Quantitative comparisons are also presented.

2) Analysis of CI-speculate approaches (Chapter 5):

- *Analysis of overheads of repairing CD instructions.* The thesis analyzes issues associated with removing the wrong CD instructions from the middle of the window and inserting the correct CD instructions in the middle of the window. Processor resources impacted by CD repair are identified and possible solutions are discussed.
- *Comparison of resource and bandwidth overheads for repairing CIDD instructions*. The thesis analyzes factors that reduce the performance of previous CI-speculate approaches and quantifies the impact of these factors.

3) Transparent Control Independence (Chapter 4 and Chapter 6):

- *TCI concept and microarchitecture*. A new approach is proposed that fully decouples misprediction-independent instructions from misprediction-dependent instructions, yielding a highly streamlined microarchitecture for exploiting control independence. The key insight is checkpointing CIDI-supplied source values of CIDD instructions. Another important aspect is using a relaxed, coarse-grain retirement substrate.

- *Identifying CIDD instructions*. Novel mechanisms are developed for assembling the CIDD instructions: the control-flow stack (CFS) for detecting arbitrary and nested reconvergent points, predicting the influenced register set (IRS), poisoning registers for identifying CIDD instructions, branch-sets for identifying CIDD loads, etc.
- *RXB reconstruction*. Since CIDD slices of multiple branches are co-mingled within the RXB, servicing a branch misprediction may require repairing CIDD slices of other branches and selectively removing CIDD instructions of the resolved branch. A simple unified solution – identify CIDD instructions in the recovery program itself, as was done the first time for the speculative program – enables arbitrary adjustments to the RXB while preserving its simple FIFO policy.
- *Renaming partial programs*: We propose a novel technique for renaming the recovery program despite its CIDI gaps.

1.4 Thesis organization

Chapter 2 discusses the experimental method followed in this thesis. Chapter 3 describes how branch misprediction tolerance techniques function, discusses relevant related work, and provides a qualitative and quantitative comparison among the techniques. Chapter 4 discusses control independence support mechanisms. Chapter 5 investigates control independence implementations and challenges, including qualitative and quantitative comparisons with related work. Chapter 6 presents the TCI microarchitecture in detail, including results and additional related work. Finally, Chapter 7 provides a summary and future work.

Chapter 2

Experimental method

2.1 Simulators

I have developed two custom timing simulators. Both use the PISA ISA from the Simplescalar toolkit (Burger, et al., 1996).

2.1.1 Trace-driven timing simulator

In Chapter 3, I investigate previous branch misprediction tolerance techniques and evaluate their effectiveness. I use a fast trace-driven timing simulator to generate the results. The simulator focuses on modeling fetch bandwidth, execution bandwidth, and true dependencies among instructions in detail, since the various techniques tolerate branch mispredictions with different bandwidth requirements and dependency stalls. Structural resources are unbounded to assess the potential of all techniques. The window size is, however, limited to 8192 instructions.

Instruction fetch is modeled using an ideal trace cache and a perceptron branch predictor (Jimenez, et al., 2001). Oracle memory disambiguation is used. The memory hierarchy consists of a 64KB L1 data cache, a 64KB L1 instruction cache, and a 2MB unified L2 cache.

For the baseline, predication, CI-skip, and CI-speculate, the trace-driven simulator models fetch and execution bandwidth consumed by wrong-path instructions. On the other hand, in modeling multipath, we opted for an upper performance bound using some oracle information. First, the simulator oracally identifies the correct thread and only allows forking from this thread. This allows our multipath implementation to achieve higher performance by avoiding forking from the incorrect threads, saving fetch and execution bandwidth accordingly. Second, we only allow instructions from the correct thread to consume execution bandwidth. However, we do model sharing fetch bandwidth among all active threads. Despite only modeling sharing of fetch bandwidth, multipath performs weakly when compared with the other branch misprediction tolerance techniques being studied.

2.1.2 Detailed execution-driven timing simulator

The different CI-speculate architectures in Chapter 5 and the Transparent Control Independence architecture (TCI) in Chapter 6 are modeled using a detailed execution-driven cycle-level simulator. The simulator fetches and executes both correct and incorrect instructions as a real processor would, producing speculative values that affect the processor’s state, generating bad events such as load exceptions and so forth. A functional simulator is run independently and in parallel with the detailed execution-driven timing simulator to verify its retired outcomes.

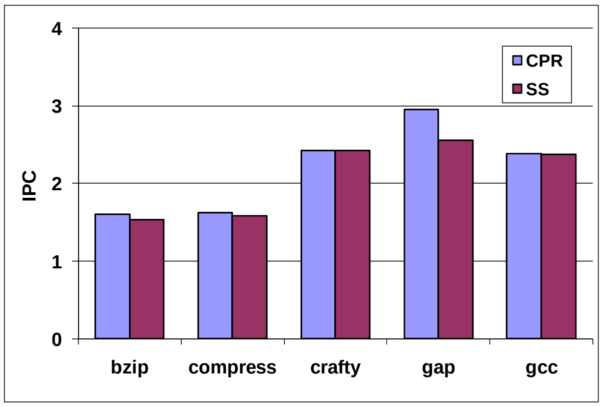
Table 1 shows the baseline microarchitecture parameters. For uniform comparisons, the baseline is TCI with the dynamic reconvergence predictor disabled, which ensures conventional (full) recovery for all branch mispredictions. Thus, the baseline is a checkpoint-based superscalar processor with aggressive register reclamation (Akkary, et al., 2003).

Table 1. Baseline microarchitecture configuration.

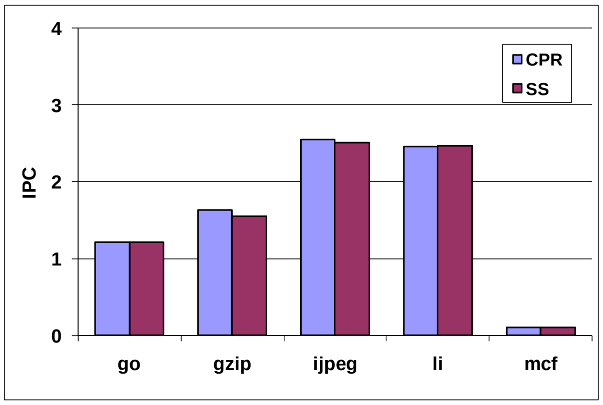
L1 I & D caches	**64KB, 4-way, 64B line, LRU, L1hit = 1 cycle**
L2 unified cache	**2MB, 8-way, 64B line, LRU, L2hit = 10 cycles, L2miss = 200 cycles**
Branch predictor	**perceptron (128KB)**
Memory dependence prediction	**store/branch sets**
Physical registers	**256**
Checkpoints	**16**
CFS	**16 entries**
Issue width	**4 or 8**
# pipeline stages	**20**
Issue queue	**32 or 64**
Load/store queue (LSQ)	**512**
Re-execution buffer (RXB)	**256**
Temp buffer (TB)	**128**

Checkpoint-based processors use cycle-critical resources efficiently. They can form very large logical windows with small cycle-critical physical resources. However, checkpoint-based processors introduce a penalty when servicing mispredicted branches that do not have checkpoints. In this case, misprediction recovery requires rolling back the processor state to the closest prior branch checkpoint, squashing even good instructions between the checkpoint and the mispredicted branch. On the other hand, conventional superscalar processors do not have this additional penalty.

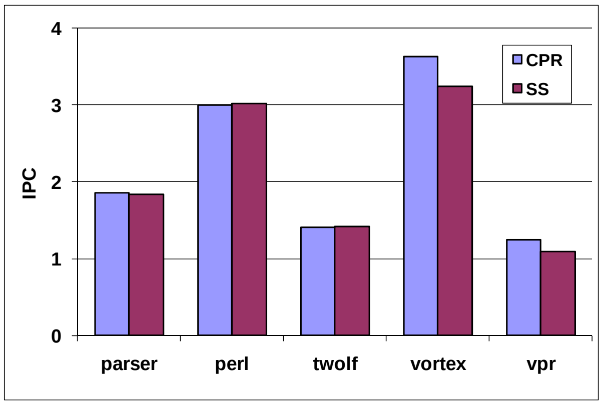
We compare our checkpoint-based baseline to a conventional ROB-based superscalar processor, to justify using the former as a baseline. Figure 4 shows IPCs of the checkpoint-based superscalar processor (CPR) and a conventional superscalar processor (SS) with equal resources. Figure 4(a)-(c) give the results of individual benchmarks, whereas Figure 4(d) shows several harmonic mean IPC results. In this experiment, CPR is allocated only 16 branch checkpoints, whereas SS is allocated an unbounded number of checkpoints. From the figures, we observe that CPR outperforms SS on average.



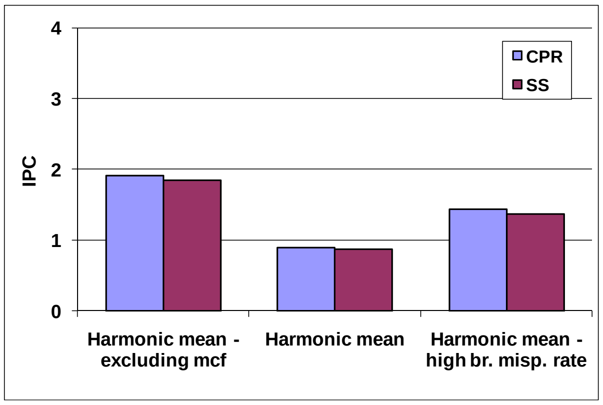
(a)



(b)



(c)



(d)

Figure 4. IPC for CPR baseline vs. SS baseline.

2.2 Benchmarks

We use 11 SPEC2K integer benchmarks and 4 SPEC95 integer benchmarks compiled with the gcc-based Simplescalar compiler (Burger, et al., 1996) for the PISA ISA with -O3 optimization. Reference inputs are used.

For all benchmarks, a single simulation point of 100 million instructions was selected using the SimPoint 3.2 toolkit (Sherwood, et al., 2002). In addition, predictors and caches are warmed up for 10 million instructions prior to starting the simulation point.

Table 2 shows benchmarks, inputs, and selected simulation points.

Table 2. Benchmarks.

Benchmarks	SimPoint 3.2 (100m)
bzip2-program-ref	406
compress95-bigtest-ref	374
crafty-ref	1466
gap-ref	1619
gcc-expr-ref	89
go95-5stone21-ref	138
gzip-graphic-ref	774
ijpeg95-specmun-ref	84
li95-ref	329
mcf-ref	441
parser-ref	2803
perlbmk-diffmail-ref	117
twolf-ref	1075
vortex-two-ref	407
vpr-route-ref	528

Chapter 3

Branch misprediction tolerance techniques

Branch prediction can only improve the performance of a superscalar processor, because the processor will otherwise stall waiting for the branches to execute. This significant performance gain is a result of keeping the window full of useful instructions. However, branch prediction has the disadvantage of wasting fetch and execution bandwidth when a misprediction occurs limiting the effective window size. Branch misprediction tolerance techniques like branch delay slots (Hennessy, et al., 1982) (Gross, et al., 1982) (Patterson, et al., 1981), multipath (Ahuja, et al., 1998) (Heil, et al., 1996) (Klauser, et al., 1998) (Uht, et al., 1995) (Wallace, et al., 1998) (Wallace, et al., 1999), predication (Allen, et al., 1983) (Kim, et al., 2006) (Kim, et al., 2005) (Klauser, et al., 1998) (Mahlke, et al., 1995) (Smith, et al., 2006), and control-independence (Al-Zawawi, et al., 2007) (Hilton, et al., 2007) (Cher, et al., 2001) (Chou, et al., 1999) (Gandhi, et al., 2004) (Rotenberg, et al., 1999) (Rotenberg, et al., 1999) (Sodani, et al., 1997) have the potential to reduce the penalty associated with branch mispredictions.

3.1 Branch delay slots

When pipelined processors were first introduced, processors had to deal with the fact of not having the outcomes of branches at fetch time. The outcome of the branches would only be known some cycles later when the branch instructions traveled down the processor pipeline and executed. This introduced a control hazard in the pipeline. The simplest remedy for the control hazard was to insert stalls in the pipeline from the time the branch is fetched and until the branch outcome is known. This solution degraded the potential performance of pipelining since the processor is stalling and not doing useful work.

Delayed branches were one of the first techniques to tolerate branch stalls in pipelined processors. This technique attempts to overlap branch stall cycles with useful instructions by delaying acting on the outcome of the branch. The ISA architects a fixed number of branch delay slots that are guaranteed to be fetched and executed after the branch regardless of its outcome. A compiler would then try to fill the branch delay slots with branch-independent instructions. If the branch delay slots are completely filled, then the branch stalls are completely hidden, otherwise, only partial tolerance is achieved. This technique evolved on single-issue in-order processors, where the branch stall penalty was only a few instructions (1-3 instructions). In fact, the compiler cannot always fill even a single delay slot (Gross, et al., 1982).

As an alternative to branch delay slots, branch prediction was later successfully used to overcome the control hazard in pipelined processors. Moreover, branch prediction is capable of tolerating hundreds of stall cycles in modern out-of-order processors. Unfortunately, branch prediction is not always effective in tolerating a control hazard. Occasionally, branch predictions are wrong and the stall cycles are exposed, causing significant performance loss. In these situations, branch delay slots can play a limited role in minimizing the impact of a branch misprediction by reducing the misprediction penalty by a few instructions. In a modern processor, a branch misprediction can have a penalty that spans hundreds of instructions. Filling this many delay slots, statically, is impractical. Hence, branch delay slots are not considered a viable solution to tolerate branch mispredictions in the presence of more effective techniques.

3.2 Multipath

Multipath reduces the penalty of a branch misprediction by speculatively executing both paths of a branch. When the branch outcome is known, the incorrect path is discarded. This allows multipath to avoid part of the misprediction penalty, since some instructions on the correct path have been executed.

Figure 5 presents an example of covering a branch misprediction with multipath. After fetching the branch in Figure 5(a), the processor follows both possible paths. Hence, the processor avoids the need to predict a single path to follow. Figure 5(b) shows how both the wrong CD instructions from path 1 and the correct CD from path 2 are fetched and executed simultaneously. Notice that the CI instructions are duplicated and that the CIDD instruction consuming R5 will execute with different source operands independently on both paths. Unfortunately, CIDI instructions will needlessly execute redundantly on both paths, since we know that their results will be the same. In Figure 5(c), the branch outcome is known and the wrong path is discarded. Resources are reclaimed from path 1 and reallocated toward path 2.

Multipath branch misprediction tolerance comes at the price of dividing the processor’s resources between correct and incorrect execution paths. This facet can also degrade the performance of correctly predicted branches covered by multipath needlessly, because not all resources are allocated to the correctly predicted path as would be the case in a normal processor. This problem is worsened when covering multiple branches with multipath, as the number of simultaneous paths pursued increases exponentially with the number of covered branches. Prior implementations of multipath have lessened the impact of this problem by selectively applying multipath based on branch confidence (Heil, et al., 1996) (Klauser, et al., 1998) (Wallace, et al., 1998) and other heuristics (Uht, et al., 1995).

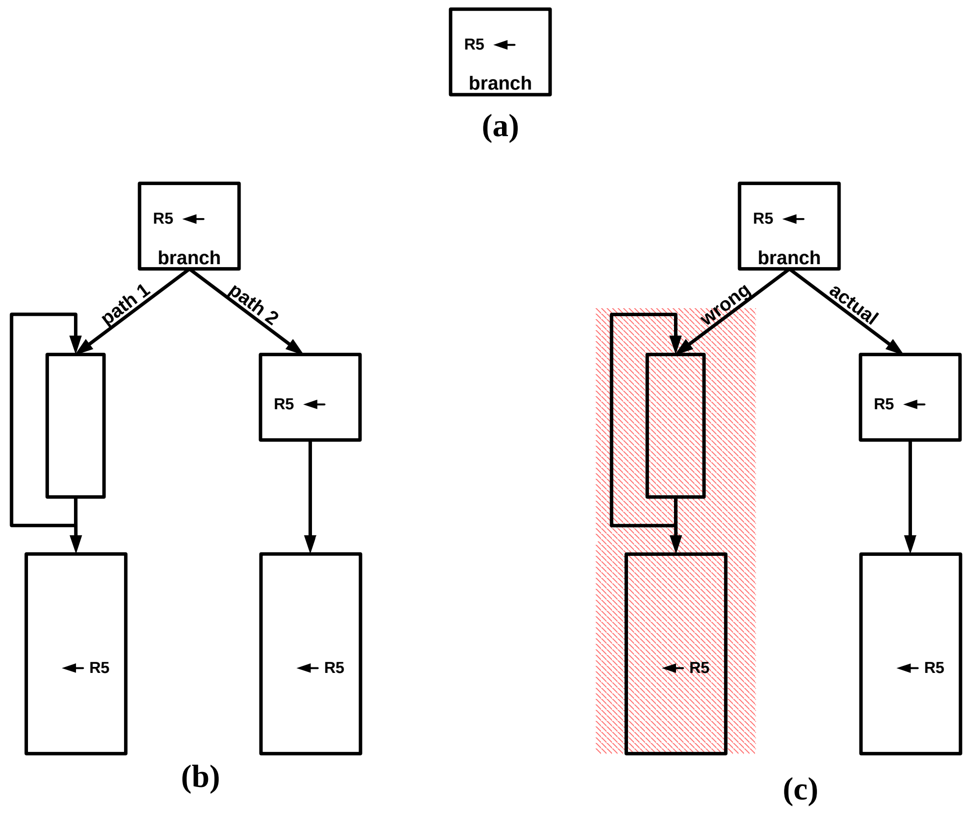


Figure 5. Multipath example.

3.3 Predication

Predication takes a different approach to branch misprediction tolerance. In predication, all control-dependent instructions are fetched regardless of the branch outcome and the execution of the control-independent data-dependent (CIDD) instructions is delayed until the branch is resolved. In doing so, predication avoids the need to predict covered branches, hence, the branch mispredictions and their negative effects are eliminated altogether. Predication can be performed statically by a compiler (Allen, et al., 1983) (Mahlke, et al., 1995), or dynamically in the processor (Kim, et al., 2006) (Kim, et al., 2005) (Klauser, et al., 1998).

Static predication leverages ISA support and typically can convert 30% of branches (Tyson, 1994). Static predication's low coverage can be attributed to:

1) Some control-flow constructs are hard to represent in static form after being predicated (for example: loops).
2) Predication may require in-lining some functions. This increases the size of the code and could be a limiter on the amount of predication that can be practically done.
3) Some branches do not have their targets available at compile time due to indirect calls or calls to functions in dynamically linked libraries.
4) Some branches may have many possible paths, such as switch statements, making predication unreasonable.

On the other hand, dynamic predication can partially overcome these challenges and achieves higher branch coverage, by leveraging dynamic information available to the processor and the ability of the processor to unwind complex control-flow dynamically into simple dynamic hammocks (Kim, et al., 2006) (Klauser, et al., 1998).

Predication avoids the need for predicting a branch but at a cost. In predication, both CD paths of the branch are fetched and the execution of CIDD instructions is delayed until the predicate outcome of the branch is known. This penalty affects all predicted branches regardless of whether or not the branches would have been correctly predicted or mispredictions under normal branch prediction. This indiscriminate penalty limits the benefit of predication and could even degrade performance. Dynamic predication, with the help of branch confidence, reduces this negative effect by trying to avoid predicating correctly predicted branches (Kim, et al., 2006) (Kim, et al., 2005) (Klauser, et al., 1998).

Figure 6 shows an example of covering a branch misprediction with dynamic predication. When the branch is first fetched in Figure 6(a), the processor identifies the branch's possible control paths either using branch prediction or leveraging information provided by the compiler. In Figure 6(b), the processor fetches both CD paths, one after another. Both the correct and incorrect CD instructions are executed. When the reconvergent point is reached in Figure 6(c), the processor continues to fetch the CI instructions. Since the branch outcome is not known, the processor needs to delay the execution of the CIDD instructions (guard the CIDD instructions) to avoid consuming the wrong source operands (Figure 6(d)). Once the branch outcome is known in Figure 6(e), the processor simply allows the CIDD to execute with the correct source operands. Predication typically uses proxy move instructions to forward the correct register productions from the CD region to the CI region.

In addition, we can optimize performance by stopping execution of the wrong CD instructions once the branch outcome is known, since their results are not needed.

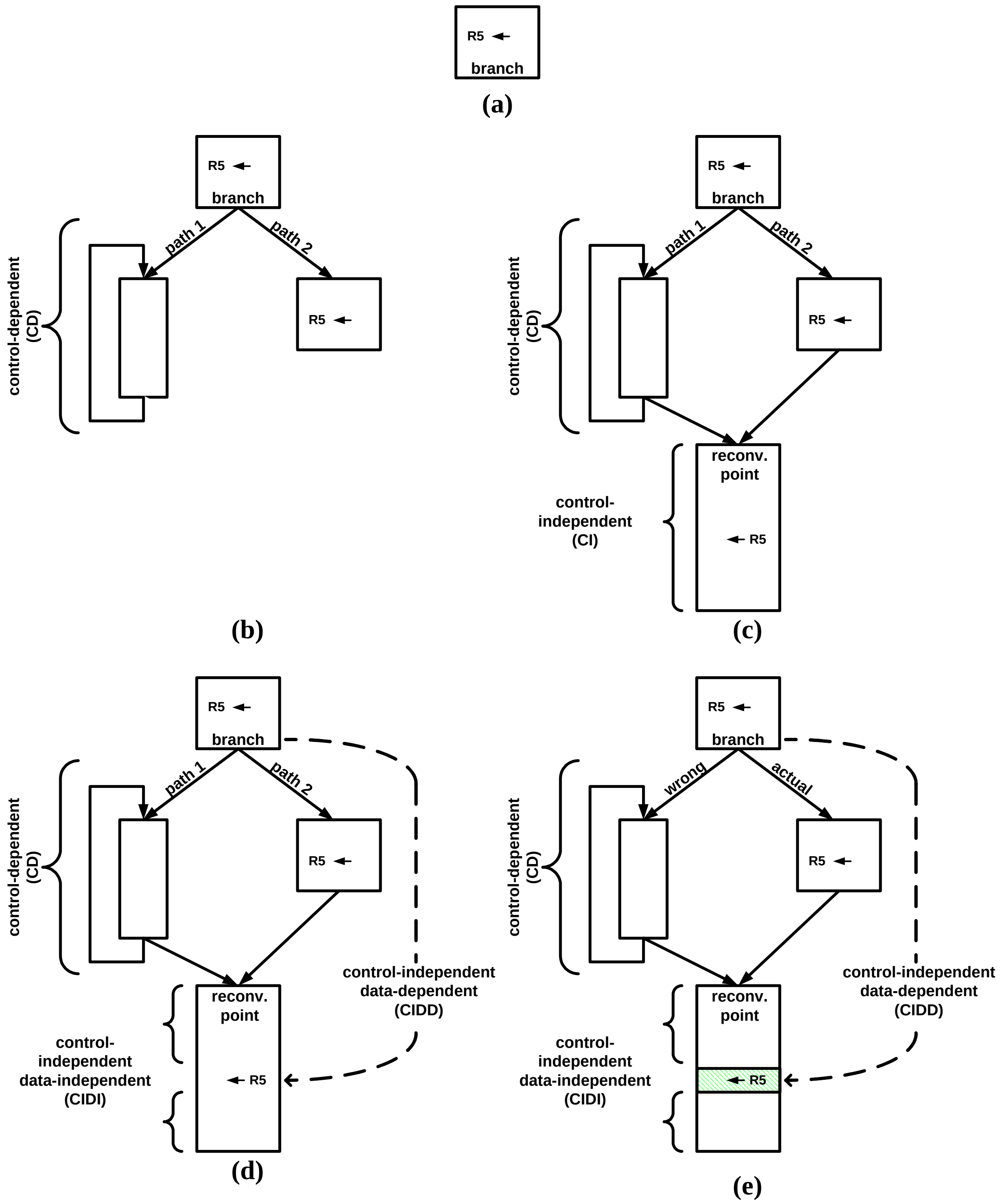


Figure 6. Dynamic predication example.

3.4 Control independence

Control independence is a dynamic technique that tolerates the penalty of a branch misprediction by selectively repairing the processor’s state, meanwhile preserving the work done by CIDI instructions (Al-Zawawi, et al., 2007) (Hilton, et al., 2007) (Cher, et al., 2001) (Chou, et al., 1999) (Gandhi, et al., 2004) (Rotenberg, et al., 1999) (Rotenberg, et al., 1999) (Sodani, et al., 1997). Two styles of control independence exist based on the way they handle the CD instructions. The first style of control independence skips over the CD instructions of a branch (the branch is not predicted) and then executes the CIDI instructions while guarding the execution of the CIDD instructions (CI-skip) (Cher, et al., 2001). When the branch outcome is known, the correct CD instructions are fetched and executed and then the guarded CIDD instructions are allowed to execute.

In Figure 7, we go through an example of covering a branch misprediction with CI-skip. After the branch is fetched in Figure 7(a), the processor needs to identify its reconvergent point. The processor then diverts fetch to the reconvergent point in Figure 7(b), avoiding fetching any CD instructions. Next, in Figure 7(c), CIDD instructions are identified and guarded since their source operands may change. When the branch outcome is known in Figure 7(d), the processor goes back and fetches the correct CD instructions. Finally, in Figure 7(e), the guarded CIDD instructions are allowed to execute.

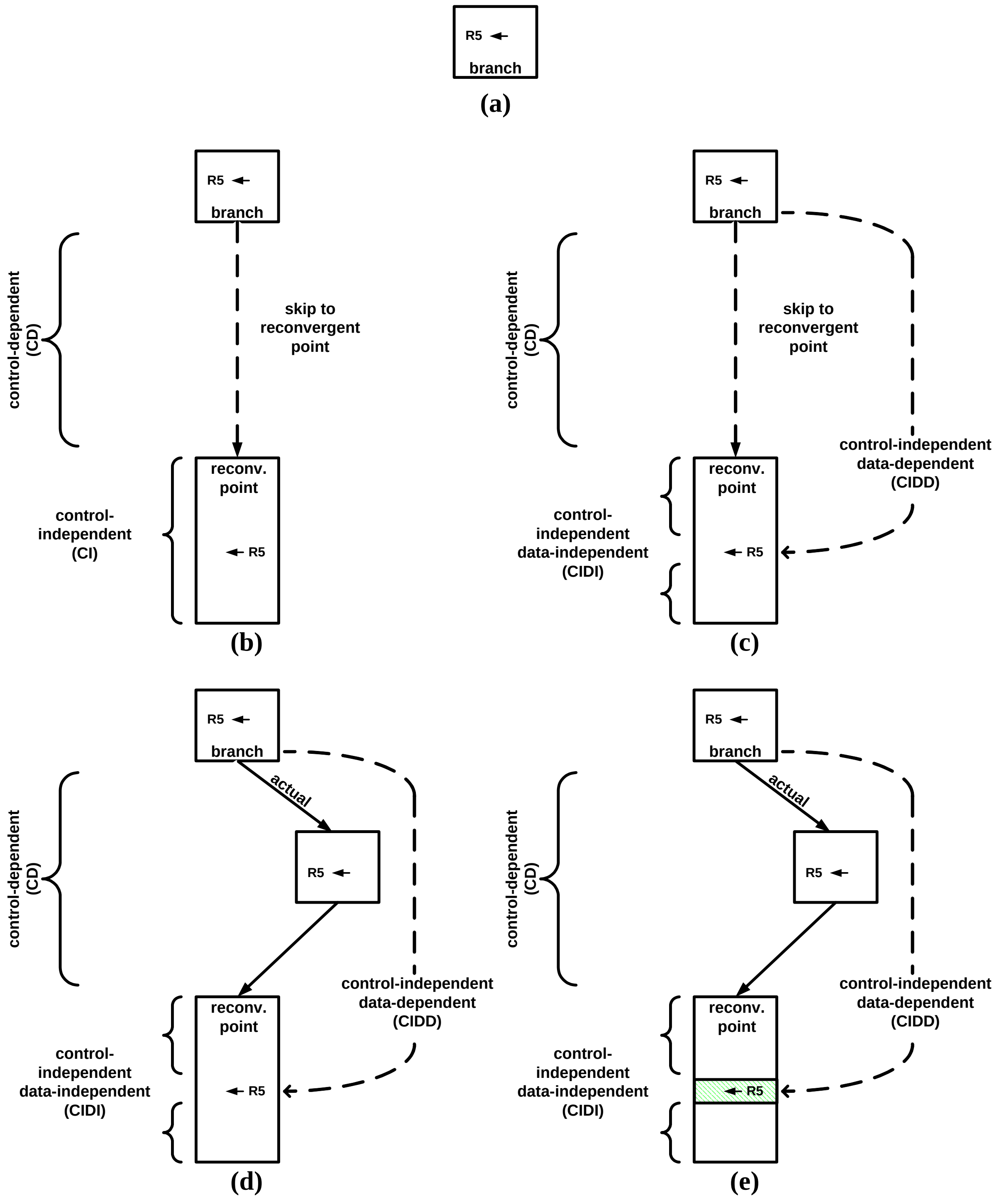

R5 ←
branch
(a)
R5 ←
branch
control-dependent (CD)
skip to reconvergent point
reconv. point
control-independent (CI)
← R5
(b)
R5 ←
branch
control-dependent (CD)
skip to reconvergent point
control-independent data-dependent (CIDD)
reconv. point
control-independent data-independent (CIDI)
← R5
(c)
R5 ←
branch
actual
control-dependent (CD)
R5 ←
control-independent data-dependent (CIDD)
reconv. point
control-independent data-independent (CIDI)
← R5
(d)
R5 ←
branch
actual
control-dependent (CD)
R5 ←
control-independent data-dependent (CIDD)
reconv. point
control-independent data-independent (CIDI)
← R5
(e)

Figure 7. Control-independence skip (CI-skip) example.

The second style of control independence speculates down the predicted path of the CD region and executes all CI instructions (CI-speculate) (Chou, et al., 1999) (Gandhi, et al., 2004) (Rotenberg, et al., 1999) (Rotenberg, et al., 1999) (Sodani, et al., 1997). When a branch misprediction is detected, the wrong CD instructions are selectively removed and replaced by the correct CD instructions, followed by selectively re-executing the CIDD instructions. CI-speculate only has to recover when a misprediction is detected, whereas CI-skip penalizes a correctly predicted branch (unless the correctly predicted branch has no CD instructions on the correct path and no CIDD instructions are encountered until the branch resolves). CI-speculate benefits from the fact that prediction is correct most of the time and avoids skipping correct CD instructions needlessly. So, CI-speculate achieves the performance of speculation by not degrading performance when a branch is correctly predicted. Moreover, CI-speculate is able to cover branch mispredictions and reduce their performance penalty.

Figure 8 shows an example of a branch misprediction covered by CI-speculate. After the branch is fetched in Figure 8(a), the processor predicts the outcome of the branch using the branch predictor. In Figure 8(b), we fetch the CD instructions on the predicted path. Once the reconvergent point is detected in Figure 8(c), the CD region has ended and we continue to fetch the CI region. Notice that all CI instructions are fetched and executed with the assumption that the prediction was correct. In Figure 8(d), the branch outcome is known and we must start to recover from the branch misprediction. We first squash the wrong CD instructions and go back to fetch the correct CD instructions from the actual path. Finally, in Figure 8(d), we re-execute the CIDD instructions to complete branch misprediction recovery.

CIDD re-execution is needed since these instructions have executed initially with wrong source operands produced by the wrong CD instructions.

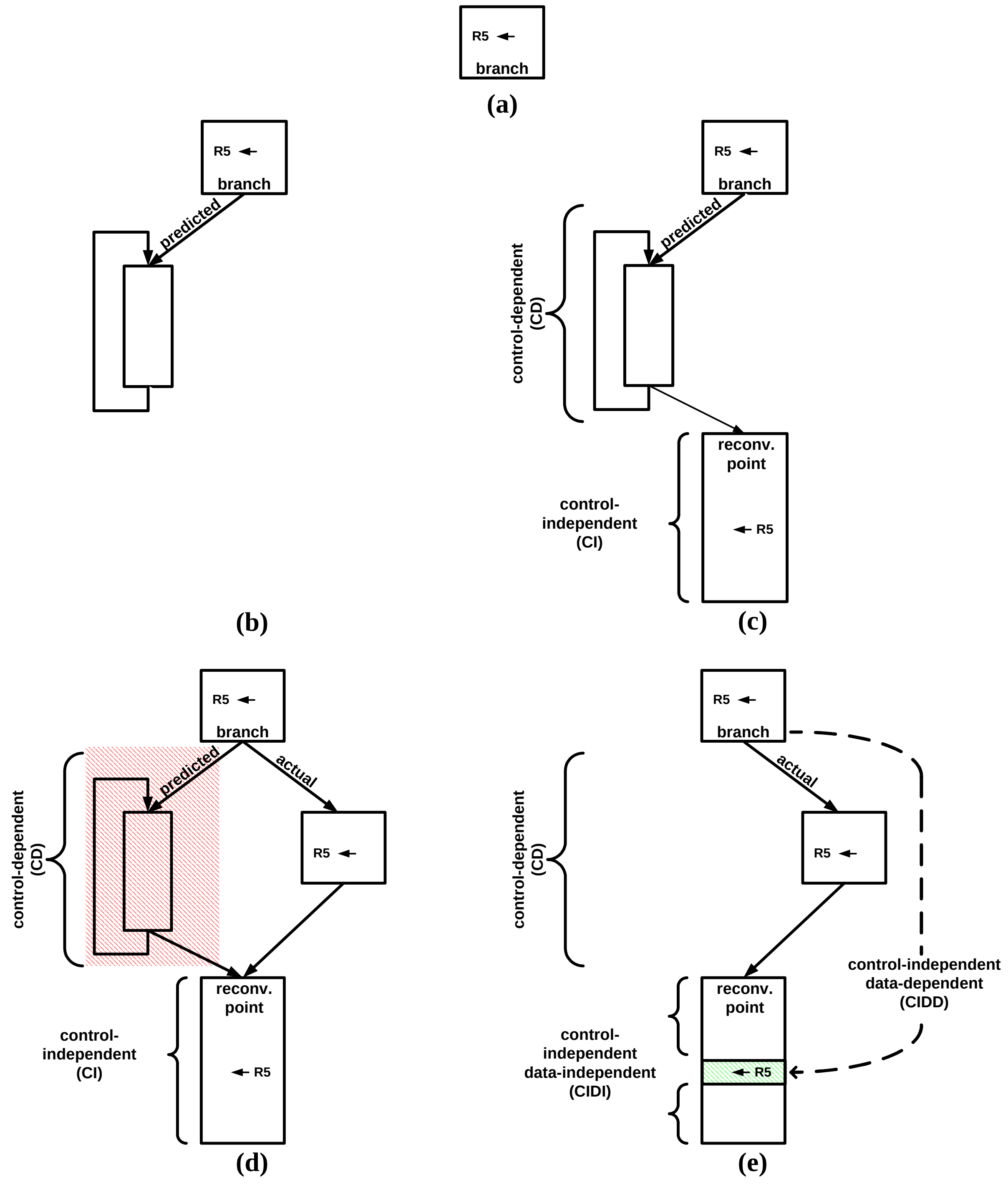


Figure 8. Control-independence speculate (CI-speculate) example.

3.5 Qualitative comparison

The goal of branch misprediction tolerance techniques is to achieve the performance of perfect branch prediction (all branches are correctly predicted). When using perfect branch prediction, the processor is always occupied by correct and useful instructions, and the processor does not encounter the delays and stall cycles associated with branch mispredictions.

Hence, the success of a branch misprediction tolerance technique depends on three factors:

1) Branch misprediction coverage: The percentage of branch mispredictions that can be covered by a given branch misprediction tolerance technique dictates the branch misprediction coverage.
2) Misprediction penalty: When a branch misprediction is covered with a given technique, the penalty still exposed to the processor in the form of stall cycles or wasted work (fetch and execution bandwidth) represents the branch misprediction penalty of the technique.
3) Correct prediction penalty: Since branch mispredictions are not known at the time of fetching the branch, we need to cover branches speculatively in anticipation of a misprediction. If a correctly predicted branch is covered, the tolerating technique may incur a penalty associated with this coverage. This penalty may degrade the performance of the overall system compared with conventional misprediction recovery which does not incur any penalty for a correctly predicted branch.

The ideal branch misprediction tolerance technique would cover all branch mispredictions, have no misprediction penalty, and have no penalty for covering a correctly predicted branch. Achieving this ideal solution is as challenging as achieving perfect branch prediction. We talked briefly about the different solutions toward the branch misprediction problem. Branch delay slots, multipath, static/dynamic predication, and skip/speculate control independence variants try to improve performance by reducing the penalty of branch mispredictions, however, each technique brings with it different compromises with respect to branch misprediction coverage, exposed misprediction penalty, and overhead of covering a correct prediction.

Multipath has the potential to cover all branch mispredictions but with a bandwidth requirement that grows exponentially with the number of unresolved branches in the window. This requirement can be potentially reduced using branch confidence. However, for a limited issue machine, multipath is not able to achieve reasonable performance despite good branch coverage. Whenever multipath covers a branch (whether correctly predicted or not), it takes away from the available processor bandwidth. So, by definition, multipath cannot achieve perfect utilization.

On the other hand, predication and control independence reduce the bandwidth requirements needed by multipath, by avoiding duplicating instructions after the reconvergent point is reached. This is very important, as we only waste bandwidth on the CD instructions. This brings us that much closer to our goal of perfect bandwidth utilization.

Static predication has low branch coverage (as discussed in Section 3.3). On the other hand, dynamic predication and control independence implementations achieve much higher branch coverage than static predication by leveraging the ability of the dynamic instruction window to inline all the code and resolve complex control-flow into simple sequential traces of instructions without any loops.

Predication, CI-skip, and CI-speculate differ in the way they deal with the misprediction-dependent instructions (CD instructions and CIDD instructions) of a covered branch. This difference dictates the amount of misprediction penalty that can be tolerated and the amount of penalty added to correctly predicted branches. In predication, both CD paths are fetched (and possibly executed, depending on the implementation) and the execution of the CIDD instructions is delayed (guarded) until the branch outcome is known. In CI-skip, both fetching the CD instructions and executing the CIDD instructions are delayed until the branch outcome is known. In CI-speculate, the predicted CD path is fetched and the CIDD instructions are allowed to execute speculatively. However, when a branch misprediction is detected, CI-speculate replaces the wrong CD instructions with the correct CD instructions and re-executes the CIDD instructions to repair their state.

Table 3 compares the different branch recovery models with respect to branch coverage, exposed misprediction penalty, and penalty incurred by covering correctly predicted branches. In the context of an aggressive superscalar processor, it is important to choose a branch misprediction tolerating technique that does not degrade the performance of the base system. We noticed that branch delay slots, predication and multipath have the potential to degrade performance when the branch is correctly predicted. This penalty can be reduced at the expense of reduced branch coverage, using branch confidence. In addition, branch delay slots do not have enough branch misprediction tolerance to cover the full penalty. On the other hand, Control independence represents the middle ground, exhibiting medium branch coverage (due to branches with no reconvergent points or branches with very large CD regions) and low overheads. Within control independence styles, we choose to trust the branch predictor and go with CI-speculate versus CI-skip that conceptually injects a misprediction at every skipped branch. Notice, CI-speculate is the only technique that does not degrade the performance of a correctly predicted branch, making it a suitable technique to complement conventional branch recovery.

Table 3. Comparison of branch misprediction tolerance techniques.

Recovery scheme	**Branch Coverage**	**Misprediction Penalty**	**Correct Prediction Penalty**
Squash-based recovery(Base)	High	Wrong CD + CIDD + CIDI	None
Branch delay slots	High	Base misprediction penalty – # filled delay slots	# empty delay slots
Multipath	High	Base misprediction penalty X (1 – 1 / #paths)	Base misprediction penalty X (1 – 1 / #paths)
Predication - static	Low	Wrong CD + CIDD	Wrong CD + CIDD
Predication - dynamic	Medium	Wrong CD + CIDD	Wrong CD + CIDD
CI-skip	Medium	CIDD	Correct CD + CIDD
CI-speculate	Medium	Wrong CD + CIDD	None

3.6 Quantitative comparison

3.6.1 Effect of branch confidence

Figure 9 shows the harmonic mean IPC results for a 4-issue processor. The four modeled branch misprediction tolerance techniques (CI-skip, CI-speculate, dynamic predication, and multipath) are run with varying confidence thresholds (TH: 4, TH: 8, TH: 16, and TH: 32) and varying maximum region sizes (RS= ∞, RS= 256, and RS= 32). The figure also shows the results for a processor with squash-based misprediction recovery (Base), a processor with perfect branch prediction (Perfect), and the branch misprediction tolerance techniques with oracle confidence (Oracle Conf).

From Figure 9 (a)-(b), we observe that Perfect achieves significant performance gains over Base and comes very close to the peak utilization of the processor. Perfect results mark the performance upper bound for any branch prediction or branch misprediction tolerance technique.

The upper bound for a specific branch misprediction tolerance technique can be observed by leveraging oracle branch confidence (Oracle Conf). With Oracle Conf, all branch mispredictions are covered, leading to the highest possible misprediction tolerance, and all correctly predicted branches are not covered, preventing possible performance degradations.

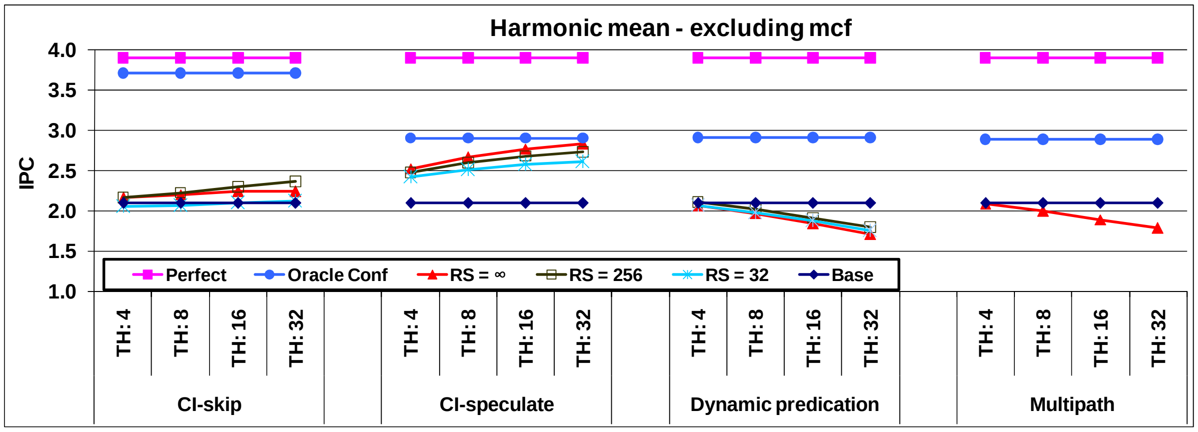
In Figure 9 (a)-(b), we notice that CI-skip with Oracle Conf has very high potential, coming close to Perfect. This confirms that CI-skip has very high branch misprediction tolerance. It is able to hide most of the branch misprediction penalty, exposing only the guarding of the CIDD instructions (see Table 3). On the other hand, CI-speculate, dynamic predication, and multipath only have medium performance potential with Oracle Conf. This can be attributed to the fact that all three models expose a bigger portion of the branch misprediction penalty when compared to CI-skip. In all three models, the wrong CD instructions component of the misprediction penalty is exposed (see Table 3). It is also interesting to see that these three models achieve very close results with oracle branch confidence, averaged over the benchmarks

Now that we have seen the upper bound of each technique using Oracle Conf, we investigate the real-world performance using real branch confidence. A 4096-entry branch confidence predictor with resetting counters (Jacobsen, et al., 1996) is used to generate the results. By varying the confidence threshold from 4 to 32, the number of branch mispredictions covered by the branch misprediction tolerance technique is increased at the cost of covering additional correctly predicted branches. In addition, the maximum branch region size covered is varied. The region size of a branch is an indicator of the cost of covering the branch with a given technique. The larger the branch region, the higher the cost to cover the branch and the less likely the benefit is.

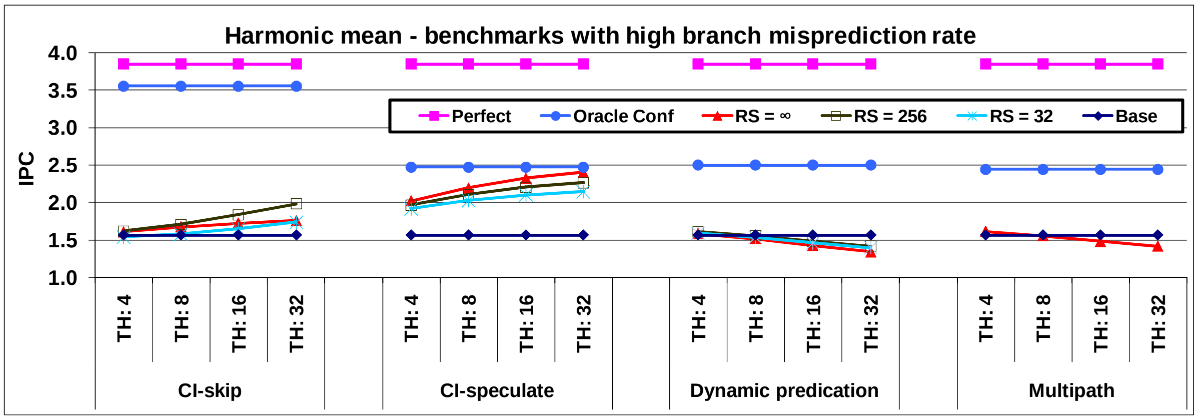
In Figure 9 (a)-(b), CI-skip shows slight improvement over Base with real branch confidence. The performance peaks with a branch confidence threshold of 32 and a maximum region size of 256. CI-skip performance is very far from the potential upper bound shown by Oracle Conf, because CI-skip introduces a branch misprediction when covering a correctly predicted branch. The added mispredictions add a penalty that offsets the savings of covering true mispredictions and could possibly degrade performance with respect to Base.

On the other hand, CI-speculate shows substantial performance gains compared to CI-skip. In fact, CI-speculate performance approaches the performance of Oracle Conf with a threshold of 32 and without setting a maximum region size. The reason for this phenomenon is that correctly predicted branches covered by CI-speculate do not perceive a penalty, allowing CI-speculate to cover many branches, mispredicted or not, with low overheads.

As for dynamic predication, it degrades performance with respect to Base. Dynamic predication performs best with a threshold of 4. Dynamic predication favors only covering relatively small region sizes below 256. This degradation can be attributed to the high cost of covering branches in a relatively narrow machine. Both multipath and dynamic predication are very sensitive to branch confidence. To achieve reasonable results in a narrow machine, we need a more accurate confidence predictor. Branch confidence is less of an issue in very wide machines, as the overhead of covering correctly predicted branches is reduced.



(a)

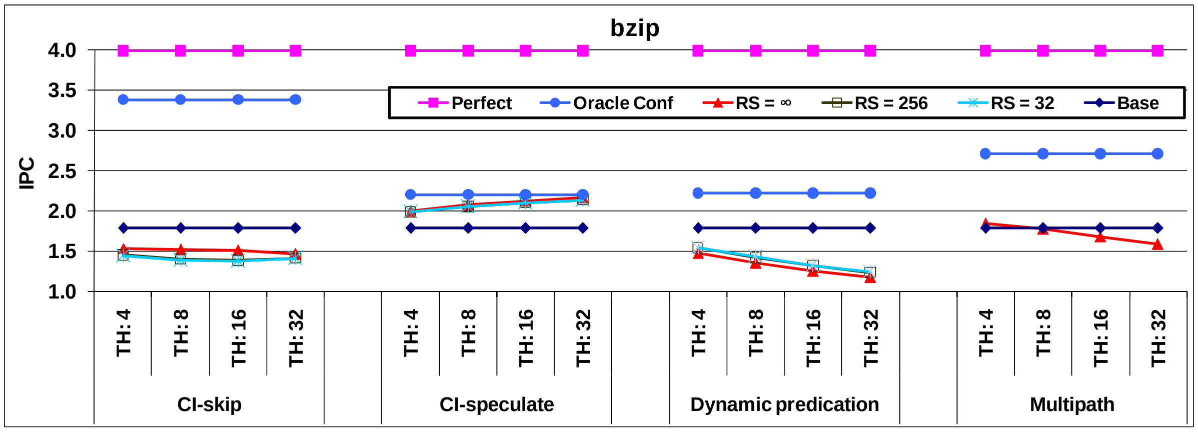


(b)

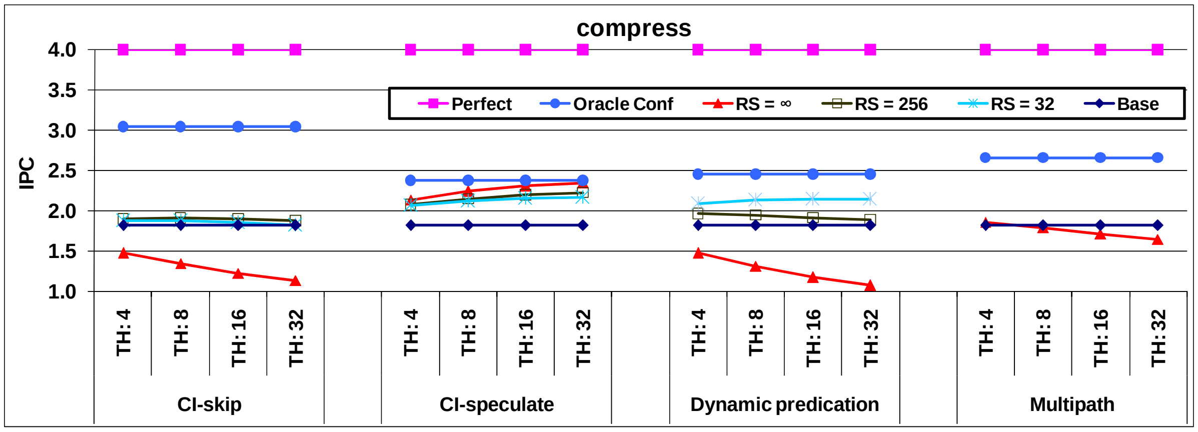
Figure 9. Branch misprediction tolerance techniques with varying confidence thresholds (TH) and maximum branch region size (RS).

Figure 10, Figure 11, Figure 12, Figure 13, and Figure 14 present the IPC results for CI-skip, CI-speculate, dynamic predication, and multipath for the individual benchmarks. Several trends can be observed:

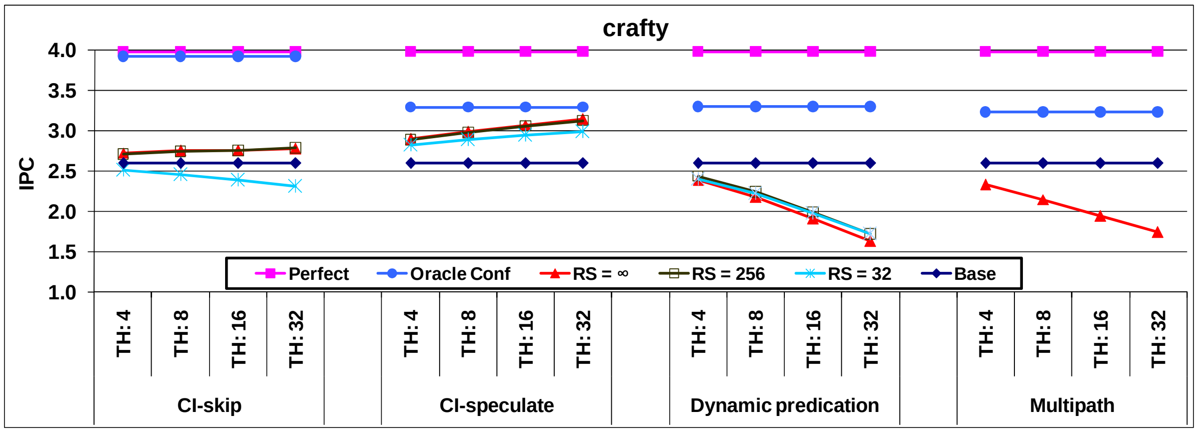
1) CI-speculate never degrades performance with respect to Base.
2) CI-speculate Oracle Conf results are lower than the Oracle Conf results of the other techniques in some benchmarks (for example: bzip, compress, and li). Even so, CI-speculate outperforms the other models in these benchmarks.
3) CI-skip outperforms CI-speculate in some benchmarks (for example: gap and twolf).
4) CI-skip has the potential to degrade performance with respect to Base (for example: bzip, compress, crafty, gcc, li, and perl).
5) CI-skip favors different confidence threshold levels with different benchmarks. In gap, for example, it favors a threshold of 32. However, in perl, it favors a threshold of 4.
6) Dynamic predication degrades performance most of the time in the narrow processor, but shows some improvement in some benchmarks (for example: compress, gzip, ijpeg, twolf, and vpr).
7) Multipath degrades performance on most benchmarks and only sees slight speedups in vpr. The reason is that multipath is not suited for narrow processors and prefers very wide processors.



(a)

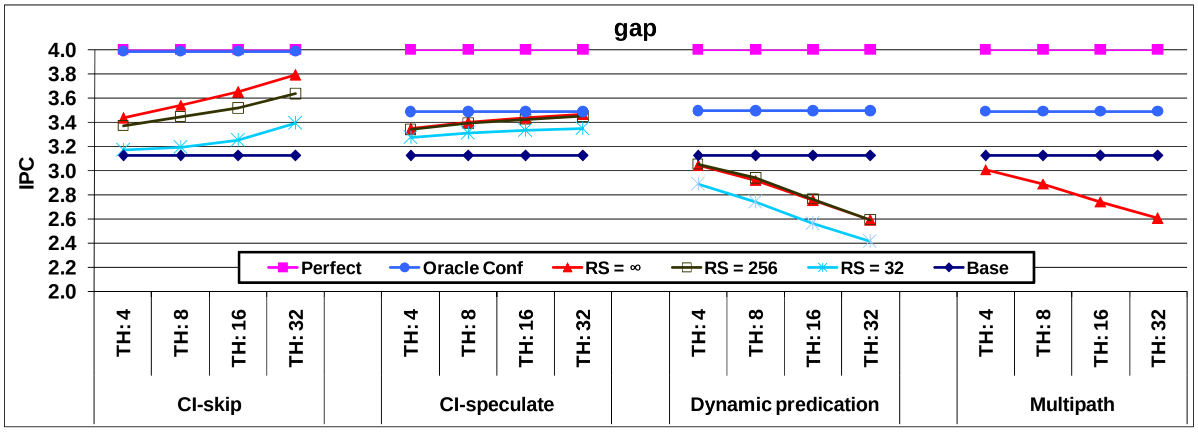


(b)

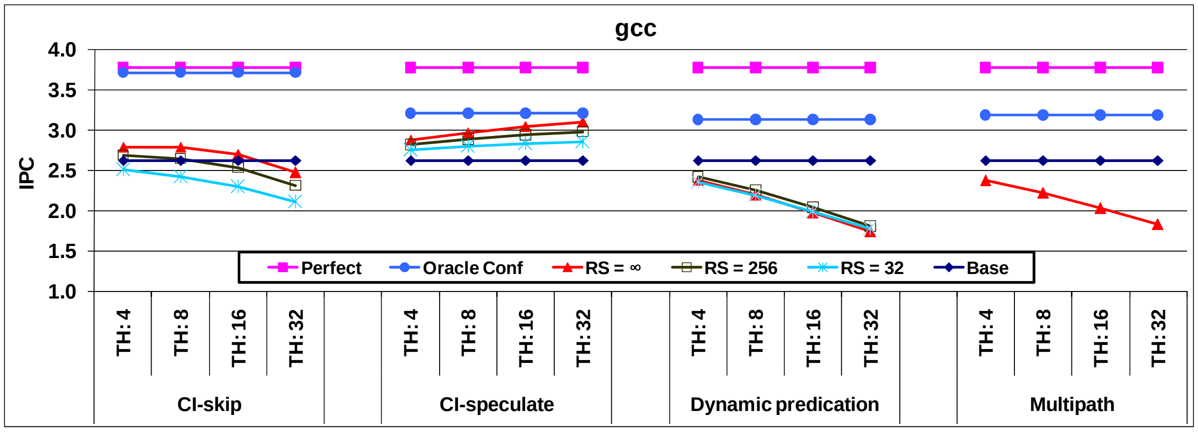


(c)

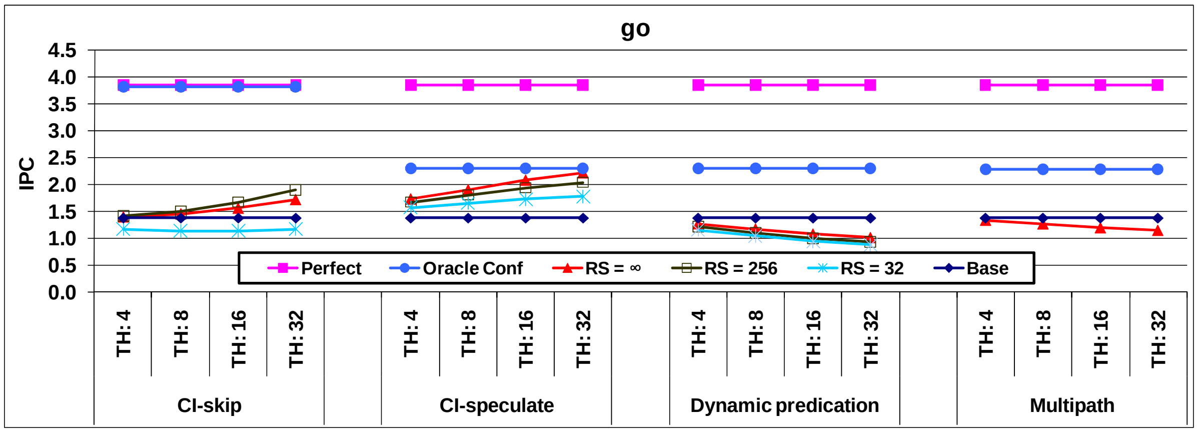
Figure 10. Branch misprediction tolerance techniques with varying confidence thresholds (TH) and maximum branch region size (RS) (individual benchmarks).



(a)

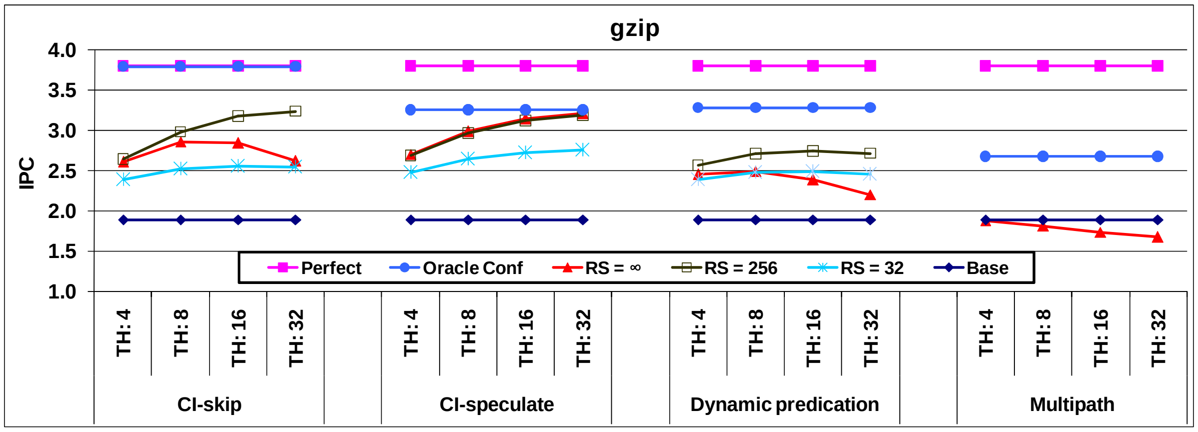


(b)

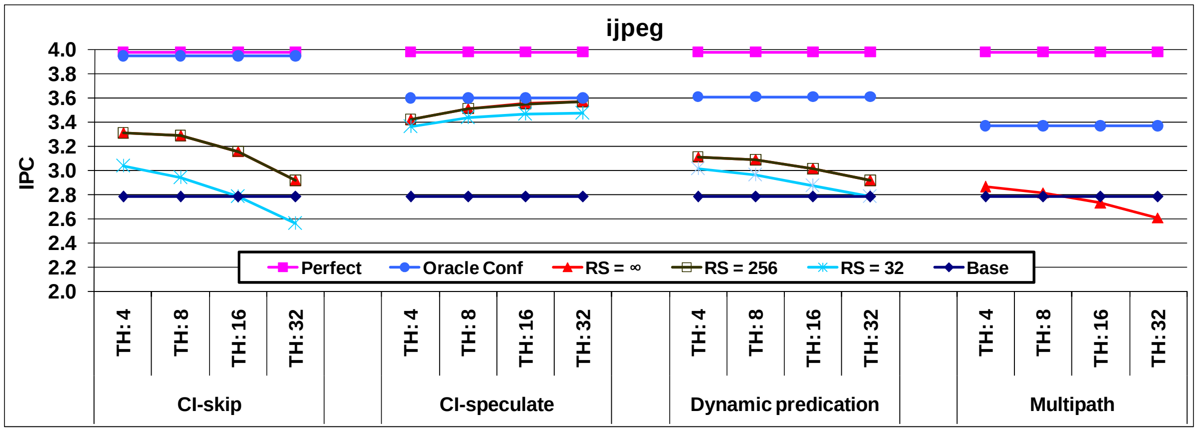


(c)

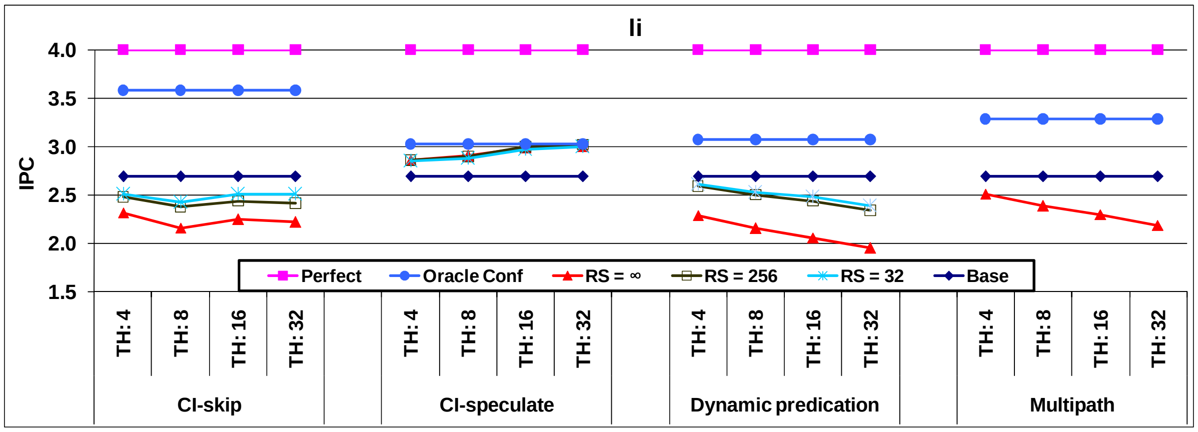
Figure 11. Branch misprediction tolerance techniques with varying confidence thresholds (TH) and maximum branch region size (RS) (individual benchmarks).



(a)

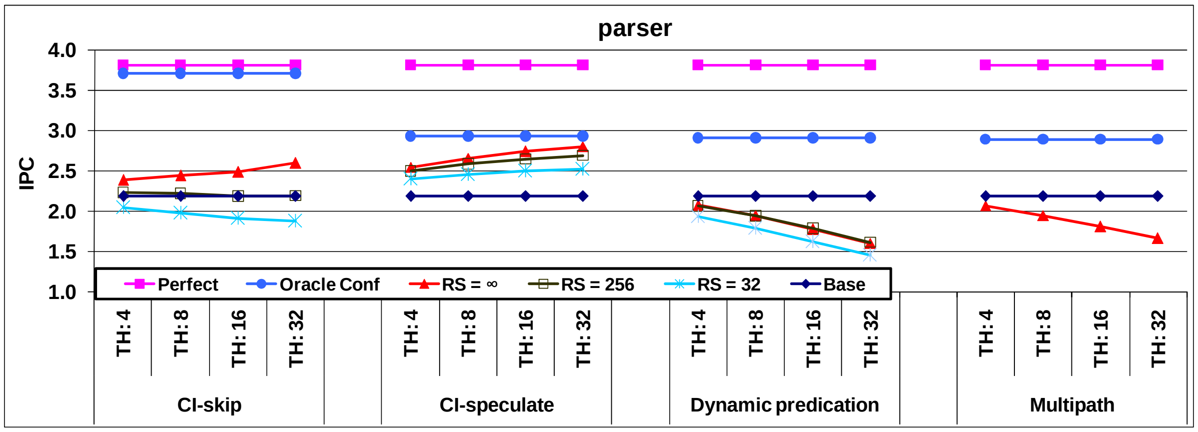


(b)

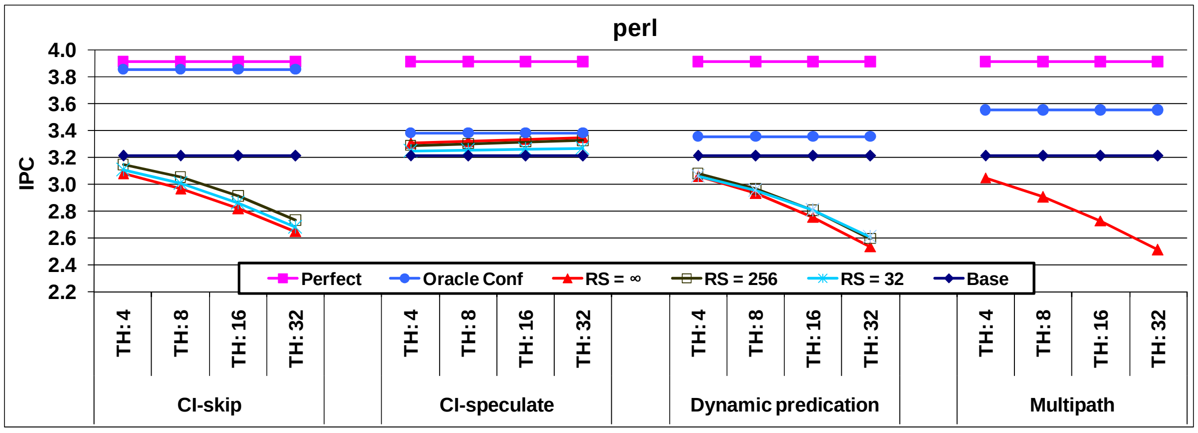


(c)

Figure 12. Branch misprediction tolerance techniques with varying confidence thresholds (TH) and maximum branch region size (RS) (individual benchmarks).

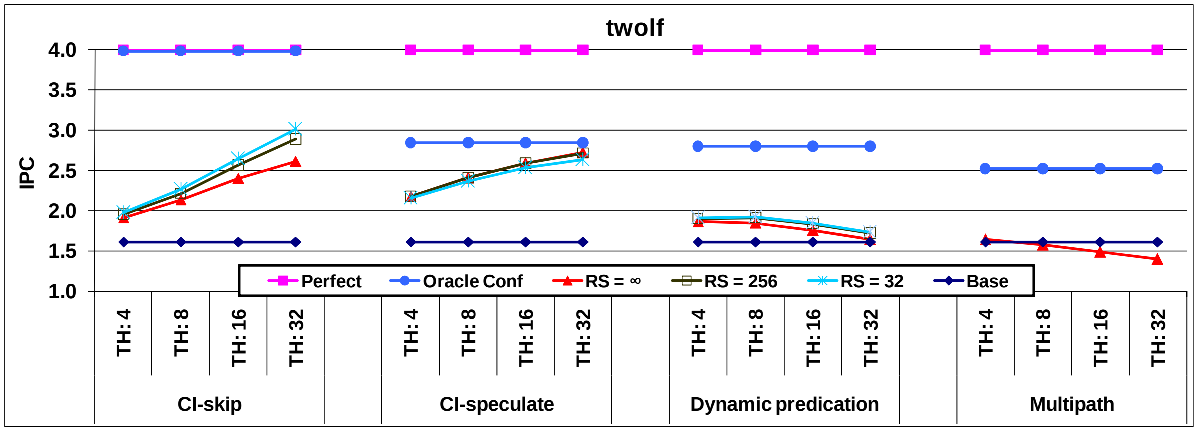


(a)

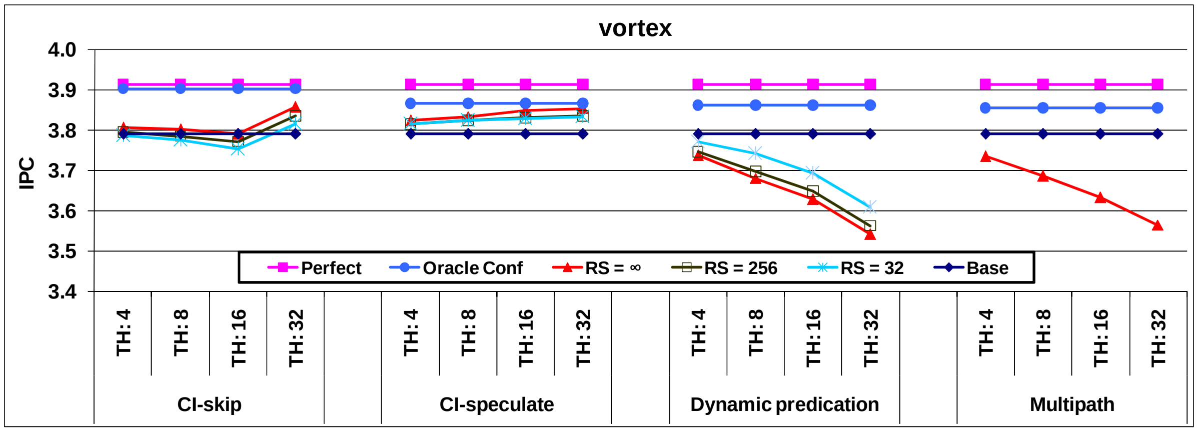


(b)

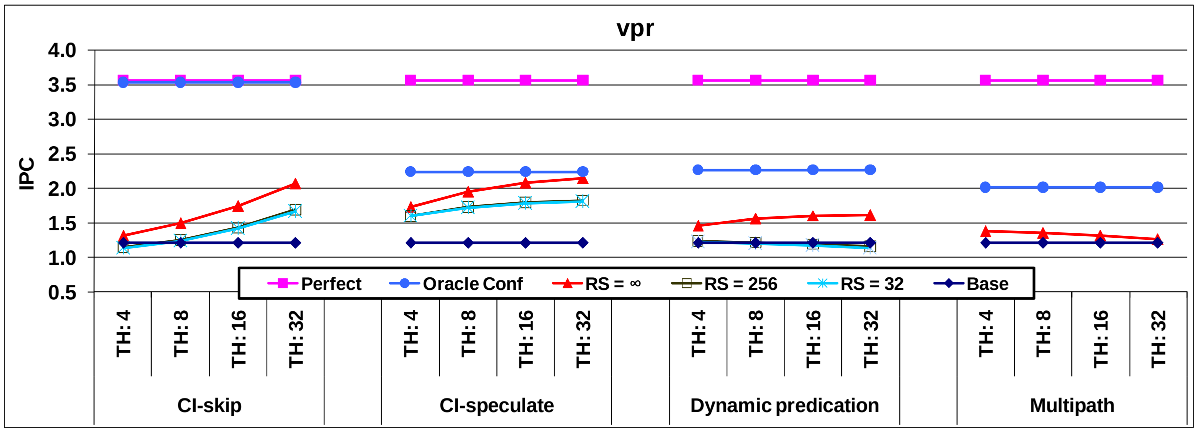
Figure 13. Branch misprediction tolerance techniques with varying confidence thresholds (TH) and maximum branch region size (RS) (individual benchmarks).



(a)



(b)



(c)

Figure 14. Branch misprediction tolerance techniques with varying confidence thresholds (TH) and maximum branch region size (RS) (individual benchmarks).

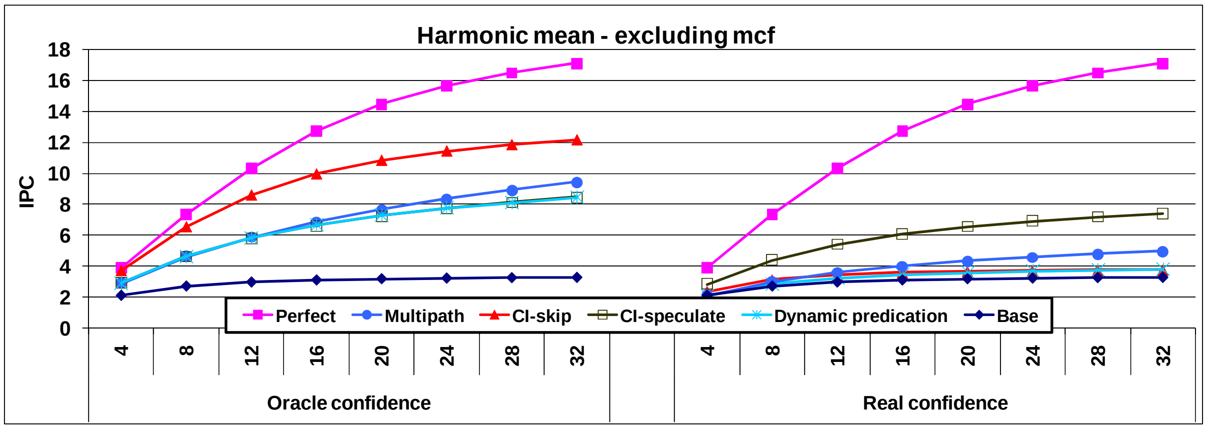
3.6.2 Performance potential in wider processors

Figure 15 shows the harmonic mean IPC results of six models with varying issue widths (issue width: 4 to 32). The four modeled branch misprediction tolerance techniques are shown (CI-skip, CI-speculate, dynamic predication, and multipath), with oracle branch confidence on the left and real branch confidence on the right. Each branch misprediction tolerance technique is run using the branch confidence threshold (TH) and the maximum branch region size (RS) that maximizes its performance (leveraging the results presented in Section 3.6.1). The figure also shows results for a processor with squash-based misprediction recovery (Base) and a processor with perfect branch prediction (Perfect).

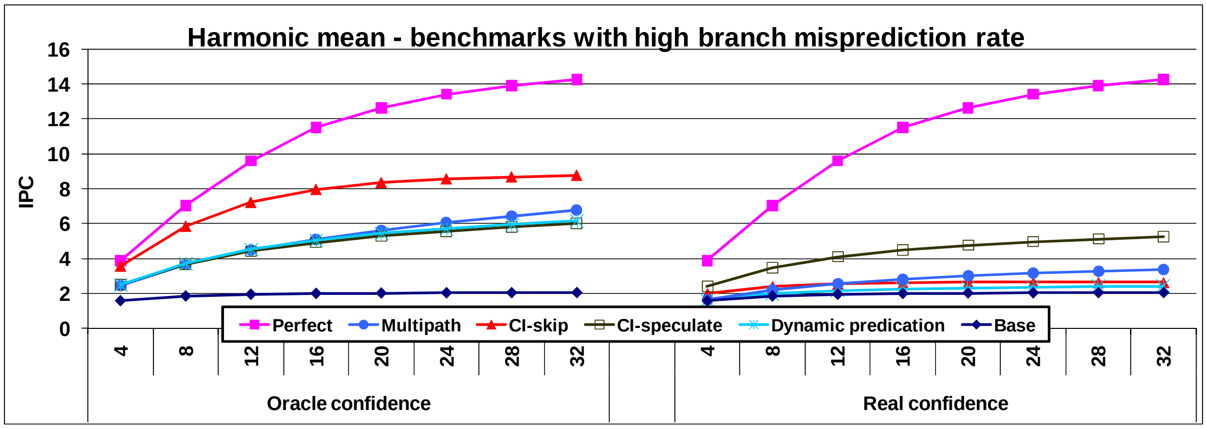
In Figure 15 (a)-(b), we observe that Perfect achieves significant improvement over Base. Perfect achieves close to full utilization of the processor bandwidth with low issue widths (less than 8 issue width). However, as the issue width is increased, Perfect fails to achieve the full utilization of the machine due to the limited ILP available to the processor's window (the window size is 8192 entries as described in Section 2.1.1).

In Figure 15 (a)-(b), we notice that CI-skip continues to outperform all branch misprediction tolerance techniques when applying oracle confidence. CI-speculate, dynamic predication, and multipath have similar results across the different issue widths with oracle confidence. In addition, multipath continues to show improvement potential at issue widths higher than 24.

When applying real branch confidence to the branch misprediction tolerance techniques, we observe that CI-speculate outperforms CI-skip, dynamic predication, and multipath. Although the performance of multipath has fallen with real confidence, it still shows performance potential with increased issue width, allowing it to outperform CI-skip and dynamic predication.



(a)

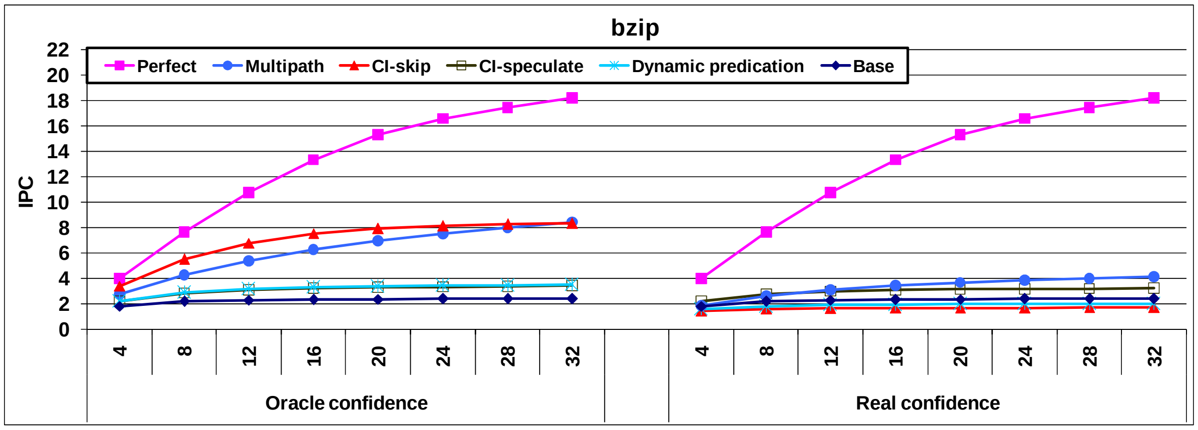


(b)

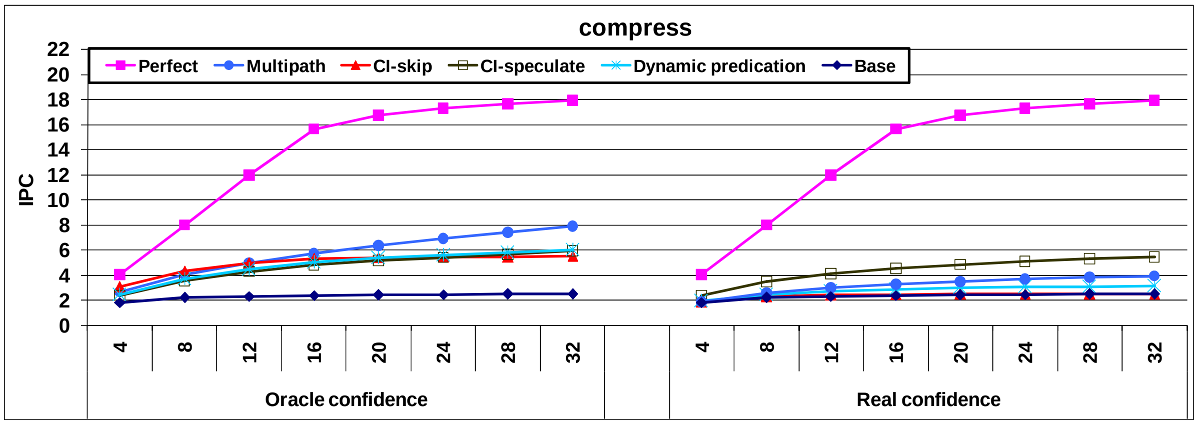
Figure 15. Branch misprediction tolerance techniques with varying issue width.

Figure 16, Figure 17, Figure 18, Figure 19, and Figure 20 present the IPC results for CI-skip, CI-speculate, dynamic predication, and multipath for the individual benchmarks. Several trends can be observed:

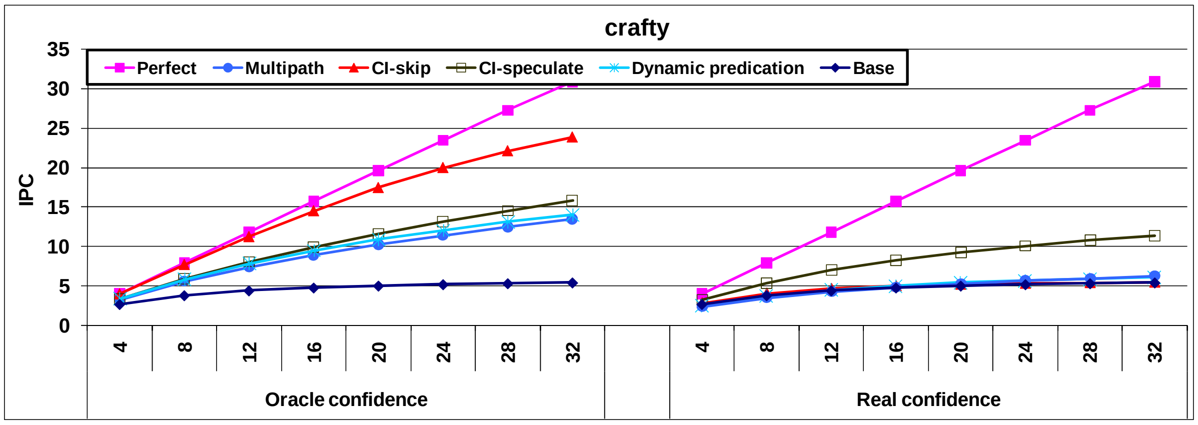
1) CI-speculate outperforms the other techniques when using real confidence, most of the time.
2) Multipath's high branch coverage gives it an advantage with very wide processors. For example, in bzip, li, and perl, multipath approaches or overcomes CI-speculate.
3) Interestingly, CI-skip, CI-speculate, and dynamic predication outperform Perfect in vortex with oracle confidence. In addition, CI-speculate outperforms Perfect with real confidence and an issue width of 32. Vortex has a low branch misprediction rate and the reason for this advantage is that CI-skip, CI-speculate, and dynamic predication allow the processor to open up the window quicker than sequential fetch. By taking the shorter path of a branch region, the advantage of exposing the future instructions outweighs the penalty of the branch misprediction, in this benchmark.



(a)

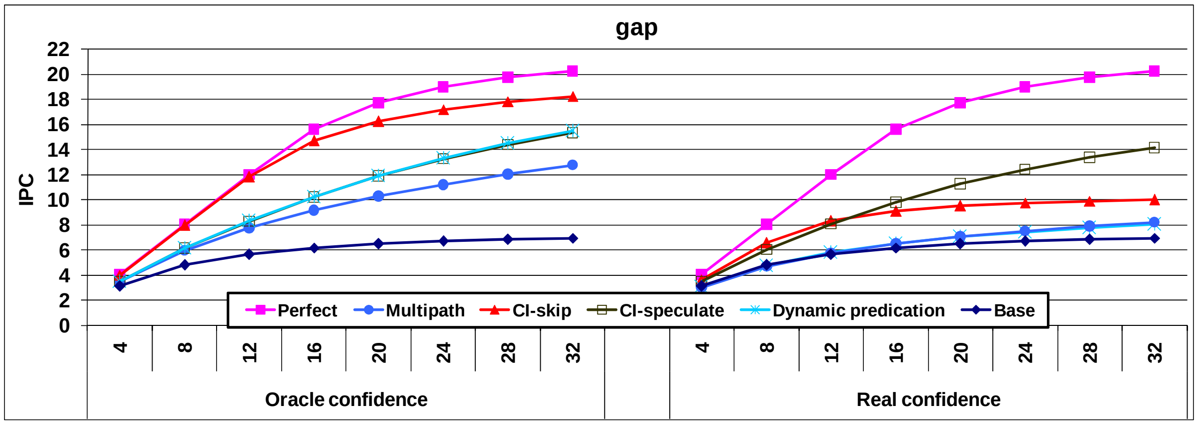


(b)

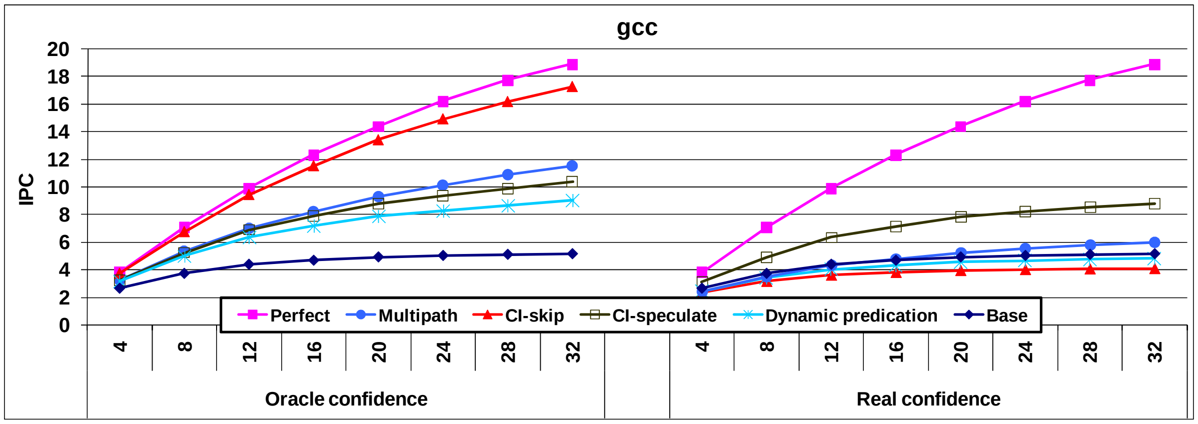


(c)

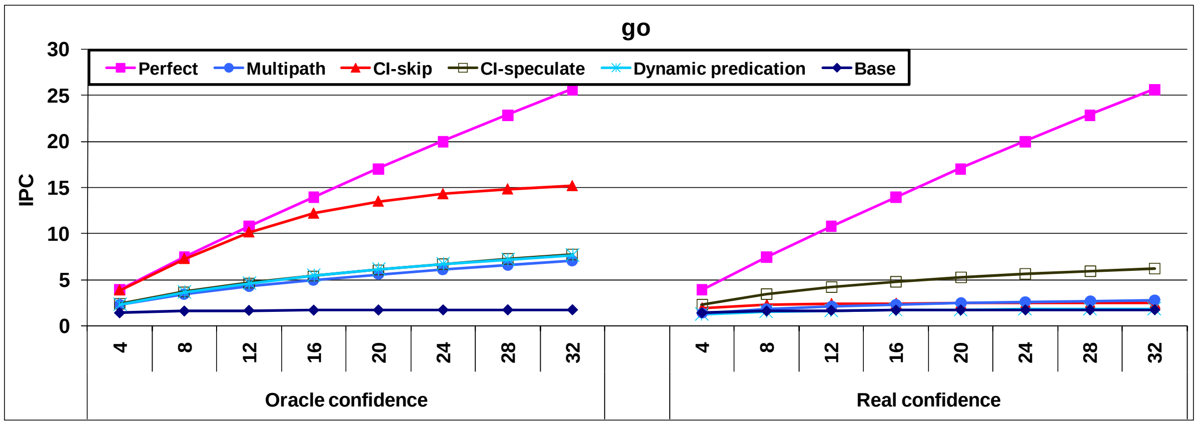
Figure 16. Branch misprediction tolerance techniques with varying issue width (individual benchmarks).



(a)

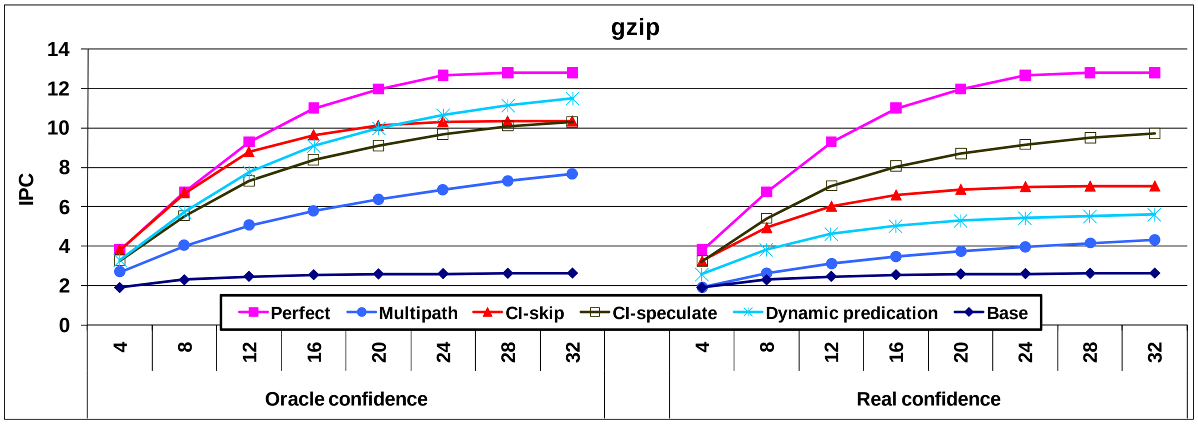


(b)

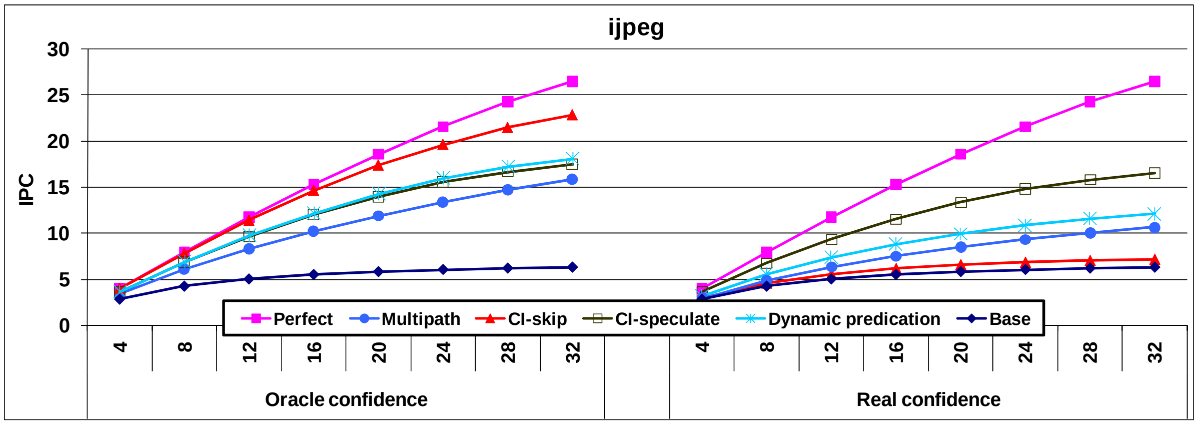


(c)

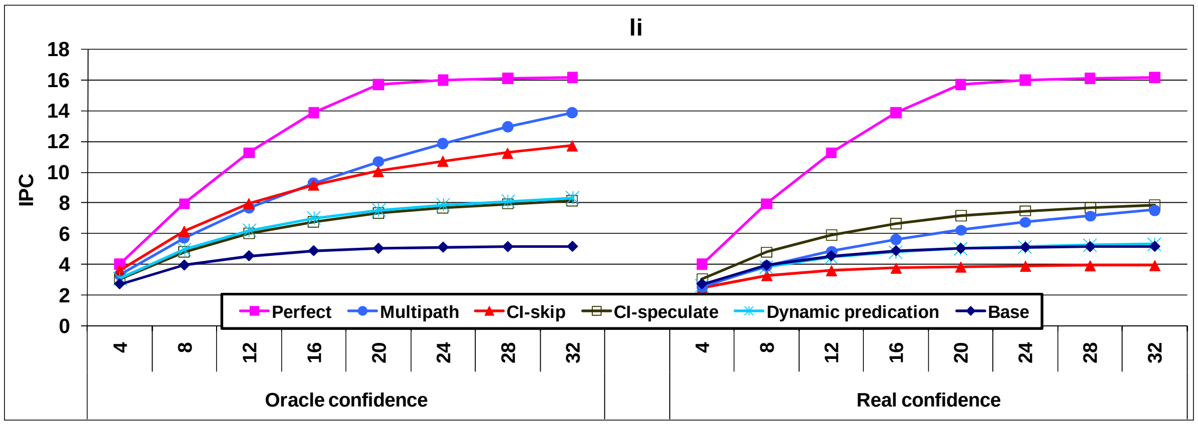
Figure 17. Branch misprediction tolerance techniques with varying issue width (individual benchmarks).



(a)

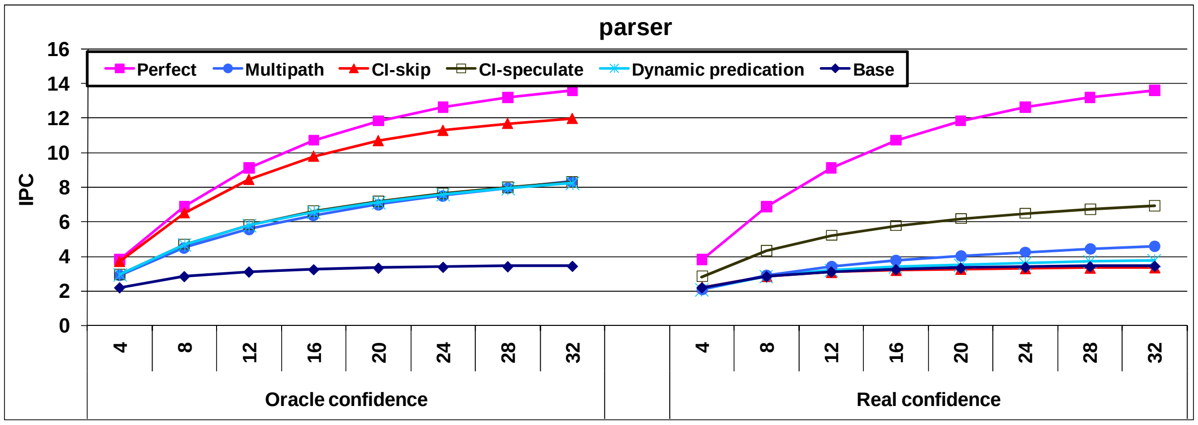


(b)

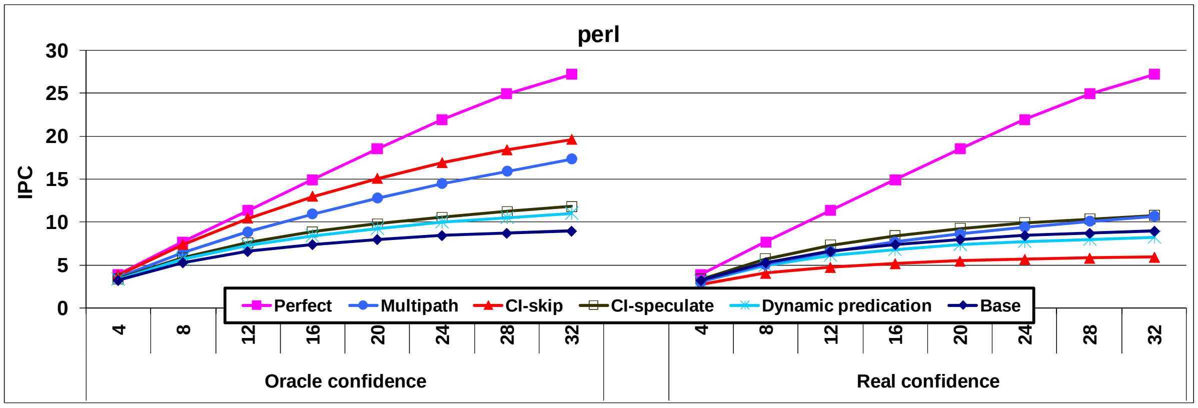


(c)

Figure 18. Branch misprediction tolerance techniques with varying issue width (individual benchmarks).

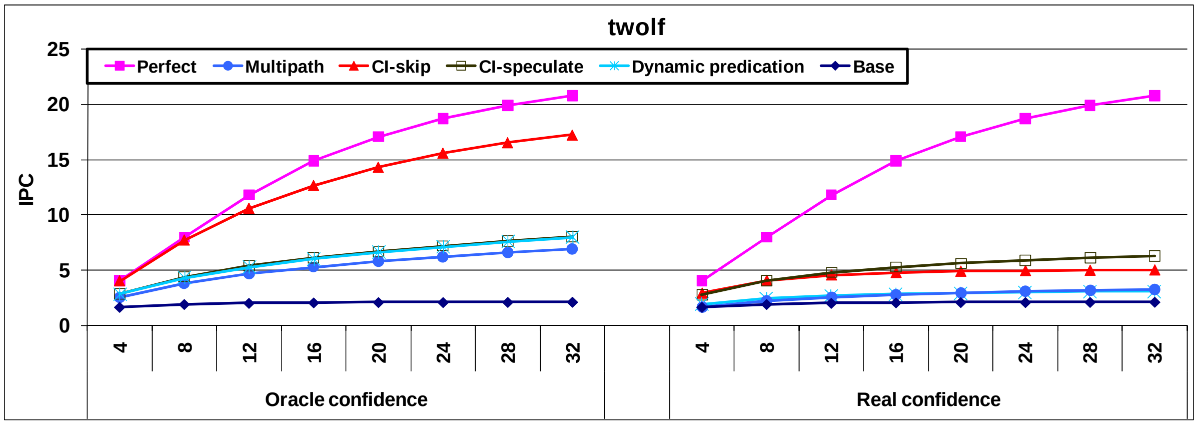


(a)

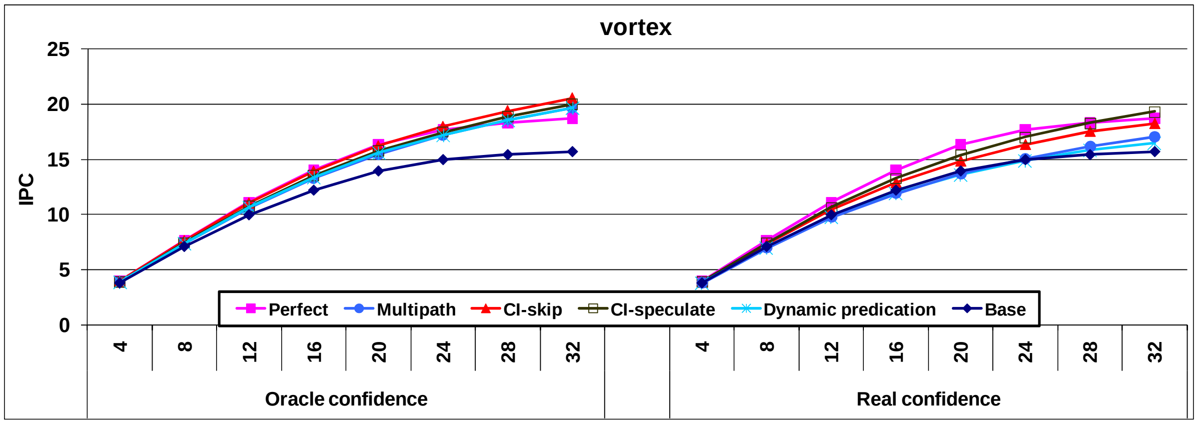


(b)

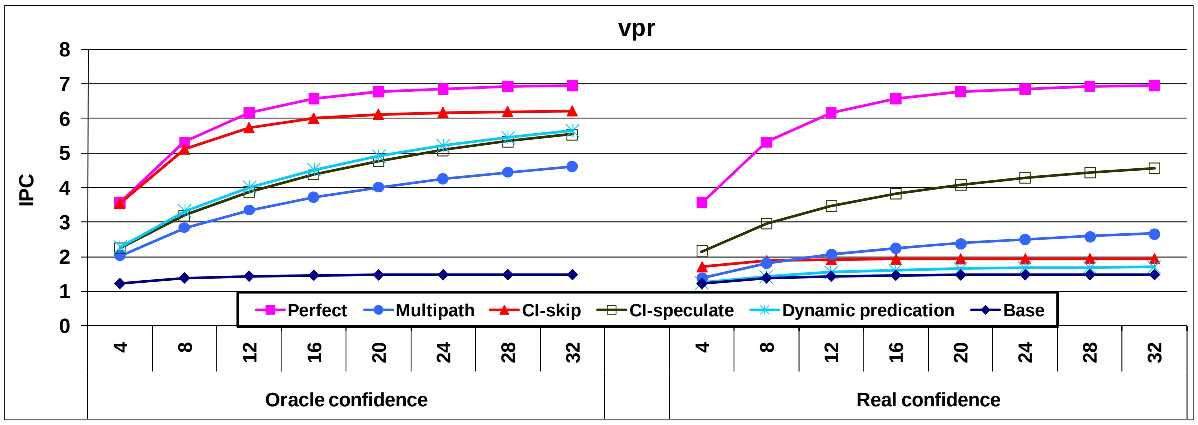
Figure 19. Branch misprediction tolerance techniques with varying issue width (individual benchmarks).



(a)



(b)



(c)

Figure 20. Branch misprediction tolerance techniques with varying issue width (individual benchmarks).

Chapter 4

Control independence support mechanisms

Covering a branch with control independence has the advantage of reducing the branch misprediction penalty. However, to successfully cover a branch, we need to identify its control independent (CI) instructions accurately. This requires us to accurately know the branch's reconvergent point. In addition, we need to be able to detect and track reconvergent points in the dynamic instruction stream efficiently. Finally, we need an accurate account of all CIDD instructions influenced by the branch outcome.

To maximize the performance of our system, we need to leverage control independence on as many branch mispredictions as possible. However, achieving high branch misprediction coverage is difficult and requires dealing with complex and unstructured control-flow. For example, some branch mispredictions involve complex control-flow, such as nested branches and recursive functions, which may lead to incorrect reconvergent points or the inability to detect the reconvergent points. Therefore, the recruited mechanisms must be robust: resilient to all control-flow constructs, able to recover from incorrect reconvergence information, and, meanwhile, flexible enough to adapt to the available resources and optimize system performance.

4.1 Reconvergent point

A reconvergent point is an instruction that post-dominates a branch. This requirement is necessary to consistently distinguish between instructions before and after the reconvergent point. If a post-dominating point is not selected, some control-flow paths in the branch region may miss the reconvergent point. Hence, we cannot leverage control independence on this branch misprediction because we cannot distinguish the control-dependent from the control-independent instructions. A branch's reconvergent point can be generated using a compiler, simple control-flow heuristics, or a hardware predictor.

4.1.1 Compiler

The compiler needs to provide the reconvergent point of each branch and convey this information to the processor. For example, gcc's existing post-dominator analysis can be used to locate reconvergent points. The generated reconvergent points are conservative since the compiler considers all possible control-flow paths between the branch and its reconvergent point, including rarely traversed paths.

Once all the reconvergent points have been identified, a mechanism is required to convey this information to the processor. One solution is to encode this information into the original binary. The ISA would provide optional support for the compiler to specify the immediate reconvergent point of a branch. For 64-bit instruction encodings or variable-length instruction encodings, it may be feasible to add a pc-relative offset to encode the reconvergent PC of a branch. Otherwise, a new instruction is needed for conveying reconvergent PCs and would immediately precede corresponding branch instructions. Both forms could be supported, an offset for most reconvergent PCs and an instruction for reconvergent PCs that exceed the offset. Reconvergent PCs do not effect the program's function (branches with no reconvergent points would leverage conventional (full) branch recovery), so branches with reconvergent points that cannot be encoded or branches with unknown reconvergent points can be safely excluded.

4.1.2 Heuristics

A simple approach for generating reconvergent point information is to leverage imprecise heuristics. Heuristics exploit common control-flow constructs that tend to have obvious post-dominators. Using heuristics, we can avoid using the compiler and modifying the binary to pass the reconvergent point information to the processor. In essence, heuristics sacrifice branch misprediction coverage for simplicity and the ability to cover programs without the need for recompiling them.

- *Return heuristic*: Functions tend to isolate all control-flow within them except in extraordinary circumstances such as long-jump instructions and exit conditions. Hence, the return heuristic builds on the fact that a return of a function tends to be a post-dominator for all instructions within the function and can act as a valid reconvergent point (Rotenberg, et al., 1999).
- *Loop heuristic*: In structured code, the loop exit tends to be a post-dominating point for all instructions within the body of the loop. The loop exit is often identified by a backward branch. The loop heuristic is not precise, as some backward branches may not correspond to loop exits (Rotenberg, et al., 1999).
- *Hammock heuristic*: By observing the if-then construct in structured code, we notice that the branch's target points to a post-dominating point for all instructions in the if-then construct body. Hence, the branch's target can be used as the branch's reconvergent point (Gandhi, et al., 2004) (Rotenberg, et al., 1999).

4.1.3 Dynamic reconvergence predictor

The dynamic reconvergence predictor proposed by Collins et al. (Collins, et al., 2004) can also be leveraged for providing the reconvergence information needed for control independence. The hardware reconvergence predictor combines the advantages of both the compiler approach and the heuristic approach. It is able to achieve high branch misprediction coverage while avoiding the need to recompile the targeted programs. The reconvergence predictor categorizes branches into four categories based on the location of the reconvergent point with respect to the branch and the control-flow leading to it (Collins, et al., 2004).

- *Reconverge below max:* This category includes branches with their reconvergent points below them. This is the most common type of branch category. Instructions below the reconvergent point and at the same call depth can never be fetched between the branch and its reconvergent point. An example of a branch in this category is a simple forward hammock with no embedded branches or the backward branch in a "for loop".
- *Reconverge above max:* Branches with their reconvergent points above them are considered within this category. Instructions between the reconvergent point and the branch in the same call depth can only be fetched after the reconvergent point.
- *Rebound reconverge:* Branches that have their reconvergent points below them but are not part of the reconverge below max category are part of this category. These branches were not part of the reconverge below max category because some instructions from below the reconvergent point are fetched between the branch and its reconvergent point. This can occur because of some control-flow after the branch that

pass the reconvergent point and then branch backward to the reconvergent point. This situation is commonly associated with switch-case construct.

- *Return reconverge:* Branches in this category reach one or more return points (with the same call depth) before reaching a common reconvergent point.

The reconvergence predictor continuously monitors the retired instruction stream and tries to detect changes to the current reconvergent points. The predictor can monitor a limited number of branches at any given time. These active branches are located in the Active Reconvergence Table (ART). Each monitored branch can potentially fall within any of the four branch categories; hence, the predictor maintains four reconvergent points for each branch. When the reconvergent point for a given category is seen, that category is deactivated. If a retired PC violates the assumptions of an active category, we update its reconvergent PC with the newly retired instruction's PC. For example, in the reconverge below max category, if we see a PC below the reconvergent point and with the same call depth before reconverging, we detect that the old reconvergent point is incorrect and we update our entry with the new reconvergent point. When all categories are inactive, the branch leaves the ART and updates the Reconvergence Prediction Table (RPT). The RPT is indexed by the branch's PC. If a branch hits in the RPT, the most likely reconvergent point is selected from the four reconvergence categories.

In this thesis, the predictor is augmented to provide additional information. Confidence counters separate accurate from inaccurate predictions. The confidence counter of a given branch is incremented whenever an instance of the branch and its reconvergent point retires without causing a change to the predictor's state. If the retired reconvergent point invalidates the previously stored reconvergent points then we update the entry and reset the confidence counter. In addition, for each branch, the predictor keeps track of the maximum path length through the branch's control-dependent (CD) region, among paths that were traversed. This information is useful for guiding when to apply control independence. We select a maximum CD path length above which it is not worthwhile to exploit control independence due to the sheer number of incorrect control-dependent instructions.

4.1.4 Performance impact of reconvergent point selection

The distance of the reconvergent point from its branch affects the performance potential of control independence. The closer the reconvergent point is to the branch, the less work that is squashed and wasted (incorrect CD instructions) when a branch misprediction occurs and the more the potential for saving CIDI instructions.

True reconvergent points, which are valid on all control-flow paths, may not yield the highest performance. By identifying that some control-flow paths are infrequently traversed, speculative reconvergent points emerge with higher performance potential. For example, Figure 21 shows a branch with two possible control-flow paths. However, the right path encompasses a branch that transfers control outside the main branch's two paths. Hence, the true reconvergent point is located at the target of the infrequent control-flow path. This control-flow construct is common in branches that contain early loop exits and in error checking code where the branch ends the program if an error condition occurs.

If we ignore the infrequent control-flow path, the speculative reconvergent point is located at the intersection of the main branch's two control-flow paths, which is much closer to the branch. However, in the case we traverse the infrequent control-flow path, we can no longer take advantage of control-independence to reduce the penalty of the misprediction on the main branch and must revert to squash-based branch misprediction recovery. The tradeoff between selecting speculative versus true reconvergent points must be balanced to achieve the highest performance gains.

Moreover, considering CI-skip as a possible sequencing model changes tradeoffs with respect to reconvergent point selection. With CI-skip, we no longer have to worry about wasted work done on CD instructions. The CD path is delayed until the branch resolves. However, the effect of reconvergent point selection on performance depends mainly on the quality of the CI region. A CI region with a high fraction of CIDI instructions is desirable. Since the number of CIDI instructions depends on the nature of the data dependencies and how much of them are influenced by the productions in the branch's CD region, it is difficult to evaluate if selecting a given reconvergent point over another is beneficial. Quantifying this tradeoff is involved and optimizing performance requires finding points where the number of CIDI instructions is maximized.

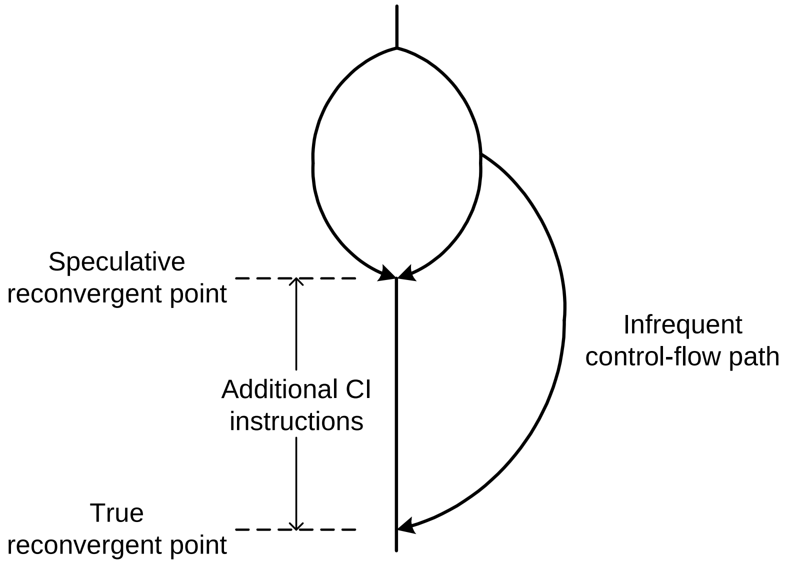


Figure 21. Example of true vs. speculative reconvergent points.

4.2 Control-Flow Stack (CFS)

The previous section discussed ways to associate a reconvergent point with each branch. Using this information, we need a mechanism to detect when a branch has reconverged, during instruction fetch. Detecting reconvergence of multiple concurrent and nested branches is a challenge that needs to be addressed efficiently to achieve high branch coverage with the nearest reconvergent point (highest performing).

The *control-flow stack* (CFS) is a hardware mechanism, which enables robust and accurate detection of reconvergent points in the fetch stream.

4.2.1 Single branch

We will address how reconvergence is detected for the simplest control-flow construct, a single branch that does not encompass any branches or calls.

To detect the reconvergence of a single branch, a stack with depth one is sufficient. When a branch is dispatched, its reconvergent PC is pushed onto the CFS top-of-stack. The reconvergent point in the dynamic instruction stream is detected by comparing the PCs of newly dispatched instructions to the reconvergent PC at the top-of-stack. If there is a match, then the branch corresponding to the current top-of-stack has reconverged and we pop the top-of-stack entry. This frees the stack to be used by other branches and marks the beginning of the control-independent instructions. However, if the branch corresponding to the top-of-stack completes before reconvergence is detected, then detecting reconvergence is no longer necessary and we can pop the top-of-stack for future branches to utilize it.

Therefore, the CFS top-of-stack can be freed using either of two criteria:

1) Popping the top-of-stack when reconvergence is detected.

2) Popping the top-of-stack when the corresponding branch completes.

4.2.2 Nested branches

To support multiple nested branches concurrently, the mechanism employs a multiple-entry stack. When we see a new branch, we push its reconvergent PC on to the stack making it the new top-of-stack. Assuming correct reconvergent points, we are assured to see the reconvergent PC in the top-of-stack entry before any reconvergent PCs in other stack entries. This is true because all reconvergent points are, by definition, immediate post-dominating points. Like in the single branch scenario, the top-of-stack is popped when its reconvergent PC is detected. This exposes the next stack entry as the new top-of-stack and its reconvergent PC as the new monitoring candidate.

Branches that execute before reconverging no longer need their stack entries. Unlike the single branch scenario, the stack entry being removed may not always be located at the top of the stack. Removing stack entries in the middle of the stack may violate normal stack semantics that only support pushing and popping. To address this problem, two implementations are possible:

1) The first solution delays removing entries from the middle of the stack until popping is possible. When a branch completes, its corresponding stack entry is marked as invalid. If an invalid entry becomes the top-of-stack or the bottom-of-stack, the invalid entry is removed using normal popping. This solution maintains the simple semantics of the stack at the expense of delaying the freeing of stack entries. There is no extra cost associated with popping multiple entries at either end of the stack,

because this simply involves moving the head or tail pointer to the next active entry in the stack.

2) If immediate stack entry removal is desired, a stack implementation that can collapse away entries is necessary. This implementation is more complex but yields higher utilization of the stack. Invalid entries need not occupy stack entries for a long time.

The size of the stack dictates the number of nested branches (case 1 above) or nested unresolved branches (case 2 above) that can be monitored concurrently. If the stack is full, we no longer can detect reconvergence for new branches. Branches with no stack entry are forced to "give up" their reconvergent points. However, branches with no stack entry can inherit the reconvergent point of the closest encompassing branch, located on the top-of-stack. This is correct because the reconvergent points of encompassing branches satisfy the criteria of being post-dominators for inner branches. Reconvergent point sharing is discussed in the more detail in the next section.

4.2.3 Reconvergent point sharing and CFS merging

When a branch is dispatched, we normally push its reconvergent point on the CFS. However, some branches may not have reconvergent points. This can occur either because the compiler or reconvergence predictor failed to provide a valid reconvergent point, or because the branch was forced to give up its reconvergent point because the CFS is full. These branches can still benefit from control-independence by inheriting the encompassing branch's reconvergent point. The closest active (unresolved) encompassing branch can be found on the CFS top-of-stack. The branch simply inherits the encompassing branch's reconvergent point. In addition, some branches may find that the CFS top-of-stack already has the same reconvergent point, anyway.

Some branches may have the same static reconvergent point. Programming constructs such as the “switch” statement contains many branches that converge at the end of the switch statement. More generally, branches may share dynamic reconvergent points. There are many cases in which multiple dynamic branches share the same reconvergent point. A common example is the multiple instances of the backward branch of a loop where they all have the loop exit as a reconvergent point. Fortunately, the CFS can easily detect cases in which multiple branches have the same dynamic reconvergent point.

If the newly dispatched branch does not have a reconvergent point, or if its reconvergent PC matches the reconvergent PC at the CFS top-of-stack, or if the CFS is full, then the new branch and the branch corresponding to the CFS top-of-stack will share the reconvergent point of the CFS top-of-stack. In this case, the new branch does not push a new entry onto the CFS, implicitly “merging” with the CFS top-of-stack. Hence, merged branches share the same stack entry. When the reconvergent point corresponding to the top-of-stack is dispatched, we pop the top-of-stack and detect reconvergence for all merged branches at once.

If all merged branches complete before detecting their shared reconvergent point, then the shared CFS entry can be freed. The CFS accomplishes this by maintaining a branch merge counter for each CFS entry. When a newly fetched branch merges with the CFS top-of-stack, the branch merge counter for that entry is incremented. When a branch completes, the corresponding branch merge counter is decremented (assuming the reconvergent point has not been detected yet). If the branch merge counter reaches zero before detecting the reconvergent point, then the CFS entry can be freed.

4.2.4 Recursion

Correct functioning of the CFS is based on the fact that, when trying to detect reconvergence of a specific branch, the first occurrence of its reconvergent PC is the correct reconvergent point that post-dominates the branch. However, due to recursion, the CFS may encounter a reconvergent PC that matches the top-of-stack mistakenly. This reconvergent PC is actually a different dynamic instance of the reconvergent PC that does not post-dominate the initiating branch.

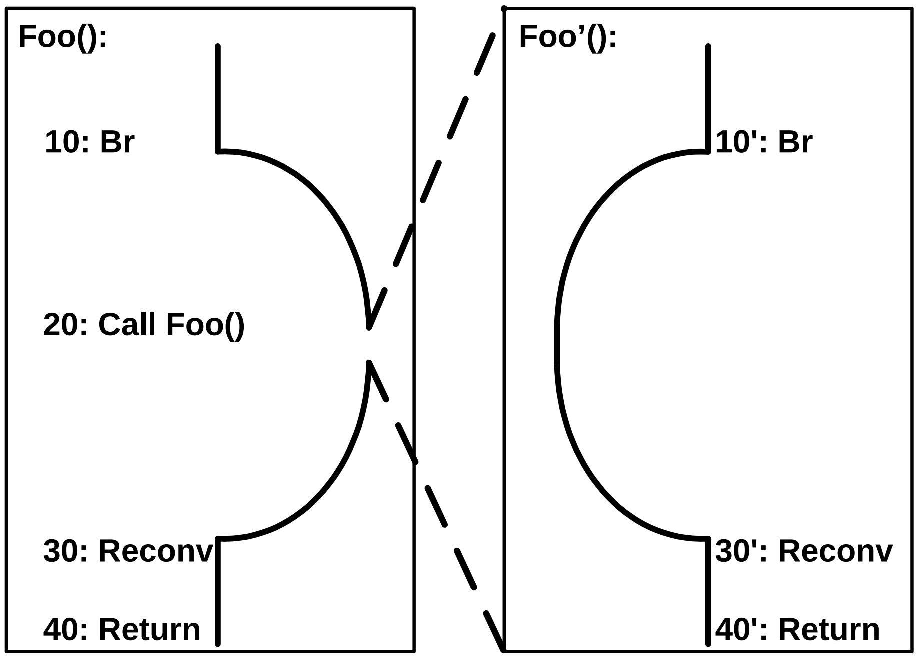


Figure 22. Function "foo" recursively called.

To illustrate this problem, Figure 22 shows a dynamic sequence of instructions where function "foo" is called recursively. Instructions are shown with their PCs. The reconvergent point of the branch at PC 10 is located at PC 30. If we follow the instruction sequence in the example initially assuming an empty CFS, the branch “10: Br” will push the reconvergent PC 30 onto the stack. Next, we will encounter the call instruction which will take us to a new instance of the “foo” function labeled “foo’”. The branch “10’: Br” will try to push its reconvergent PC onto the CFS, only to merge since it also has 30 as its reconvergent PC. This is not a problem itself, since the reconvergent PC of the first branch post-dominates the second branch. Next, we encounter instruction “30’: Reconv” and are forced to pop the stack prematurely since 30 matches the CFS top-of-stack. However, this reconvergent point does not satisfy the criterion of being a post-dominator of the outer branch “10: Br” and hence is an incorrect reconvergent point.

If instruction “10: Br” happens to be a branch misprediction, the incorrect reconvergent point may cause us to corrupt program state during recovery. As we discard the incorrect CD path, some incorrect instructions will persist by mistake. Instructions between PC 30’ (the perceived reconvergent PC) and PC 30 should have been discarded. However, because of premature reconvergence, these instructions remain, causing incorrect program behavior.

To address this problem, we make the reconvergence test definitive by tracking call depth in the dispatch stage and including call depths in CFS entries. In other words, if the new branch’s reconvergent PC and call depth match the CFS top-of-stack, then the branches have the same dynamic reconvergent point. Otherwise, these reconvergent points are different although they share the same PC. Therefore, the reconvergent point is defined by both the reconvergent PC and the call depth of the reconvergent instruction. This distinguishes the two reconvergent PCs and, hence, prevents premature reconvergence. From the previous example, PC 30 will not match PC 30’ because of the difference in call depth.

4.2.5 CFS violation detection and recovery

As a consequence of using heuristics or dynamic hardware predictors to predict reconvergent points, an incorrect reconvergent point may be predicted. An incorrect reconvergent point does not post-dominate the branch it belongs to; hence, this branch cannot be covered by control independence. The incorrect reconvergent point also affects all encompassing branches (lower stack entries), by preventing them from detecting their reconvergent points. This is a consequence of the incorrect reconvergent point tying up its CFS entry until its branch resolves. This degrades performance, as we lose opportunity to employ control independence on the branch with the incorrect reconvergent point and for the encompassing branches below it in the CFS.

It is crucial to detect incorrect reconvergent points promptly to minimize their negative performance effects. There are three symptoms of incorrect reconvergent points that can be used to identify them.

1) Leaving branch frame: By comparing the CFS top-of-stack call depth with the currently dispatched instruction’s call depth, we can detect when we leave the frame of the reconvergent point. This is detected when a call depth lower than that of the CFS top-of-stack has been encountered. Since we define a reconvergent point to be the pair of PC and call depth, then a reconvergent point must be in the same function (frame) as that of the branch. Otherwise, the call depth portion of the reconvergent point will never be satisfied (except in another instance of the function, which is wrong). Since there is no chance to detect the reconvergent point at this time, it fails the branch post-

dominating test. So, the concerned CFS entry must contain an incorrect reconvergent point.

2) Encompassing branch reconvergence: If an encompassing branch detects its reconvergent point before the CFS top-of-stack branch does, then this implies that the CFS top-of-stack contains either an incorrect reconvergent point or sub-optimal reconvergent point. In either case, it is far more valuable to use the encompassing branch's reconvergent point instead. To detect this scenario, the CFS needs to be modified. Instead of only comparing the CFS top-of-stack to newly dispatched instructions for a match, we need to compare all CFS entries in parallel. This is feasible for CFSs with relatively few entries, but may have power implications if the number of entries is large. Fortunately, the CFS size is a function of the maximum number of unresolved nested branches covered by control independence (which tends to be small for most benchmarks).

3) Exceeding region size: Leveraging the provided maximum region size of a branch region, we can detect anomalies. One possible reason for exceeding the maximum region size is having an incorrect reconvergent point. Although this test does not conclusively say that a reconvergent point is incorrect, it does give a strong indication of problem. To implement this check, we would need to add a counter to each CFS entry. Incrementing is done when instructions dispatch. When any counter exceeds the maximum region size, we can flag a possible incorrect reconvergent point.

If the violation is detected on the first pass before servicing a branch misprediction, then we can repair the CFS. Two repair policies are possible. First, we could flush the whole stack, forcing all branches on the stack to use squash-based branch recovery (since no reconvergence will be detected). The second approach is higher performing and attempts to merge the violating entries with the next available non-violating entry in the hope that that entry will reconverge correctly and some control independence benefit will be preserved. If no entries are available to merge with, then the violating entry is removed and conventional (full) recovery is used for the concerned branch.

If the violation is detected during branch misprediction recovery, then we must forego selective recovery and fall back to conventional (full) recovery (simply do not reconverge and discard all CI instructions).

4.2.6 Additional CFS functions

The CFS is an essential mechanism for control independence that enables us to cope with complex control-flow constructs simply. The usefulness of the CFS can go beyond its main function of detecting reconvergent points. The CFS can also be used to propagate control dependence vectors if desired. The control dependence vector identifies all branches that an instruction depends on from a control-flow standpoint. These vectors are computed by setting all the bits corresponding to branches currently on the stack. The control dependence vectors are useful in some control independence implementations that require selective squashing of the CD instructions (note that TCI does not require this). When the CD instructions of a given branch need to be squashed, the bit corresponding to the branch is asserted and instructions that have that bit set in their vectors are squashed.

Additionally, the CFS can detect looping behavior in general through its merging ability. When branches merge their reconvergent points on the CFS, this is a sign of a possible loop.

The loop count can be estimated by the number of reconvergent point merges divided by the number of unique branch PCs being merged. One advantage of the CFS over using heuristics to detect loops (backward branches) is that it can detect looping behavior even in the absence of a traditional looping construct.

Finally, the CFS can be used as a method to optimize the global history of the branch predictor. By observing that encompassing branches’ outcomes do not change between the different dynamic instances for a given control-dependent branch, we can exclude their history bits from the global history, allowing for longer history lengths. The reasoning for this phenomenon is that, for the concerned branch to be fetched, encompassing branches have to take a given control-flow direction, which is constant. Therefore, the encompassing branches do not add any information toward the outcome of the predicted branch. One concern needs to be addressed for loops that contain no internal branches, while using this optimization; in this case, the loop branch would not see any history bits from previous loop iterations. This can possibly degrade branch prediction accuracy as the predictor would not be able to identify the loop exit. Note that using the CFS to optimize global history was proposed by my colleague, Vimal Reddy.

4.3 Identifying CIDD instructions

Identifying the reconvergent point enables us to mark the beginning of the control-independent (CI) instructions. Furthermore, we need to be able to distinguish between control-independent data-independent (CIDI) and control-independent data-dependent (CIDD) instructions. CIDD instructions are influenced by the outcome of the branch, as their outcomes may change depending on the control-flow path traversed. This indirect dependence on the branch is caused either by register data dependencies or memory data dependencies.

4.3.1 Register dependencies (Influenced Register Set (IRS))

In an out-of-order superscalar processor, there are potentially many versions of a given architectural register at any given time. Instructions have to execute with the correct physical register mapping (source names) to ensure correct program execution. In conventional processors, in-order register renaming ensures that instructions always get the newest physical register mappings in the rename map.

Control independence violates in-order register renaming by preserving CI instructions when a branch misprediction is detected. Hence, the CI instructions are renamed before the correct CD instructions are fetched, making some of their source names potentially stale. Physical register mappings are altered during branch misprediction recovery because of either squashing the incorrect CD path of the branch, which could remove some register productions, or fetching the correct CD path of the branch, which may introduce new register productions. Hence, CI instructions must be repaired to ensure correct program behavior.

CI instructions can be broken down into two groups, CIDI and CIDD instructions. CIDI instructions are not affected by changes in register mappings at all. Their source names stay the same. On the other hand, CIDD instructions have one or more of their source names change during branch misprediction recovery. In addition, instructions dependent on these root CIDD instructions directly or indirectly are also considered CIDD instructions.

Since the cause of register mapping changes for root CIDD instructions is the insertion or removal of register productions in the CD region of the branch, we must identify register productions on all traversed control-flow paths in the CD region. This collection of register productions is called the Influenced Register Set (IRS). Each bit in the IRS corresponds to a single architectural register. Hence, the size of the IRS is bounded by the number of architectural registers specified by the ISA. For example, if the bit corresponding to R5 is set, then any CI instruction consuming a version of R5 from before the reconvergent point, directly or indirectly through a chain of CIDD instructions, is considered CIDD. With the help of the IRS, we have the information necessary to identify CIDD instructions allowing us the opportunity to service them.

The IRS can be easily generated by a compiler or a hardware predictor.

4.3.1.1 Generating the IRS using a compiler

The compiler would collect all register productions between the branch and its reconvergent point statically using basic data-flow analysis. This information is then conveyed to the processor with a new ISA instruction. For example, in the PISA ISA, a new 64-bit instruction specifies the IRS. PISA has 31 integer (excluding register 0), 16 double-precision floating-point, and 3 other (HI, LO, FCC) logical registers. 16 bits are used for the floating-point registers. If any of these are set, FCC is implied to be in the IRS. 1 bit is used for both the HI and LO register. Thus, 48 bits encode the IRS. The IRS instruction is inserted before each branch whose reconvergent PC is specified.

4.3.1.2 Generating the IRS using the reconvergence predictor

Using a hardware predictor to predict the IRS is also feasible. The IRS is highly predictable given a predictable reconvergent point. A learning mechanism is added to the dynamic reconvergence predictor to collect a branch’s IRS. As the predictor monitors retired instructions for reconvergence, it accumulates logical registers being written to after the branch but before reconvergence is detected. Influenced registers are also accumulated across all seen retired paths. This provides a way to identify all productions for a given branch's control-dependent region. The IRS must be accurate. The use of confidence ensures repetition, so that enough different paths are traversed through a branch's CD region to yield a representative IRS.

4.3.1.3 Performance impact of IRS

The performance gain of covering a branch misprediction with control-independence is directly related to the number of CIDI instructions preserved. A high percentage of CIDI instructions in the CI region is desirable and would leave few CIDD instructions needing recovery, improving performance. The number of registers set in the IRS affects the number of CIDD instructions in the CI region.

If all bits in the IRS are set, then all instructions in the CI region would be CIDD instructions and we would have to repair the whole CI region. This is tantamount to conventional (full) branch misprediction recovery. On the other hand, if none of the bits in the IRS are set, then all instructions are CIDI instructions (except for violating loads and their dependents) and the CI region need not be repaired. The number of CIDD instructions is also affected by which registers are set in the IRS and what is the nature of the data dependencies in the targeted CI region. For example, if the stack pointer register (R29 in PISA) is set in the IRS, then the number of CIDD instructions can be large. This is the case because stack pointer arithmetic forms a long serial dependence chain and because the address calculations for many stack loads and stores depend on it.

Therefore, a sparse, optimized IRS is favorable. If we use an overly conservative IRS, the number of perceived CIDD instructions will increase and the number of CIDI instructions will decrease, which will reduce the advantage we have over conventional (full) branch recovery. On the other hand, if we use an overly optimistic IRS, there may be many violations (Section 4.3.1.5) causing frequent downgrades to conventional recovery.

Ideally, the IRS would be sparse without leading to lost coverage. The optimal/actual IRS would only contain register productions on the incorrect control-flow path of the mispredicted branch and register productions on the repaired correct control-flow path. The actual IRS is completely known only after recovering from a branch misprediction at the point of reconvergence. At that point, the processor has examined both the correct CD instructions and incorrect CD instructions. If the IRS is required before recovering from a branch misprediction, then we must rely on a speculative IRS provided by the compiler or the reconvergence predictor. The compiler and the reconvergence predictor can achieve the optimal/actual IRS in some situations where the targeted branch only has two possible paths from the branch to its reconvergent point. However, when branches have more than two control-flow paths leading to the reconvergent point, conservative IRSs will be produced. Regions with multiple control-flow paths occur because of branches with multiple targets such as jump indirect instructions (JALR and JR) or because of internal control-flow (branches, loops, etc.) that increases the number of unique paths from the branch to its reconvergent point.

4.3.1.4 Optimizing the IRS

Due to the limited number of architectural registers, the compiler, during the register allocation phase, is forced to spill registers onto the stack. Spilling to the stack is also used to deal with function calls to avoid caller/callee register conflicts. The compiler has set conventions on how to handle saving registers to allow for correctness across function calls. Caller-saved registers are registers that are saved if the calling function is using these registers and they are its responsibility. Callee-saved registers are registers that the caller assumes will be intact after returning from the called function and can assume their correctness. This pact is guaranteed by the callee function.

Based on the observation that there is typically no net change in callee-saved registers before and after a subroutine, we can optimize the IRS generation criteria to reduce the number of registers set in it. Instead of collecting all register productions between a branch and its reconvergent point, the optimized IRS would only collect register productions in its call depth level between the branch and its reconvergent point (function calls would set their return registers in the IRS). Hence, any productions observed in internal functions (higher call depths) are omitted from the optimized IRS. The optimized IRS is not inherently safe, as some CI instructions may receive the mapping of one of the omitted register productions leading to lost coverage as some CIDD instructions are considered CIDI. To make the optimized IRS safe, we need to isolate the effect of the omitted registers by short-circuiting their mappings with their corresponding mappings from before the branch. This is guaranteed to be correct because the callee function preserves the value of these registers by saving and restoring them. Leveraging the control-flow stack, the branch would checkpoint any register mapping not in the optimized IRS. If the branch resolves before reconverging, then the CFS entry will be freed and the IRS is not needed. On the other hand, if the reconvergent point is reached first, then the top of the CFS loads the saved register mappings into the rename map. This allows the CI instructions to short-circuit the omitted IRS productions safely.

4.3.1.5 IRS violation detection and recovery

The compiler or reconvergence predictor provided IRS may not match the optimal/actual IRS. It can be either too conservative or overly optimistic. If the IRS has additional bits set, it is conservative. In this case, control independence will function correctly, but at an opportunity cost of identifying some CIDI instructions as CIDD. However, if the IRS is too optimistic, then some CIDD instructions needing repair after a branch misprediction will be neglected (they are incorrectly classified as CIDI). This prevents control independence from successfully repairing a branch misprediction. Fortunately, an inadequate IRS can be detected, by comparing the actual IRS (observed by the processor) to the predicted IRS.

The actual IRS is composed of register productions by incorrect and correct CD instructions. After fetching the incorrect CD instructions and detecting the reconvergent point, the predicted IRS is augmented by the actual IRS registers observed thus far. Since no CI instructions have been fetched yet, this allows us to repair the predicted IRS preventing future IRS violations. Later, when a branch misprediction is detected, recovery includes fetching the correct CD instructions until the reconvergent point is reached again. At this point, the actual IRS is compared with amended predicted IRS. If the actual IRS contains registers not present in the amended predicted IRS, then this is a true IRS violation. To recover from an IRS violation, we are forced to fall back to conventional (full) branch misprediction recovery, discarding all CI instructions.

4.3.2 Memory dependencies (load poisoning)

We have seen how a branch misprediction can affect CI instructions indirectly through register dependencies. This effect on CI instructions is further extended through memory dependencies. Specifically, CI load instructions may be affected by CD store instructions. For a load to execute correctly, it must receive the most recent value of the memory address it is loading. A branch misprediction can introduce false store instructions on the incorrect CD path, or delay correct store instructions on the correct CD path, which can influence a CI load's result. Load instructions that are influenced by stores in a given branch region are considered CIDD with respect to that branch. During selective branch misprediction recovery, CIDD loads must be re-executed, like other CIDD instructions. For control independence to be leveraged, it is necessary that CIDD load instructions produce correct results; otherwise, we will lose control independence coverage and need to fall back to conventional (full) recovery. Hence, identifying CIDD loads through accurate memory dependence prediction is crucial for good performance.

Unlike register data dependencies, which are bounded by the number of architectural registers and can be compactly represented and easily predicted, memory dependencies are more dynamic. Fortunately, memory dependencies are somewhat stable (although not as stable as register dependencies) and relationships between loads and stores can be accurately predicted. Traditional memory dependence predictors, such as store-sets (Chrysos, et al., 1998), have been successful in providing accurate results in the context of aggressive load speculation in superscalar processors.

4.3.2.1 Store/branch set predictor

Traditionally, the store-set predictor can only deal with stores currently in the window. However, in the context of control independence architectures, fetching of stores may be delayed until a branch misprediction is serviced. In a control independence architecture with unresolved branch mispredictions, the store-set predictor may decide mistakenly that a given load is not CIDD and allow it to execute as such. This is wrong, because selectively recovering a branch misprediction may introduce some new stores that were not available in the window when the store-set predictor made its prediction. To overcome this problem, the store-set predictor needs to be modified, making it aware of potential stores introduced late, during selective branch misprediction recovery.

Branch mispredictions covered by control independence introduce holes into the instruction stream. A CI load accessing the store-set predictor before all prior branch mispredictions have been resolved, will not observe the complete instruction stream, leading to an incorrect memory dependence prediction. This situation can be identified through the branches themselves. A low-confidence branch and its CD region represent a possible hole in the instruction stream, where store instructions may be removed or inserted. Therefore, low-confidence branches act like proxies for stores that may be removed or inserted due to control-flow changes.

In addition to the normal store-to-load memory dependency (store set) tracked by memory dependence predictors, the modified predictor will need to track dependencies between low-confidence branches and loads (branch set). The store/branch set predictor now has the capability to detect possible holes in the instruction stream (that typically influence a given load) and delay the final execution of influenced loads until the low-confidence branches have been resolved.

The CD region, a possible hole in the instruction stream, can be identified by either the branch or its reconvergent point. In the store/branch set predictor, using the reconvergent PC is more beneficial than using the branch PC for three reasons:

1) The reconvergent point can represent multiple branches (CFS merging). This reduces the number of points needed to be tracked by the predictor.
2) The reconvergent point is located at the point of separation between the CD instructions and CI instructions. When CI loads access the memory dependence predictor, the reconvergent point will act as a barrier between the possibly incorrect CD instructions and correct CI instructions.
3) If a branch misprediction is being serviced, the branch may retire before servicing is completed. By using the reconvergent point, we prevent the load from accessing the partially repaired CD region.

4.3.2.2 Load violation detection and recovery

As a consequence of load speculation, load violations may occur. In the context of conventional superscalar processors with conventional (full) branch misprediction recovery, load violations are detected by broadcasting the addresses of stores as they execute to younger loads. If a younger load received its value from a store older than the broadcasting store or from the cache, and its address matches the broadcasted store address, then a load violation is detected.

In the context of selective branch misprediction recovery, load violation detection needs to also address the possibility of store instruction removal and re-execution. So, in addition to broadcasting new store addresses to younger loads, we also need to broadcast removed store addresses to younger loads. In control independence architectures, store instructions can cause load violations by inserting a new address, removing an old address, or changing an address. Here is a list of the events causing load violations and the actions required to detect the violations:

1) Out-of-order execution of a store instruction or late insertion of correct CD store instruction: broadcast new address.
2) Removing a CD store instruction: broadcast old address.
3) Re-executing a store instruction: If the address changes, then broadcast both old and new addresses. If only the value changes, then just rebroadcast the old address.

Once a load violation has been detected, recovery is required to achieve correct program behavior. Recovery can be done by either redoing all the work after the load violation, or attempting to selectively repair it. In control independence architectures, the number of load violations can increase significantly, due to the holes in the dynamic instruction stream, when compared to processors implementing conventional (full) branch misprediction recovery. Fortunately, the store/branch set predictor can identify CIDD loads accurately, which reduces the number of load violations introduced by the holes in the dynamic instruction stream.

Therefore, redoing all work after a load violation may be a viable alternative with an accurate store/branch set predictor.

Chapter 5

Control independence: Analysis of implementation aspects and performance factors

Control independence architectures need to accomplish two tasks to recover from a branch misprediction successfully. First, they must repair the CD region of the mispredicted branch. This involves discarding incorrect instructions on the mispredicted control-flow path and replacing them with the correct instructions from the branch's actual control-flow path. Second, they must repair the CI region of the mispredicted branch. The CI region is composed of CIDI and CIDD instructions. CIDI instructions are correct and need not be repaired. CIDD instructions indirectly depend on the branch outcome through data dependencies and need to be repaired by correcting their source values and then re-executing them.

This chapter analyzes implementation aspects and performance factors of repairing the CD region (Section 5.1) and the CI region (Section 5.2).

5.1 Repairing the CD region

To implement control independence, the processor must be able to repair the CD region. This involves two steps. The first step is to discard the incorrect CD instructions from the middle of the window. This entails reclaiming the resources they hold and making them available to future instructions. The second step is to fetch the correct CD instructions and insert them into the middle of the window. This requires allocating the newly fetched instructions resources, so they can execute. These two steps are not fully supported by traditional superscalar processors.

Instructions can hold many types of resources that include: reorder buffer (ROB) entries, load/store queue (LSQ) entries, issue queue (IQ) entries, branch checkpoints, and physical registers. Traditional processors manage (allocate and reclaim) these resources in different ways, some compatible with control independence while others are not. These resources can be grouped based on the way they are managed, into *unordered resources* and *ordered resources*.

5.1.1 Unordered resources

Unordered resources do not maintain order between allocated entries. Entries can be allocated in any order and reclaimed in any order. Processors use *unordered resources* when the order between allocated entries does not have to obey a specific order and when entries must be allocated and/or reclaimed out-of-order. The flexible nature of *unordered resources* makes them compatible with control independence's requirement to insert instructions (allocate resources) in the middle of the window and remove instructions (reclaim resources) from the middle of the window. In addition, *unordered resources'* ability to reclaim resources quickly, out of program order, makes them suitable to use with performance-critical resources. Branch checkpoints and issue queue entries are examples of performance-critical *unordered resources*.

In traditional superscalar processors, instructions get allocated IQ entries at the dispatch stage in program order. The IQ entries are reclaimed out-of-order when their instructions issue. When a processor detects a branch misprediction, some speculative instructions need to be discarded and their resources freed. IQ entries can be reclaimed from misspeculated instructions in two ways:

1) Wait for the misspeculated instructions to issue as normal and free their IQ entries.
2) Identify the misspeculated instructions in the IQ and free their entries. This can be accomplished by walking the ROB and using the IQ entry numbers stored in it, or by using control dependence vectors to free the IQ entries in bulk (Section 4.2.6).

Branch checkpoints are allocated to branches to recover from possible mispredictions. Each branch receives a checkpoint at the dispatch stage in program order. When a branch executes (out-of-order) and a misprediction is not detected, the branch can safely free its checkpoint. Branch checkpoints are performance-critical resources and should be freed as soon as possible. This mechanism stays the same with control independence and no modifications are needed.

5.1.2 Ordered resources

Ordered resources maintain a specific order between allocated entries. Entries usually can only be allocated and reclaimed at either end of the ordered list. Processors use *ordered resources* when program order needs to be maintained between the allocated entries, for correct functionality and/or performance. An additional advantage of an *ordered resource* is its ability to efficiently allocate and reclaim contiguous resources in bulk at either end of the ordered list. The strict order required by *ordered resources* makes them incompatible with control independence's requirement to insert instructions (allocate resources) in the middle of the window and remove instructions (reclaim resources) from the middle of the window. The ROB, LSQ, and RF are examples of *ordered resources*.

The ROB requires its entries to be ordered so that it can retire instructions in the correct program order. The LSQ requires order to correctly enforce memory data dependencies between stores and loads in the window. The LSQ also requires order to commit stores in program order to the cache.

The ROB and load/store queues are circular FIFO structures. Instructions are allocated entries at the end of the FIFOs by incrementing the tail pointers. Entries are reclaimed in two ways, depending on if the instructions are correct or misspeculated. Correct instructions free their entries at retirement by incrementing the head pointer. Misspeculated instructions logically after a mispredicted branch free their entries in bulk by moving the tail pointer to the mispredicted branch's entry. Notice that allocation and reclamation happen at the head and tail of the FIFOs. FIFOs do not support arbitrary insertion and removal from the middle, which makes these resources incompatible with control independence in their current form.

To make the ROB and LSQ compatible with control independence, modifications need to be adopted to enable insertion and removal in the middle of the window. Four possible solutions can be adopted:

1) *Collapsing/expanding buffer*: By replacing the simple FIFO with a collapsing/expanding buffer, we can insert and remove instructions in the middle of the FIFO. The incorrect CD instructions are removed (collapsed away) by shifting the CI instructions in their place. Inserting the correct CD instructions requires expanding the buffer in the middle, between the branch and its CI instructions, to make space for the new instructions. The collapsing/expanding buffer is complex and power hungry. This implementation may also degrade performance by delaying servicing of a branch

misprediction. The buffer cannot collapse/expand entries in bulk and may need many cycles to complete the needed shifting of instructions. This delay increases the branch misprediction penalty.

2) *Linked-list*: The ROB and the load/store queues can be managed as linked-lists of instructions, segments, or processing elements (Rotenberg, et al., 1999), instead of a circular FIFO, at the cost of extra complexity. This implementation is compatible with arbitrary insertion and removal of entries, making it a good match for control independence. However, a problem with linked-list implementations of the ROB and LSQ is the difficulty in sequencing, allocating, and freeing multiple sequential entries in parallel. With a FIFO, it is very simple to access sequential elements in parallel because the indices of a FIFO can be pre-computed from the head pointer. Conversely, the linked-list can only sequence a single entry at a time because the index of the next entry is stored with the previous entry and cannot be pre-computed. The use of multiple next pointers (example: next-pointer 1, next-pointer 2, etc.) in each entry can assist in this effort, at the expense of added management complexity and storage requirements. The use of coarse-grained linked-lists, such as segmented linked-lists, can also mitigate the problem, at the expense of internal fragmentation.

3) *Pre-allocation and delayed reclamation of resources*: A less intrusive approach that allows the use of a FIFO is pre-allocation and delayed reclamation of resources. This solution avoids the complexity of linked-list designs and the latency for expanding a traditional FIFO. To allow easy insertion of CD instructions, one can estimate the maximum number of resources needed by the CD region and then pre-allocate these

resources (Cher, et al., 2001). Padding eliminates the need to expand the ROB and LSQ when inserting instructions in the middle of the window. Invalid entries (due to removing instructions or not using all reserved entries) are not reclaimed immediately. Instead, reclamation is delayed until invalid entries reach the head of the ROB or LSQ. Delayed reclamation and inaccurate pre-allocation of resources can cause internal fragmentation. This approach underutilizes the ROB/LSQ and may degrade performance.

4) *Temporary buffer assisted shifting*: This method achieves the same goal as the collapsing/expanding buffer but with fewer downsides. After a branch misprediction is detected, this method starts copying the CI instructions into a temporary buffer in preparation for moving them to their new locations. After the new instructions have been inserted (the correct CD instructions overwrite the incorrect CD instructions), it is clear where the CI instructions buffered in the temporary buffer need to be copied. Delaying copying of CI instructions until the right location is known emulates the need to collapse and expand the buffer many times. This reduces the hardware complexity compared to a collapsing/expanding buffer. However, this method shares one downside with the collapsing/expanding buffer in that the shifting process may take many cycles when compared with a linked-list implementation of the ROB/LSQ. This solution is described in more detail in Section 6.3.1.

Alternatively, a ROB-free checkpoint-based architecture may be used (Akkary, et al., 2003) (Cristal, et al., 2004) (Cristal, et al., 2002). The solution substitutes fine-grain retirement using the ROB with coarse-grain retirement using checkpoints. By removing the

ROB, we are only left with the LSQ as an *ordered resource* in the processor to deal with. Identifying the synergy between control independence and ROB-free checkpoint-based processors is a contribution of this thesis and is explored in depth in Chapter 6.

Unlike the ROB and LSQ, the register file's (RF) entries (physical registers) are not allocated and freed in program order, making it compatible with control independence. Unfortunately, conventional processors manage the freeing of RF entries using an ordered free list that is incompatible with control independence.

The RF's free list is ordered, giving the RF some *ordered resource* traits. Conventionally, the RF can reclaim physical registers in bulk after a branch misprediction like the ROB and LSQ. This is achieved by checkpointing the free list head pointer at branches, and restoring the checkpointed head pointer corresponding to a mispredicted branch. This single action bulk-frees the physical registers of all instructions – both CD and CI – after the mispredicted branch. Traditional management of the RF free list is incompatible with control independence. Simply restoring the free list head pointer to its checkpointed location at the mispredicted branch is not sufficient, because physical registers allocated to CI instructions must not be freed. Instead, selectively freeing physical registers requires walking the incorrect CD instructions to free only their physical registers. In addition, allocating new physical registers to the correct CD instructions in the middle of the window will require the order of the free list to be repaired (just like the ROB and LSQ).

To address this issue, we propose using an alternate register freeing mechanism that does not rely on an ordered free list. Aggressive register reclamation does not use an ordered free list and is based instead on usage counters (Moudgill, et al., 1993) (Akkary, et al., 2003). A register is freed once all known consumers read the register's value and the register is no longer referenced by any checkpoint or rename map table. Leveraging aggressive register reclamation to manage the RF makes it fully compatible with control independence. This proposed solution is used in TCI (Chapter 6) and is one of many novel contributions of this thesis.

5.2 Repairing the CI region

This section discusses implementation aspects and performance factors of repairing the CI region. The CI instructions are already in the window and need not be re-fetched. However, some CIDD instructions need to have their register or memory data dependencies repaired to reflect the repaired CD region and then all CIDD instructions need to be re-executed.

5.2.1 Repairing the data dependencies of CIDD instructions

CIDD instructions are either directly data dependent on the CD region (direct CIDD) or indirectly data dependent on the CD region (indirect CIDD). Direct CIDD instructions have potentially stale source register names or stale memory dependencies, after repairing the CD region. Indirect CIDD instructions are data dependent on the direct CIDD instructions. Instructions that depend on indirect CIDD instructions are also indirect CIDD instructions. Direct and indirect CIDD instructions can be identified with the help of information provided by the CFS, IRS, and store/branch set predictor described in Chapter 4.

Repairing the direct CIDD instructions' data dependencies involves repairing the register data dependencies and the memory data dependencies. Memory data dependencies need not be explicitly repaired. CIDD Loads will repair their memory data dependencies when they are re-executed (they will have access to the repaired CD store instructions). On the other hand, register data dependencies need to be explicitly repaired before re-execution can start. Register data dependencies are repaired by correcting the source register names of direct CIDD instructions, linking them to their correct producers.

Traditional processors with conventional (full) branch misprediction recovery do not support selectively re-renaming CIDD instructions that are already in the window. In the following three sub-sections, we look at three possible mechanisms (*Seq CI*, *Proxy*, and *Seq CIDD*) to achieve this goal and investigate their impact on a traditional processor's performance and complexity.

5.2.1.1 Sequencing CI instructions (Seq CI)

One way to correct the CIDD instructions' source register names is to use the same approach used by conventional (full) branch misprediction recovery to generate correct data dependencies. With conventional recovery, all instructions after the misprediction are squashed (incorrect CD instructions and CI instructions) and the correct instructions are fetched and renamed (correct CD instructions and CI instructions). Using this approach with control independence requires some modifications because the CI instructions are not squashed and need not be re-fetched.

The modified approach would only squash the incorrect CD instructions and fetch the correct CD instructions, but would rename all instructions after the mispredicted branch (correct CD instructions and CI instructions). Since the modified approach does not re-fetch the CI instructions, it needs to sequence through the CI instructions already buffered in the processor. Therefore, this approach is referred to as sequence CI (*Seq CI*).

The *Seq CI* approach leverages the existing register rename stage of the processor to re-rename all CI instructions after a mispredicted branch. In the re-renaming process, direct CIDD instructions' source mappings are automatically repaired, reflecting the corrected control-flow (Rotenberg, et al., 1999) (Chou, et al., 1999).

The advantage of *Seq CI* is that it uses the processor's already available mechanisms to achieve its goal. However, the process of re-renaming all CI instructions is tantamount to re-fetching all CI instructions, as implemented by conventional (full) recovery. This degrades the potential performance of control independence. Furthermore, *Seq CI* requires buffering all CI instructions in program order in the processor so that they can be sequenced in case of a branch misprediction. This requirement adds an *ordered resource* to the processor, similar to the ROB, which has the potential to further degrade performance and complicate the design of the control independence implementation.

5.2.1.2 Proxy move instructions (Proxy)

An alternative to re-renaming the CIDD instructions is to use proxy move instructions (*Proxy*). *Proxy's* goal is to insulate the CIDD instructions from source name changes caused by repairing the CD region (Cher, et al., 2001) (Gandhi, et al., 2004).

To implement the *Proxy* method, the processor needs to insert a proxy move instruction for each production in the CD region at the reconvergent point (between the CD instructions and CI instructions). The destination physical registers of the proxy move instructions are pinned. This ensures source names of direct CIDD instructions do not change with changing control-flow. After renaming the correct CD instructions, only the proxy move instructions need to be re-renamed to repair their source names. The proxy move instructions then forward the values from their re-renamed source physical registers to their pinned destination physical registers. This assures that CIDD instructions will receive the correct source values during re-execution.

CD region productions need to be identified to know which proxy move instructions to insert at the reconvergent point. This information can be provided by the IRS. In addition, to be able to re-rename the proxy move instructions, they need to be buffered in the processor.

Proxy has the advantage of eliminating the need to re-rename CI instructions when servicing a branch misprediction, however, at the cost of extra resource pressure and renaming bandwidth for the added proxy move instructions. Extra physical registers, issue queue entries, etc. are consumed by the proxy moves. More importantly, *Proxy's* resource overhead and rename overhead affects the performance of the system even when all branches are correctly predicted, forcing us to cover branches selectively for good overall performance. This problem is not present in the sequencing repair approaches (*Seq CI* and *Seq CIDD*).

5.2.1.3 Sequencing CIDD instructions (Seq CIDD)

The two re-renaming techniques discussed previously have opposing tradeoffs. On the one hand, the *Seq CI* approach has a high re-rename bandwidth requirement, but does not incur extra resource or execution bandwidth. On the other hand, the *Proxy* approach reduces the required re-rename bandwidth, but requires extra cycle-critical resources and execution bandwidth for the proxy move instructions. Ideally, we would like a solution that minimizes re-rename bandwidth while not requiring any extra cycle-critical resources.

This thesis introduces a new selective re-renaming mechanism. By pre-identifying the CIDD instructions during dispatch, this technique attempts to only re-rename the CIDD instruction stream and hence is called the sequence CIDD (*Seq CIDD)* approach. This approach has the potential to conserve re-renaming bandwidth compared to the *Seq CI* approach.

Like the *Seq CI* approach, the *Seq CIDD* approach leverages the existing register renaming stage of the processor to repair the source names of the CIDD instructions. After renaming the correct CD instructions, *Seq CIDD* re-renames the pre-identified CIDD instructions, repairing their source names. The process of re-renaming only the CIDD instructions needs special care, as this instruction stream contains holes corresponding to absent CIDI instructions.

Conventional register renaming requires processing instructions in program order to ensure that correct register linkages are produced. However, CIDD and CIDI instructions are interleaved, which makes conventional renaming inadequate for renaming only CIDD instructions. Figure 23 shows an example of a sequence of instructions from the CI region. Register R1 depends on the CD region, therefore, instructions with "*" are CIDD. When using the *Seq CI* repair mechanism, instruction #4 would receive a source mapping of P51 for R5, which is correct. However, renaming only the CIDD instructions would cause instruction #4 to receive P50 as a mapping for R5 during re-renaming, which is incorrect. This is a consequence of having holes in the CIDD instruction stream. To circumvent this problem, *Seq CIDD* requires identifying CIDI-supplied source registers, in the example of instruction #4 the source R5, and avoiding re-renaming of these source operands. Source operands not re-renamed would need to reuse their old source names. A detailed control independence implementation using the *Seq CIDD* approach is presented in Chapter 6. Note that our approach goes a step further, supplanting the CIDI-supplied source registers with their actual values.

```
#1*: Add R5(P50), R1(P10), R2(P20)
#2*: Add R3(P31), R1(P10), R5(P50)
#3 : Add R5(P51), R4(P40), R6(P60)
#4*: Add R7(P70), R1(P10), R5(P51)
```

Figure 23. Sequence of instructions from the CI region.

In addition to saving re-renaming bandwidth, *Seq CIDD* reduces the instruction buffering requirement compared to *Seq CI*. In *Seq CI*, all instructions need to be buffered (like in a ROB) in anticipation for a branch misprediction. However, in *Seq CIDD*, only the CIDD instructions need to be buffered. This is especially important when looking at very large instruction windows.

5.2.2 Re-executing CIDD instructions

After correcting the CIDD instructions' source names, all CIDD instructions need to re-execute to produce correct values. Re-execution of CIDD instructions is not fully supported by conventional processors. CIDD instruction re-execution requires all needed resources (issue queue entries and physical registers) be allocated (Section 5.2.2.1), and also requires that their source operands' resources (physical registers) be available (Section 5.2.2.2).

5.2.2.1 CIDD instructions' resources

Re-execution (like normal execution) of CIDD instructions requires that they are allocated issue queue entries and destination physical registers. One approach to support re-execution is to hold the resources (issue queue entries and physical registers) originally allocated to the CIDD instructions when they were first fetched and dispatched until the branch resolves (*Hold IQ*). This minimizes changes to the resource management of the processor. However, re-execution may occur many cycles after the first execution has occurred. Holding these cycle-critical resources for this whole period of time can create extra resource pressure. In turn, this can degrade performance compared to conventional speculation, which allows resources to be reclaimed aggressively (issue queue entries, and physical registers when using aggressive register reclamation) since full recovery squashes and re-fetches all instructions after a branch misprediction.

Alternatively, the resources allocated to CIDD instructions can be released aggressively according to conventional speculation, and reallocated only when re-execution is needed (*Drain IQ*). So, after the CIDD instructions execute for the first time, they are free to release their resources. These CIDD instructions are buffered in an auxiliary re-execution buffer (RXB) for purposes of re-execution. After the branch misprediction is detected and during recovery of the CI region, all CI (*Seq CI*) or only CIDD (*Seq CIDD*) instructions are re-renamed. During re-renaming, CIDD instructions are re-dispatched into the issue queue and reallocated destination physical registers. This provides the CIDD instructions with the needed resources for re-execution. Therefore, *Drain IQ* trades the resource pressure of *Hold*

IQ for additional re-rename bandwidth needed to reallocate resources for the CIDD instructions.

5.2.2.2 CIDI instructions' resources

In addition to the resources held by the CIDD instructions themselves, re-execution requires that all CIDD instructions' source operands be allocated physical registers. Source values feeding CIDD instructions, but not produced by other CIDD instructions, need to be available for correct execution. These producers may include CIDI instructions, CD instructions, or instructions from before the branch. Note that latest producers of architectural registers from before the reconvergent point (CD instructions and instructions from before the branch) will still have their physical registers allocated (hence referencable) because their mappings will be live in the rename map used for repairing the CD region. On the other hand, CIDI instructions are co-mingled with the CIDD instructions and need to be dealt with specially. One solution is to force all CIDI instructions to hold their resources. This solution makes control independence architectures inefficient and defeats the purpose of resource-efficient techniques such as aggressive register reclamation.

Alternatively, the CIDI instructions can release their physical registers by depositing their values in the RXB (in program order with respect to interleaved CIDD instructions). Hence, if re-execution of the CIDD instructions is required, the CIDI instructions can be reallocated physical registers and their saved destination values can be reloaded into the newly allocated physical registers. One could further optimize this solution by allowing the CIDD instructions to replace their CIDI-supplied source register mappings with the actual checkpointed values from the CIDI instructions. This avoids revisiting CIDI instructions altogether, further increasing overall efficiency by virtue of (1) not including CIDI instructions in the RXB, (2) not re-renaming CIDI instructions (re-renaming bandwidth), and (3) not allocating registers to CIDI instructions during recovery.

5.2.3 Control independence configurations

Different control independence implementations have different resource and bandwidth overheads, depending on the way they fulfill the requirements to repair the CD and CI regions. In this section, we will focus on the performance impact of the different approaches to repair the CI region, on a common substrate.

For a common substrate, we choose a ROB-free checkpoint-based processor with aggressive register reclamation, for its compatibility with control independence. First, this substrate simplifies the repair of the CD region by minimizing the number of *ordered resources* used in the substrate. The ROB, which traditionally was a major complication for control independence implementations, is removed, allowing for arbitrary removal and insertion of CD instructions. In addition, using aggressive register reclamation allows the register file to be fully compatible control independence (avoid managing the RF with the ordered free list). Physical registers are now reclaimed based on usage counters and not the ROB.

Second, this substrate is resource-efficient, which allows it to construct a large window of instructions with small cycle-critical resources. Interestingly, control independence's ability to tolerate branch mispredictions, avoiding squashing of CI instructions, also permits creating a large window of useful instructions. Hence, the resource efficiency of the checkpointed substrate coupled with the branch misprediction tolerance of control independence form a symbiotic relationship.

The only remaining complication in repairing the CD region involves the LSQ. For this *ordered resource*, we will employ the temporary buffer assisted shifting solution. This minimizes the effect of repairing the CD region, allowing us to focus on repairing the CI region.

The CI region repair process involves repairing the CIDD instructions' source register mappings and then re-executing the CIDD instructions to generate correct results. Different implementations repair CIDD instructions in different ways, and with various resource and bandwidth overheads.

Three source mapping repair mechanisms are studied. *Proxy* uses proxy move instructions to insulate the CI instructions from source name changes, needing only to re-rename the proxy move instructions after a branch misprediction. *Seq CI* re-renames all CI instructions. *Seq CIDD* re-renames only the pre-identified CIDD instructions. *Seq CIDD* requires TCI's mechanisms and is described in detail in Chapter 6 as part of a complete implementation.

For re-execution of CIDD instructions, two methods are investigated. The first is conservative and holds all instructions in the issue queue to achieve selective re-execution. This model is labeled *Hold IQ*. This selective re-execution approach is used by some prior control independence architectures designed on superscalar substrates (Rotenberg, et al., 1999) (Cher, et al., 2001) (Gandhi, et al., 2004). The second selective re-execution substrate is more aggressive and allows instructions to drain out of the issue queue fully or partially.

This model is labeled *Drain IQ*. Under this model, selective re-execution is accomplished in different ways based on the source mapping repair mechanism. For *Proxy*, the issue queue is partially drained, leaving the proxy move instructions and the CIDD instructions in the issue queue for re-execution. This is signified by *Drain IQ (partial)*. For *Seq CI*, all instructions (including data they produce) are drained and buffered in a re-execution buffer (RXB). Selective re-execution is achieved by sequencing the RXB and re-dispatching the CIDD instructions. Similarly, *Seq CIDD* also uses the RXB for selective re-execution. However, for *Seq CIDD*, only the pre-identified CIDD instructions are buffered in the RXB. The RXB-based models are signified by *Drain IQ (all)*.

Table 4. Resource and bandwidth usage for repairing CIDD instructions.

	Model	Hold resources until branch resolves			CI resequencing bandwidth		Related work
		Registers	Issue Queue	RXB	Re-renaming	Re-execution	
	Base	none	none	none	CIDD + CIDI	CIDD + CIDI	
Hold IQ	**Proxy**	all	all	none	proxy	CIDD + proxy	(Cher, et al., 2001)[a], (Gandhi, et al., 2004)
Hold IQ	**Seq CI**	all	all	none	CIDD + CIDI	CIDD	(Rotenberg, et al., 1999), (Rotenberg, et al., 1999)
Hold IQ	**Seq CIDD**	all	all	none	CIDD	CIDD	
Drain IQ	**Proxy**	some CIDI + CIDD + proxy	CIDD + proxy	none	proxy	CIDD + proxy	
Drain IQ	**Seq CI**	none	none	all	CIDD + CIDI	CIDD	(Akkary, et al., 1998), (Chou, et al., 1999)
Drain IQ	**Seq CIDD**	none	none	CIDD	CIDD	CIDD	TCI

[a]Cited for the use of proxy instructions, and not skipper style control independence.

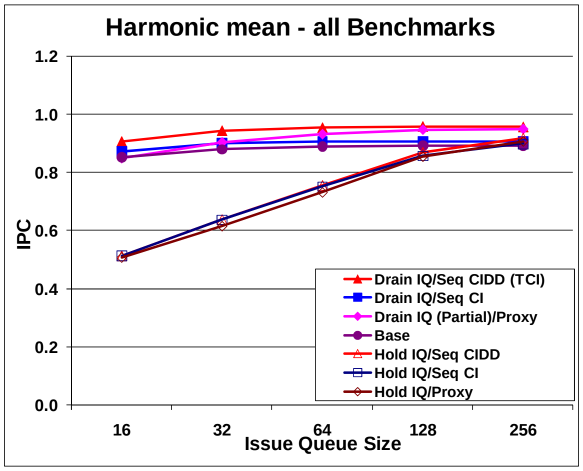
Table 4 compares the resource and bandwidth overheads for repairing the CIDD instructions, for *Base* (conventional recovery), *Proxy*, *Seq CI*, and *Seq CIDD*, on both *Hold IQ* and *Drain IQ* re-execution substrates. The *Base* model always drains instructions out of the issue queue (since conventional recovery squashes all instructions after a branch misprediction). In addition, the last column in Table 4 cites previous implementations from the literature that share the same CIDD repair mechanism (but not necessarily the base substrate or the CD repair mechanisms). Resources are divided into physical register usage (Registers), issue queue entries held (Issue Queue), and instructions occupying the RXB (RXB). Bandwidth is divided into re-renaming and re-execution bandwidth. *Drain IQ/Seq CIDD* is qualitatively the best or tied for best in every category and is the basis for the TCI implementation discussed in Chapter 6. *Drain IQ/Seq CIDD* may re-rename fewer or more instructions than *Proxy*, depending on the number of proxy instructions and CIDD instructions being re-renamed. Moreover, *Drain IQ/Seq CIDD* does not incur resource/bandwidth overheads on correctly predicted branches like *Proxy*.

5.2.4 Results (resource and bandwidth overheads)

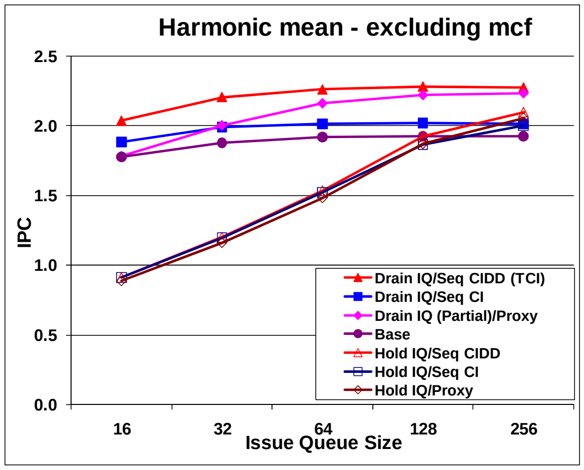
Figure 24 (a) shows the harmonic mean of IPCs for 15 of the SPEC integer benchmarks listed in Section 2.2, for the seven models. Figure 24 (b) excludes benchmark mcf from the harmonic mean because the extremely low IPC of mcf obscures trends. The issue queue size is varied to understand resource pressure. The resource inefficiency of the *Hold IQ* re-execution substrate is a major bottleneck with small issue queues. In fact, *Base* outperforms all *Hold IQ* models, for issue queues with fewer than 256 entries (128 entries excluding mcf). This is because the issue queue limits the overall window size when all instructions are held in the issue queue.

Ideally, all instructions should free all cycle-critical resources speculatively, allowing for a bigger window, and CIDD instructions should only be re-allocated resources when selective re-execution is required after a branch misprediction. *Drain IQ* strives for this goal. However, *Proxy* falls short of this ideal scenario because proxy and CIDD instructions remain in the issue queue for possible selective re-execution. The remaining issue queue pressure is evident in Figure 24: *Proxy* is unique in its sensitivity to issue queue size compared to other models with *Drain IQ*. In fact, for a 16-entry issue queue, *Proxy* has no performance advantage over conventional recovery (*Base*) let alone the other selective recovery approaches. On the other hand, a 64-entry issue queue enables *Proxy* to overtake *Seq CI*. Overall, *Seq CIDD* (TCI) performs the best due to its combined bandwidth and resource efficiency.

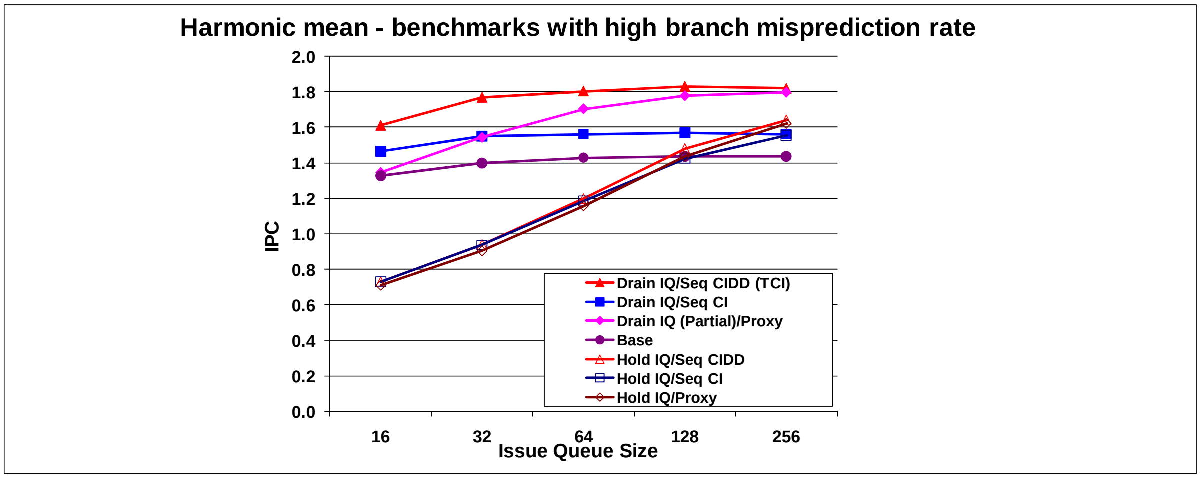
Figure 24 (c) shows the harmonic mean of IPCs for benchmarks with high branch misprediction rates, for all the models. This includes bzip, compress, go, gzip, twolf, and vpr, all of which have more than 9 branch mispredictions per 1000 instructions. In these benchmarks, the branch misprediction penalty is severe, so there is opportunity for large improvements with the help of control independence. Therefore, we notice that the improvements over *Base*, for all *Drain IQ* models, have further increased, but the trends remain the same.



(a)



(b)



(c)

Figure 24. Performance of different CIDD repair models (harmonic mean).

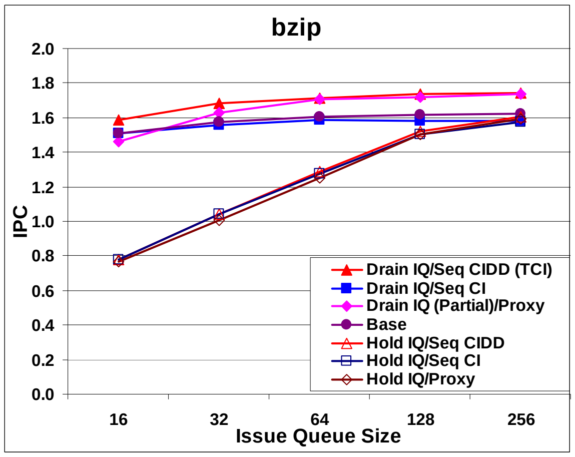
Figure 25, Figure 26, and Figure 27 show the IPC results of the individual benchmarks for the seven models. Looking at individual benchmark IPCs for the *Drain IQ* re-execution substrate, some interesting phenomena can be observed.

1) *Drain IQ/Seq CI* can sometimes degrade performance with respect to *Base*. For example, gap, vortex, and vpr show *Drain IQ/Seq CI* performs worse than *Base*, most of the time. This is due to the window size limitation caused by buffering all

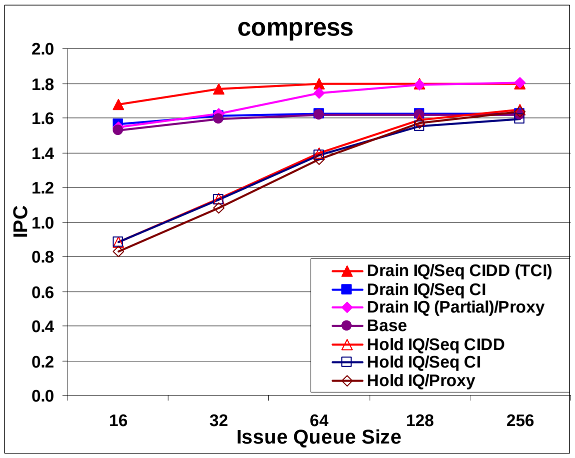
instructions in the RXB (RXB size is 256 instructions). *Drain IQ/Seq CIDD* does not have this limitation since it uses the RXB resource wisely, only storing CIDD instructions, making the RXB less of a bottleneck (or not at all).

2) *Drain IQ (partial)/Proxy* can sometimes degrade performance with respect to *Base* for small issue queues. This can be observed in bzip, crafty, li, and vpr for a 16-entry issue queue.

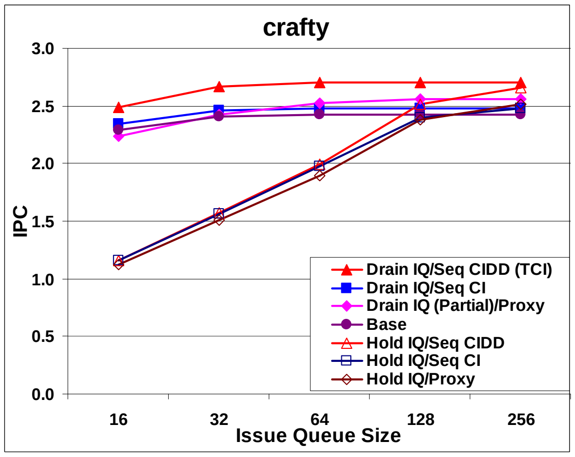
3) Although *Drain IQ/Seq CIDD* outperforms *Drain IQ (partial)/Proxy* most of the time, we notice that, for the benchmark vpr, *Drain IQ (partial)/Proxy* slightly outperforms *Drain IQ/Seq CIDD* with an issue queue size of 256. One possible reason is that proxy move instructions are no longer a resource concern with a 256-entry issue queue, and during re-renaming, there are fewer proxy move instructions to be re-renamed than the number of CIDD instructions needing re-renaming.



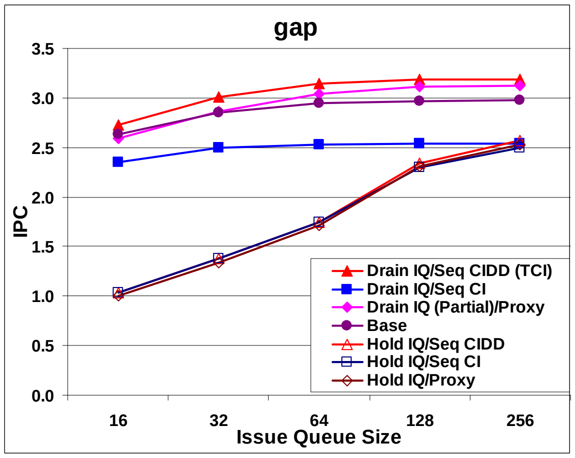
(a)



(b)

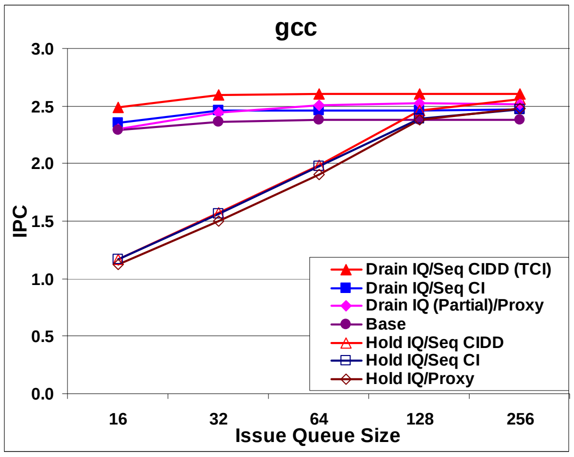


(c)

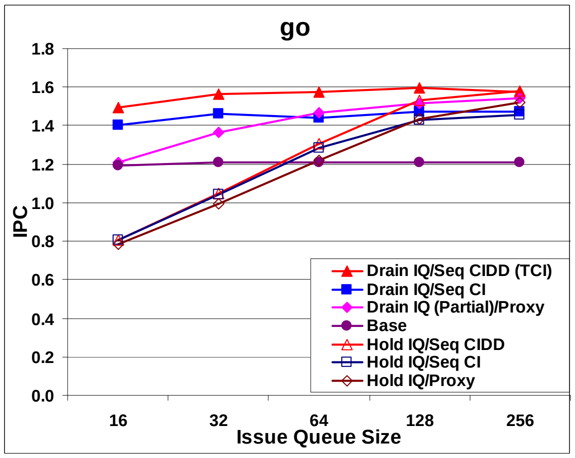


(d)

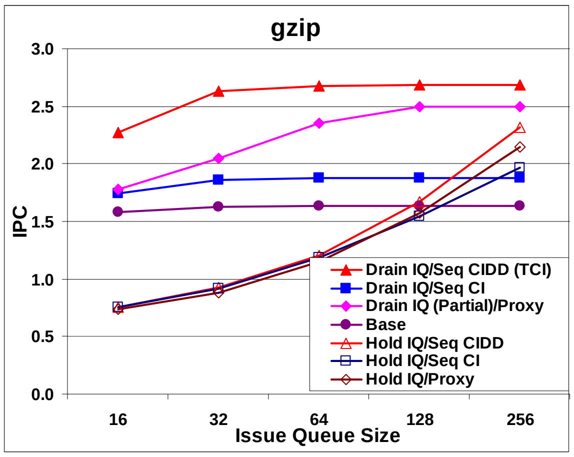
Figure 25. Performance of different CIDD repair models (individual benchmarks).



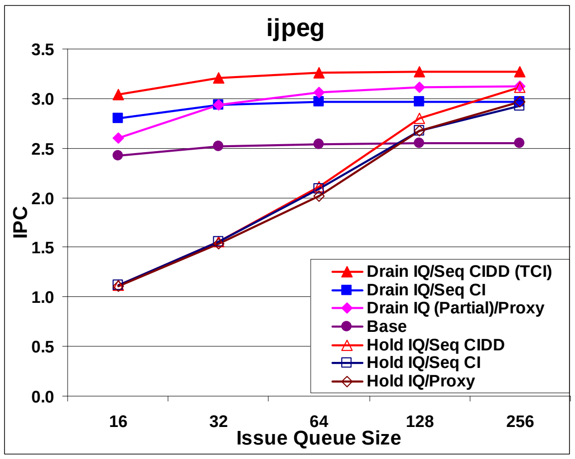
(a)



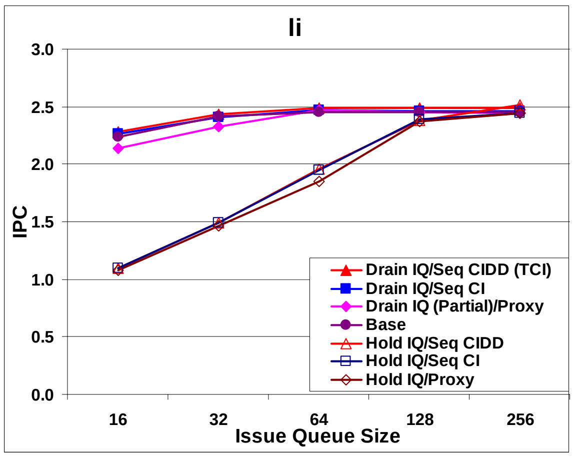
(b)



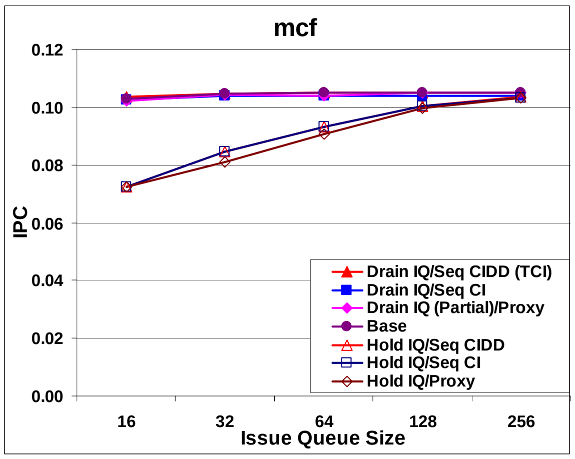
(c)



(d)

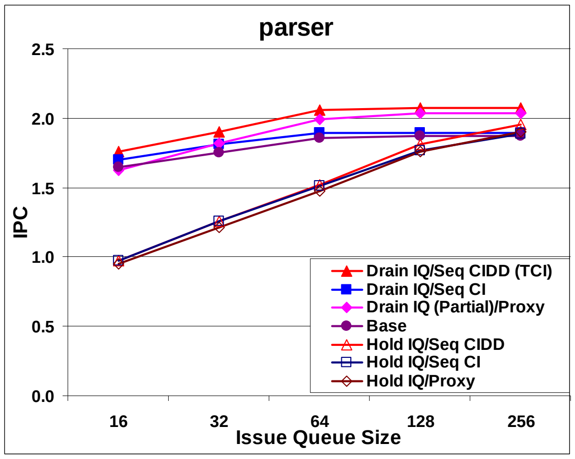


(e)

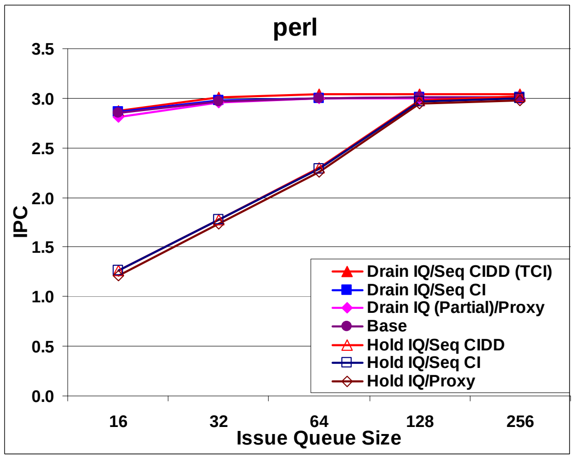


(f)

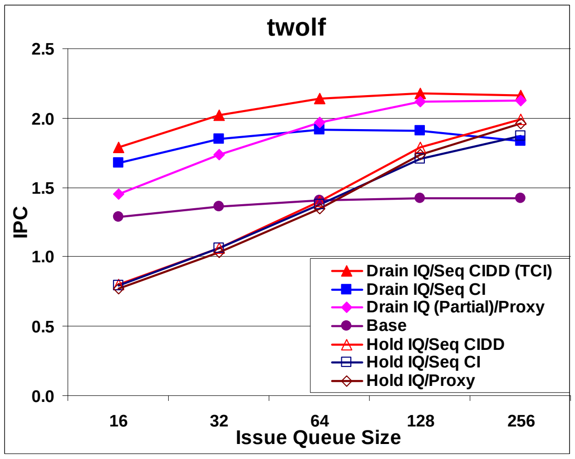
Figure 26. Performance of different CIDD repair models (individual benchmarks).



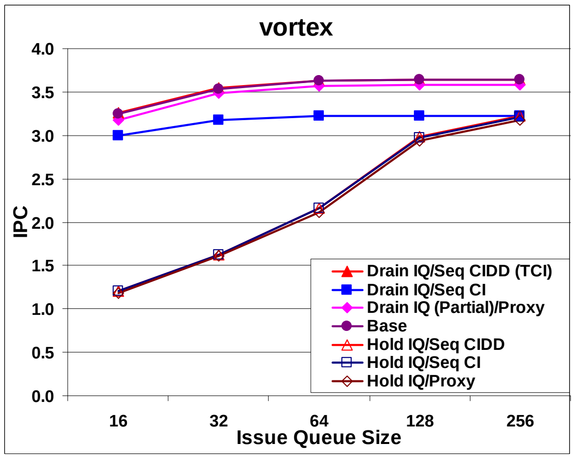
(a)



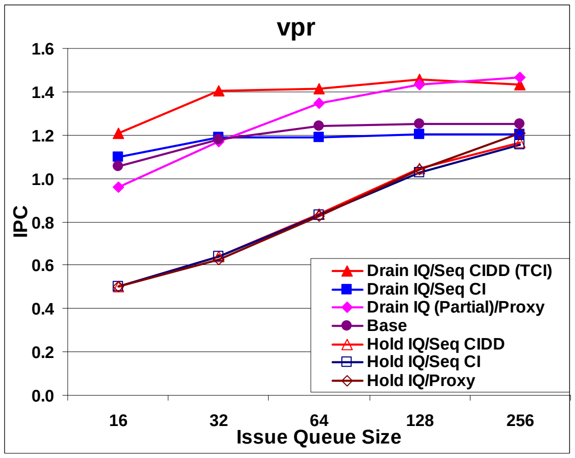
(b)



(c)



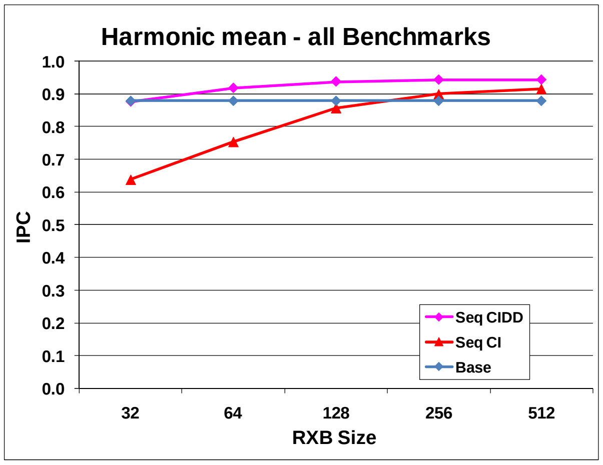
(d)

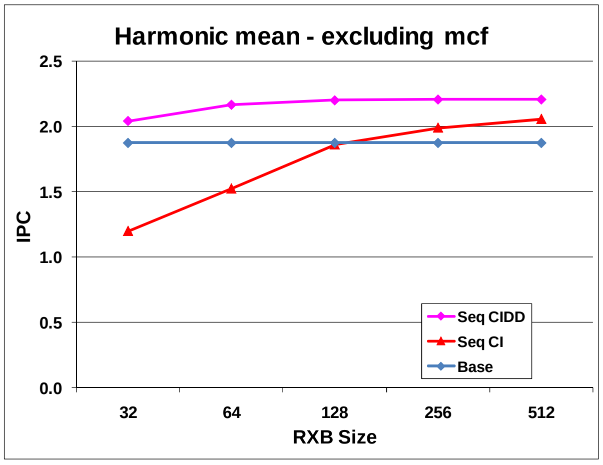


(e)

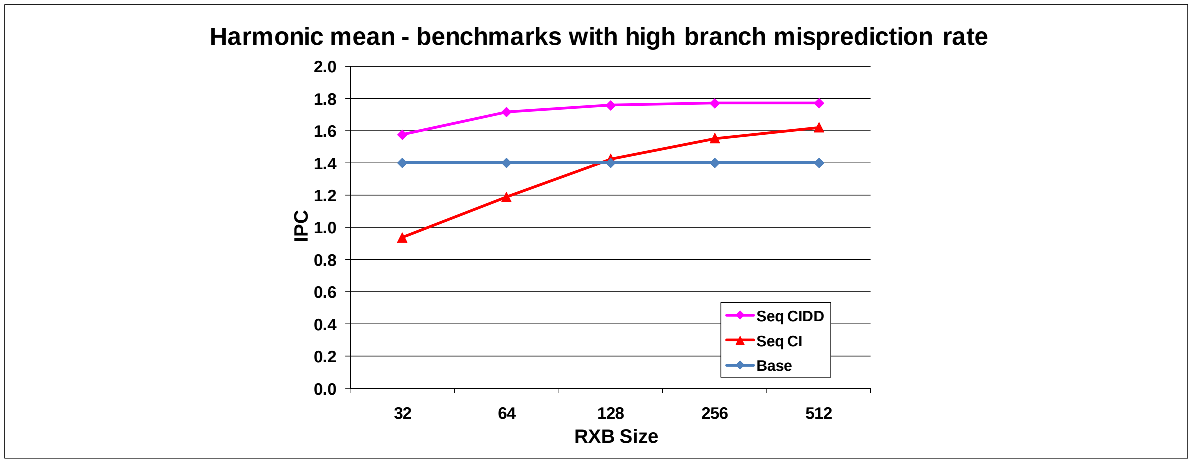
Figure 27. Performance of different CIDD repair models (individual benchmarks).

Figure 28(a-c) shows the performance sensitivity of *Drain IQ/Seq CI* and *Drain IQ/Seq CIDD* to the RXB size. In addition, the *Base* model is included for reference (*Base* is not affected by the change in RXB size). Figure 28(a) shows the harmonic mean IPC of all benchmarks, Figure 28(b) excludes mcf from the harmonic mean (to be consistent with previous graphs), and finally Figure 28(c) shows the harmonic mean IPC of benchmarks with high branch misprediction rates (more than 9 branch mispredictions per 1000 instructions). *Seq CI* is very sensitive to RXB size since all instructions are inserted into the RXB, therefore, the RXB limits the overall window size (like a ROB). In contrast, *Seq CIDD* is much less sensitive to the RXB size since only CIDD instructions are inserted into the RXB, therefore, the RXB does not limit the overall window size. This trend is observable in all the figures. In fact, *Seq CIDD* only needs an RXB of 64 entries to achieve most of the performance potential (on average).





(a) (b)



(c)

Figure 28. Sensitivity to RXB size.

Figure 29, Figure 30, and Figure 31 show the IPC results of the individual benchmarks for *Seq CI* and *Seq CIDD*. Looking at individual benchmarks, two interesting trends can be observed. First, for 10 of 15 benchmarks, *Seq CIDD* only needs an RXB with 32 entries. Second, we notice that *Seq CI* degrades performance with respect to *Base* even for benchmarks with low branch misprediction rates, such as gap and perl. On the other hand, *Seq CIDD* does not have this problem as it occupies the RXB based on branch confidence, allowing highly predictable branches' CIDD instructions to circumvent the RXB.

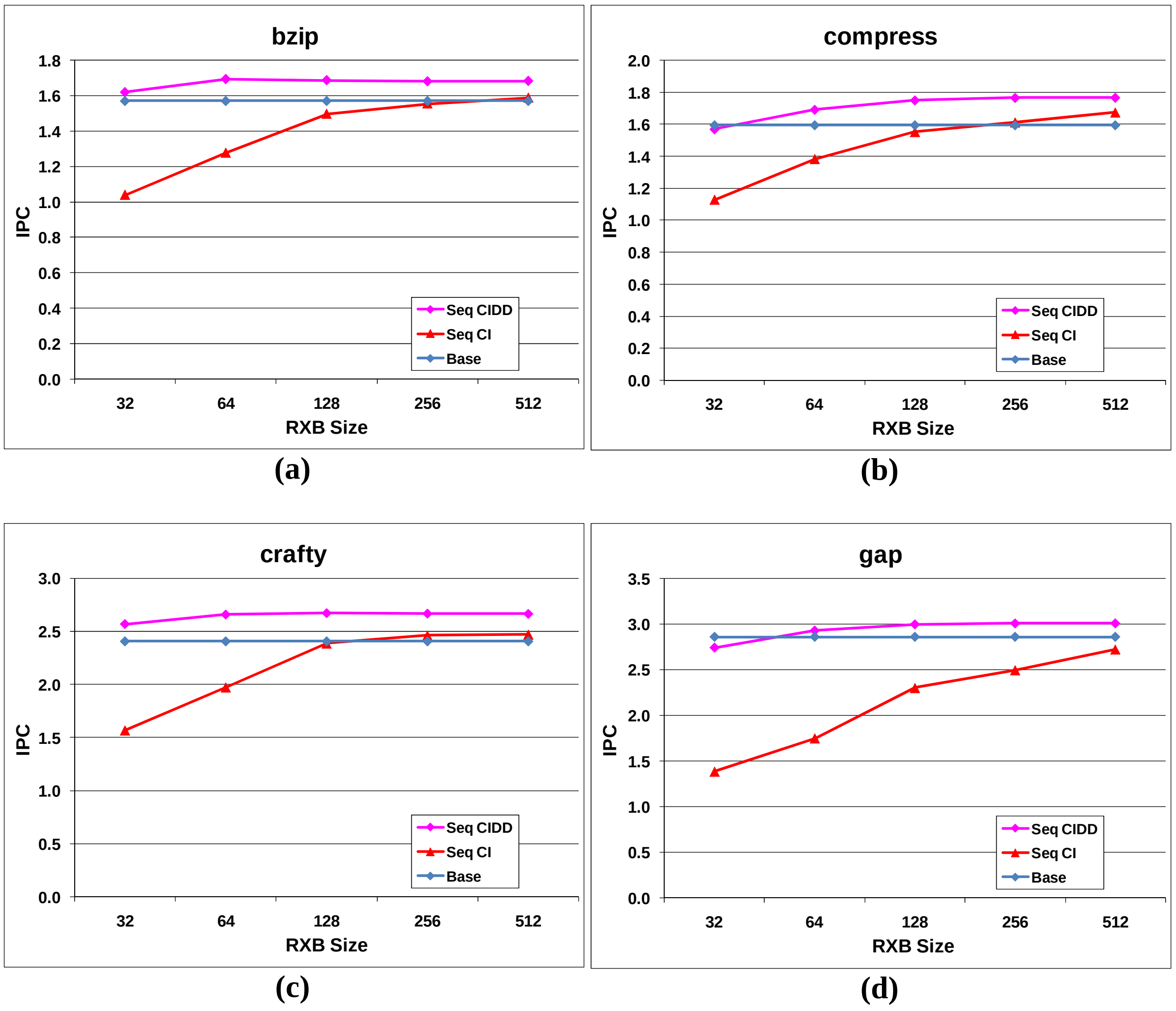
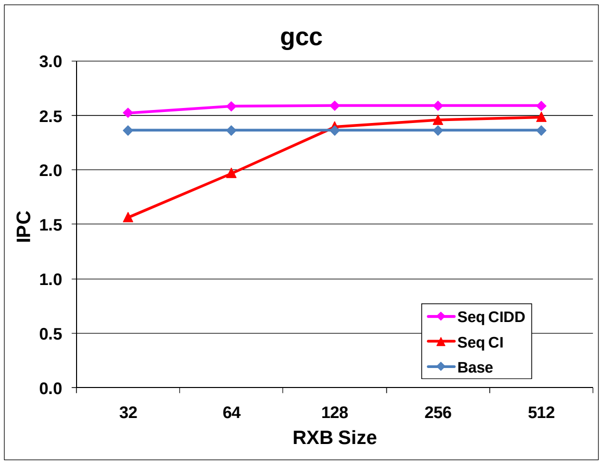
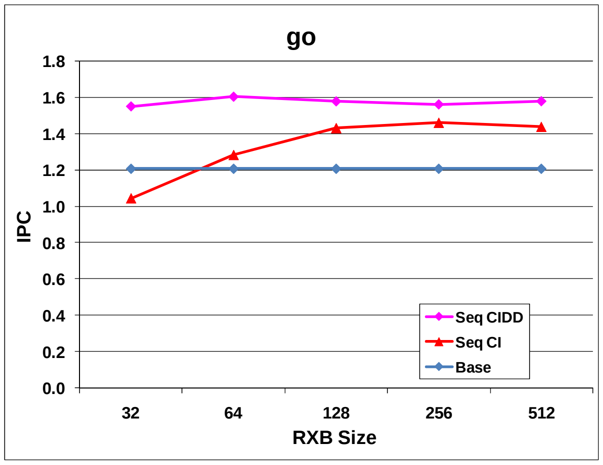


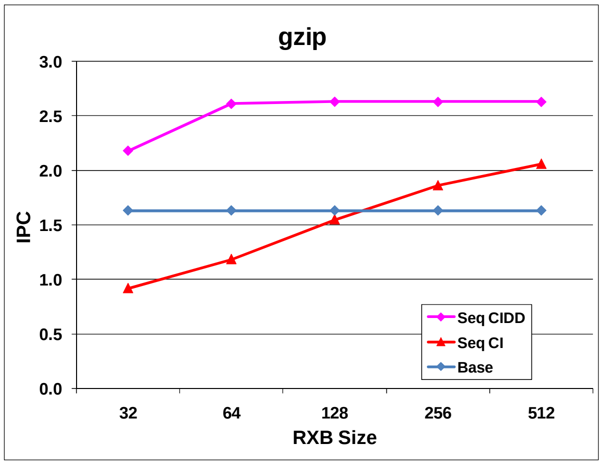
Figure 29. Sensitivity to RXB size (individual benchmarks).



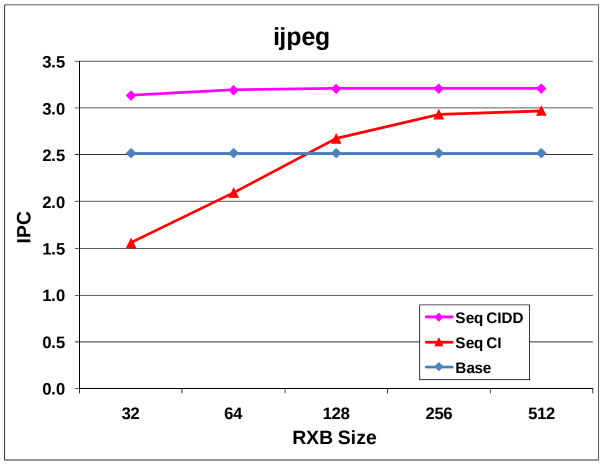
(a)



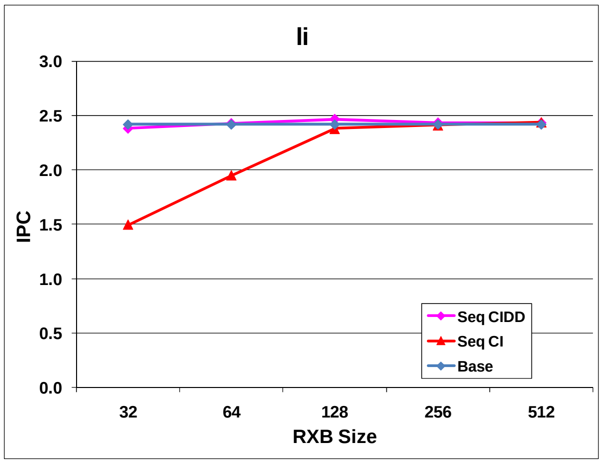
(b)



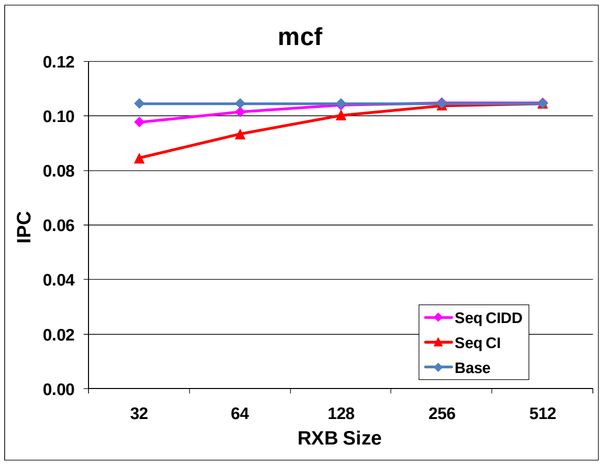
(c)



(d)

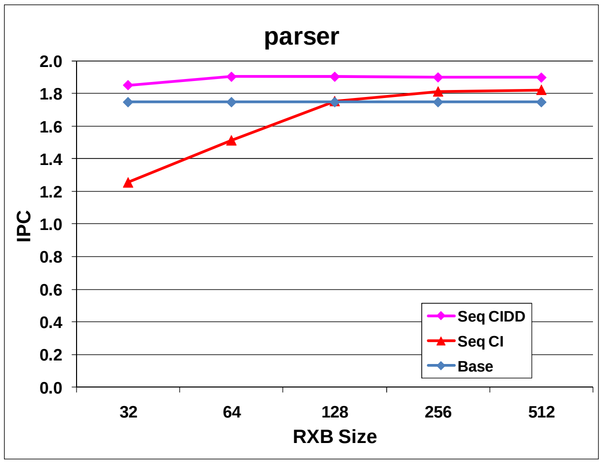


(e)

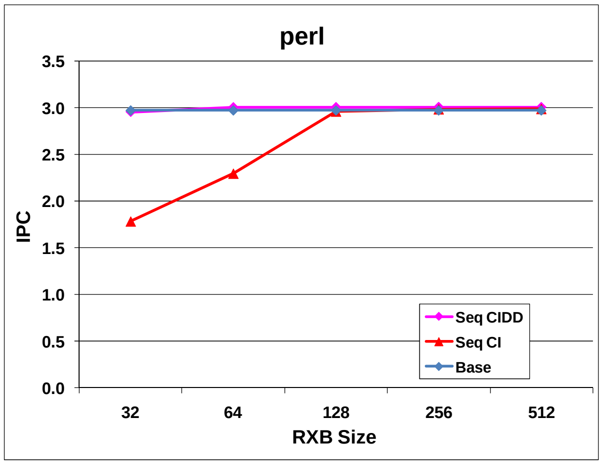


(f)

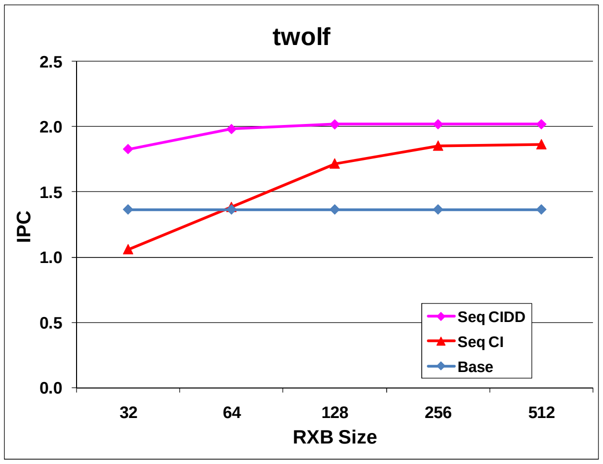
Figure 30. Sensitivity to RXB size (individual benchmarks).



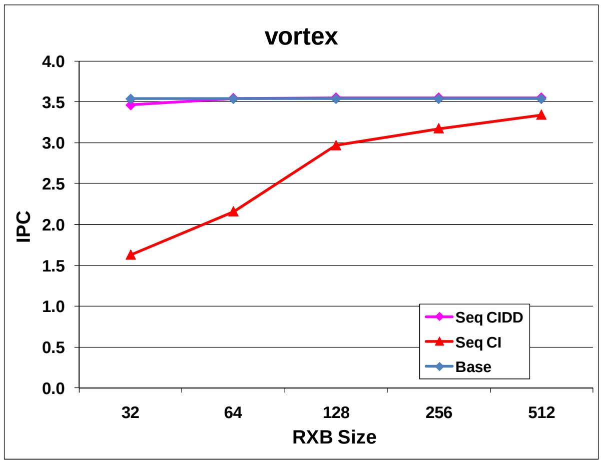
(a)



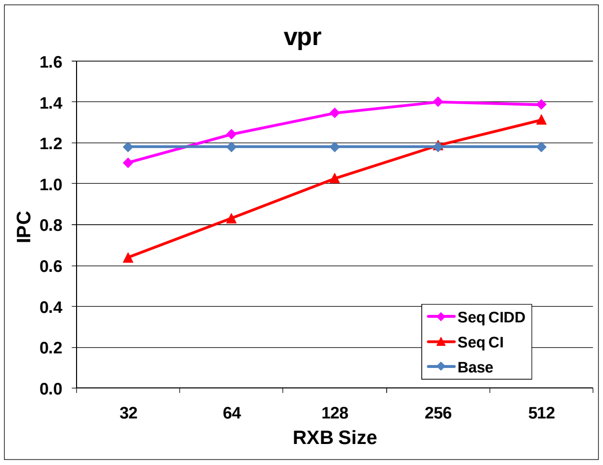
(b)



(c)



(d)



(e)

Figure 31. Sensitivity to RXB size (individual benchmarks).

Chapter 6

Transparent Control Independence (TCI)

In Chapter 5, we contrasted and quantified the resource and bandwidth overheads of *Proxy*, *Seq CI*, and *Seq CIDD*. We found that *Seq CIDD* combines the advantages of *Proxy* (bandwidth-efficient) and *Seq CI* (resource-efficient) and eliminates their disadvantages, while fulfilling the CI region re-renaming requirement. In this chapter, we present a new microarchitecture that implements the *Seq CIDD* model. By proactively identifying CIDD instructions in preparation for a branch misprediction, we only re-rename CIDD instructions and only reallocate resources to CIDD instructions when recovery is needed.

6.1 High-level overview of TCI microarchitecture

Figure 32 shows our transparent control independence (TCI) architecture. The shaded region highlights a resource-streamlined pipeline that aggressively releases resources based on conventional speculation. Correct and incorrect instructions alike flow through the pipeline as fast as they would with conventional speculation, aggressively freeing issue queue entries and physical registers (Akkary, et al., 2003) (Cristal, et al., 2004) (Moudgill, et al., 1993) (Srinivasan, et al., 2004) on the assumption that branch predictions are correct. Instructions drain from the pipeline as soon as they complete – there is no reorder buffer (ROB) and precise exceptions are achieved via checkpoints (Akkary, et al., 2003) (Cristal, et al., 2004) (Moudgill, et al., 1993) (Srinivasan, et al., 2004) (Hwu, et al., 1987).

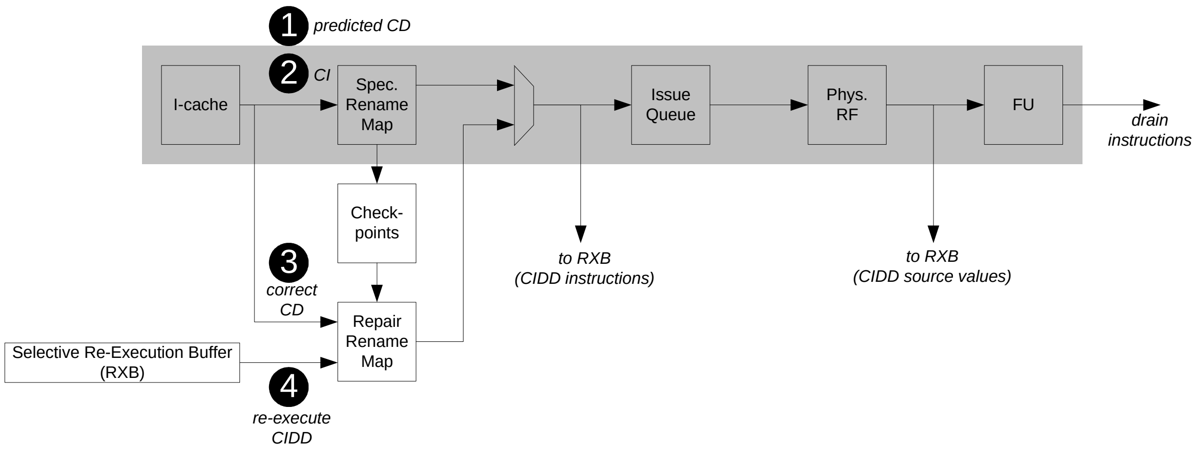


Figure 32. Transparent control independence (TCI) architecture.

When a branch is encountered in the fetch unit, its predicted CD instructions are fetched from the instruction cache (I-cache), highlighted in Figure 32 with Step-1. These are soon followed by the branch's CI instructions, corresponding to Step-2 in the figure. Both the predicted CD and CI instructions are renamed with the speculative rename map and sent down the pipeline. The branch's CIDD instructions are identified in the dispatch stage and duplicates of these instructions are set aside in a FIFO buffer, the Selective Re-execution Buffer (RXB), as shown. When these instructions issue and read their source operands from the physical register file, copies of the source values are also set aside with the corresponding instructions in the RXB. If, when the branch executes, a misprediction is detected, control is temporarily transferred to the correct target of the branch. Corresponding to Step-3 in the figure, the branch's correct CD instructions are fetched from the I-cache and renamed using the repair rename map, which is initialized from a corresponding branch checkpoint thus ensuring the correct CD instructions have values in the physical register file to begin execution with. When the reconvergent point is encountered again, control is transferred to the branch's CIDD instructions in the RXB, corresponding to Step-4 in the figure. These are also renamed using the repair rename map to establish linkages with producer instructions prior to the reconvergent point. A key point is that the branch's CIDD instructions residing in the RXB do not tie up cycle-critical resources (issue queue entries and physical registers) and are allocated resources only when control is transferred to the RXB, just like instructions that are dispatched from the I-cache. Another key point is that CIDDs' source operands that depend on CIDI instructions cannot be resolved by the repair rename map because the CIDIs' values were most likely freed from the physical register file already, and those that have not been freed are inaccessible by the repair rename map anyway; fortunately, the source values were individually checkpointed previously and are in the RXB with the CIDD instructions.

Loads issue aggressively and are speculative with or without branch mispredictions (Chrysos, et al., 1998). Store-load dependences are also resolved correctly, as we explain in Section 6.7 and Section 6.8.

6.2 Identifying and inserting CIDD instructions into RXB

This section explains how CIDD instructions are identified and inserted into the RXB by the speculative rename map, in a process called poisoning.

6.2.1 Reconvergent point and Influenced Register Set (IRS) predictor

The compiler or a hardware predictor can be used to identify branches' reconvergent points. We use the dynamic reconvergence predictor proposed by Collins et al. (Collins, et al., 2004). We augment the predictor to provide additional information for each branch. First, the predictor keeps track of the maximum path length through a branch's control-dependent (CD) region, among paths that were traversed. This information is useful for guiding when to apply control independence. We select a maximum CD path length above which it is not worthwhile to exploit control independence due to the sheer number of incorrect CD instructions. Second, we add a learning mechanism to collect a branch's influenced register set (IRS). As the predictor monitors retired instructions for reconvergence, it keeps track of logical registers written to after the branch and before reconvergence is detected. The use of confidence ensures repetition, so that enough different paths are traversed through a branch's CD region to yield a representative IRS.

6.2.2 Control-Flow Stack (CFS)

When a branch is dispatched, we must detect its reconvergent point among later instructions as they are dispatched. The reconvergent point marks the beginning of CI instructions, so it is at this point that we need to mark, or "poison", influenced registers (indicated by the branch's IRS) in the speculative rename map.

A novel hardware mechanism called the *control-flow stack* (CFS) detects reconvergent points in the dispatch stage. When a checkpointed branch is dispatched, its reconvergent PC and checkpoint tag (to identify the branch) are pushed onto the CFS top-of-stack.

The next reconvergent point in the dynamic instruction stream is detected by comparing the PCs of newly dispatched instructions to the reconvergent PC at the top-of-stack. If there is a match, then the branch corresponding to the current top-of-stack has reconverged. We know which branch this is via the checkpoint tag at the current top-of-stack. Since the beginning of control-independent instructions has been reached, the branch's IRS is used to poison influenced registers at this time. Poisoning registers is explained in the next section. Finally, the CFS top-of-stack is popped (removed), re-exposing the next reconvergent point to search for.

The CFS can detect cases in which multiple branches have the same dynamic reconvergent point. If the reconvergent PC of a newly dispatched branch matches the reconvergent PC at the CFS top-of-stack, then the new branch and the branch corresponding to the CFS top-of-stack have the same dynamic reconvergent point. They do not have the same dynamic reconvergent point if the call depths of the two branches are different, e.g., due to recursion. We make the test definitive by tracking call depth in the dispatch stage and including call depths in CFS entries. If the new branch's reconvergent PC and call depth match the CFS top-of-stack, then the branches have the same dynamic reconvergent point. In this case, the new branch does not push a new entry onto the CFS, implicitly "merging" with the CFS top-of-stack.

There are three cases in which a branch is forced to inherit the reconvergent point of its encompassing branch region: if the branch does not have a predicted reconvergent PC, if there are no free checkpoints, or if the branch is confidently predicted. The branch corresponding to the CFS top-of-stack is the closest encompassing branch. Thus, the new branch inherits the reconvergent point of its encompassing branch simply by not pushing onto the CFS and instead merging as explained above.

The CFS only needs as many entries as there are checkpoints (16 entries our default configuration). CFS entries of branches that resolve before they reconverge are collapsed away (since they are not popped).

6.2.3 *Poison vectors*

After a branch's CD region is fetched and its reconvergent point is detected by the CFS, we are ready to use the branch's IRS to poison registers and thereby identify CIDD instructions. Each influenced register specified in the IRS must be poisoned.

We provide a 16-bit *poison vector* per entry in the speculative rename map. A logical register is poisoned if one or more bits are set in its poison vector. Moreover, which bits are set indicates which branches a logical register is influenced by. A checkpointed branch is identified by its checkpoint tag. A non-checkpointed branch is identified by the checkpoint tag of the branch from which it inherited its reconvergent point (discussed in Section 6.2.2). Since we use 16 checkpoints in the default configuration, a poison vector has 16 bits in the default configuration.

When a branch reconverges, the poison vector of each influenced register, specified by the IRS, is updated in the speculative rename map. In particular, the poison bit corresponding to the branch's checkpoint tag is set.

CIDD instructions can now be identified during renaming. When an instruction's logical source registers are renamed, the corresponding poison vectors are ORed together. If the ORed vector has any bits set, the instruction is CIDD with respect to one or more branches. Also, the ORed vector overwrites the poison vector of the logical destination register, in the speculative rename map. This propagates poison status for identifying indirect CIDD instructions.

When a checkpoint is freed, the corresponding poison bit is cleared in all poison vectors. Given that all branches associated with the checkpoint are now resolved, no future instructions should be considered CIDD with respect to these branches.

Only the speculative rename map, repair rename map, and checkpoints have poison vectors. Poison vectors in the repair rename map and checkpoints are discussed in Section 6.3.

6.2.4 Inserting CIDD instructions into the RXB

CIDD instructions are inserted into the RXB in program order at the dispatch stage. When a CIDD instruction issues and reads its source values from the physical register file, it replaces its source mappings in its entry in the RXB with the source values (a bit is set within its entry in the RXB to signify that source values have replaced source mappings).

6.3 Misprediction recovery

When a misprediction is detected, the fetch unit temporarily redirects fetching to the correct target of the mispredicted branch. Correct CD instructions are fetched from the instruction cache and renamed using the repair rename map initialized from a checkpoint at the branch. The repair rename map, like the speculative rename map, has its own CFS to detect the reconvergent point again thus detecting the end of the correct CD instructions (its CFS also identifies new nested branch regions). At this point, the branch's CIDD instructions are fetched from the RXB, re-renamed using the repair rename map, and re-injected into the pipeline. Finally, the repair rename map is used to fix up the speculative rename map and checkpoints.

6.3.1 Reconstructing the RXB

The RXB contains CIDD instructions with respect to all unresolved branches. This means the RXB must be reconstructed when recovering from a branch misprediction, as follows.

- *Case A*. There may be instructions from the branch's incorrect CD path in the RXB, that were thought to be CIDD with respect to other prior branches. These have to be removed from the middle of the RXB.
- *Case B*. New instructions from the correct CD path may be CIDD with respect to other prior branches. These have to be inserted into the middle of the RXB.
- *Case C*. Instructions in the RXB that are only CIDD with respect to the branch being serviced should be selectively removed from the RXB, since they will not be revisited again. Instructions in the RXB that are CIDD with respect to other branches (whether or not they are also CIDD with respect to the current branch) must remain in the RXB. Note that these two types of instructions are co-mingled in the RXB.

There is only one solution and it is simple, because it is analogous to initial CIDD identification and insertion into the RXB described in the previous section. The recovery program for the current branch is comprised of the correct CD instructions from the instruction cache and all instructions in the RXB logically after the resolved branch's reconvergent point. (The recovery program is not as efficient as it could be because it has CIDD instructions of other branches that are not also CIDD with respect to the current branch.) *Poisoning of the recovery program via the repair rename map can once again construct the RXB contents*. As a preliminary step, the RXB tail pointer is moved back to the branch (even though the branch may not be in the RXB physically, the branch knows its logical position in the RXB). This naturally takes care of any incorrect CD instructions in the RXB since they will get overwritten by the adjusted tail pointer (*case A*). Then, poisoning the recovery program using the repair rename map will naturally (1) insert new CIDD instructions with respect to prior branches from among the correct CD instructions (*case B*), and (2) insert old CIDD instructions only if they are CIDD with respect to remaining unresolved branches (*case C*).

Since CIDD instructions are concurrently fetched from the RXB (while fetching the recovery program) and inserted into the RXB (while constructing a new recovery program), we need a mechanism to prevent overwriting CIDD instructions in the RXB before they are fetched. We set up a pre-read pointer into the RXB, that points to the first CI instruction with respect to the resolved branch. Since we moved the tail pointer to the branch, the pre-read pointer is logically after the tail pointer. The pre-read pointer is where fetching of CIDD instructions is supposed to begin. If we wait until the correct CD path is fetched, some of the CIDD instructions beginning at the pre-read pointer could get clobbered by the advancing tail pointer. Therefore, using the pre-read pointer, we begin pre-reading CIDD instructions from the RXB right away so that they cannot get clobbered. They are transferred to a Temp Buffer, from which fetching of CIDD instructions will eventually begin (after the correct CD instructions are fetched from the instruction cache).

Figure 33 shows a detailed RXB reconstruction example with two branches, B1 and B2, and respective reconvergent points R1 and R2. Logical positions of B1/R1 and B2/R2 with respect to RXB instructions are indicated with wide black arrows. RXB instructions are labeled with their position # in the dynamic instruction stream. Noncontiguous numbers merely highlight that CIDD instructions are noncontiguous. Instruction x is not numbered because it is an incorrect CD instruction of mispredicted branch B2. Furthermore, instructions are marked with either a rectangle or oval: rectangles are CIDD with respect to B1, ovals are CIDD with respect to B2, rectangle+oval are CIDD with respect to both B1 and B2. Below we step through each of the frames (a)-(g).

(a) Frame (a) shows the initial state of the RXB. B1 has no CD instructions in the RXB since there are no branches prior to it. B1 has four CIDD instructions after R1: 9, x, 16, 20. B2 has one (incorrect) CD instruction, x. Instruction x is not in the RXB because of B2 but rather because it is CIDD with respect to B1. B2 has two CIDD instructions after R2: 18, 20.

(b) In frame (b), mispredicted branch B2 is detected, causing the RXB tail to rollback to just after B2 (instruction x), and the RXB pre-read pointer to initiate at the first CIDD instruction past B2's reconvergent point R2 (instruction 16).

(c) In frame (c), new instructions 11 and 12 – correct CD instructions with respect to B2 – are fetched from the instruction cache (I$) and dispatched for the first time to the issue queue (To IQ). Moreover, instruction 12 is inserted into the RXB because it is CIDD with respect to B1. Instruction 12 is inserted at the RXB tail (which then advances) thereby replacing instruction x. Note also that pre-reading has begun: instruction 16 is transferred to the Temp Buffer so that it is not clobbered by B2's incoming correct CD instructions.

(d) Similarly, in frame (d), we continue fetching and dispatching the remainder of B2's correct CD instructions (13 and 14). Both 13 and 14 are dispatched to the issue queue but only 14 is inserted into the RXB, since 14 is CIDD with respect to B1. Meanwhile we continue pre-reading instructions (18) into the Temp Buffer.

(e) In frame (e), no more instructions are fetched from the I$ because B2's reconvergent point R2 has been reached from the correct CD path. We begin reinjecting and/or recirculating CIDD instructions from the Temp Buffer. Frame (e) shows instruction 16 leaving the Temp Buffer only to be recirculated back to the RXB (CIDD on unresolved B1). It is not reinjected into the issue queue because it is not CIDD on B2 (the mispredicted branch).

(f) However, instruction 18 in frame (f) is reinjected into the issue queue (CIDD on resolved B2) and not recirculated back to the RXB since it is not CIDD on B1.

(g) Finally, in frame (g), instruction 20 is both reinjected into the issue queue and recirculated to the RXB from the Temp Buffer, because it is CIDD on both B1 and B2. Since the Temp Buffer is empty, we are done servicing B2.

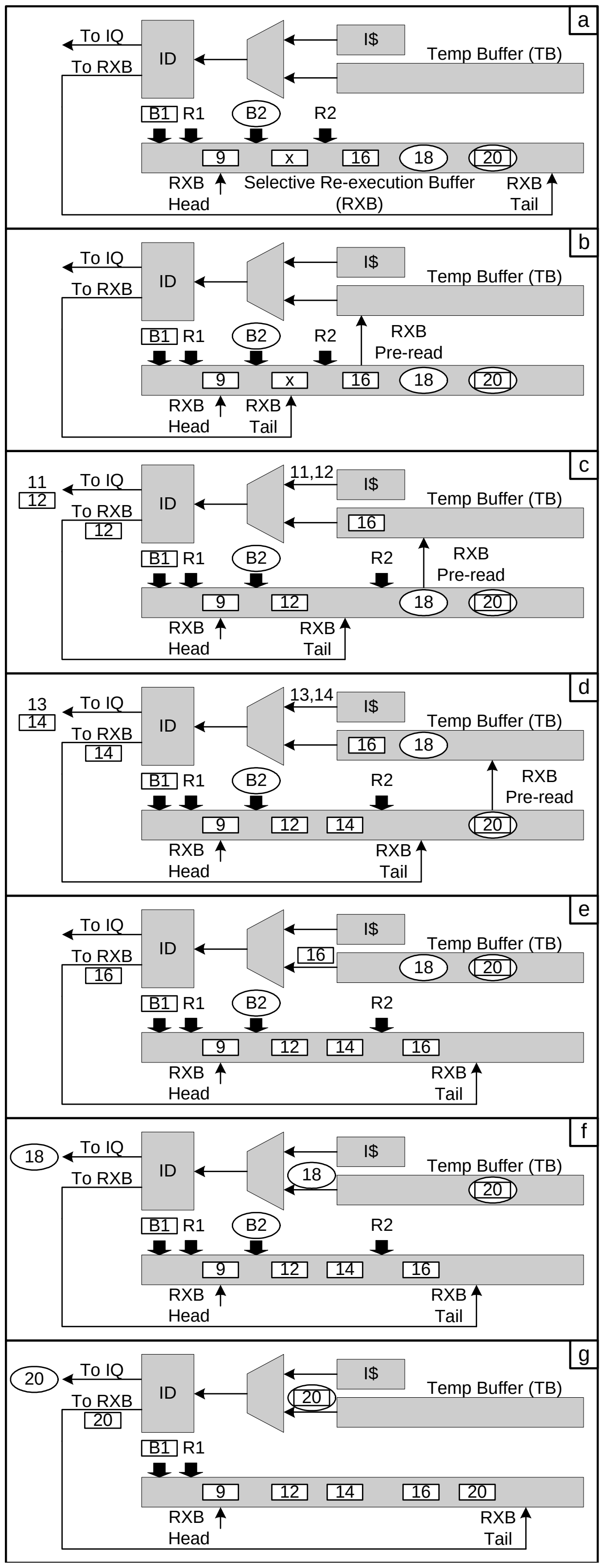


Figure 33. RXB reconstruction example.

6.3.2 Poisoning via repair rename map

The repair rename map's poison vectors are initialized from the mispredicted branch's checkpoint. While fetching the correct CD instructions from the instruction cache and CIDD instructions from the RXB, the poison vectors are managed the same way as described for the speculative rename map (Section 6.2.3), except for a subtle modification. The poison vectors of logical registers that would have been updated by CIDI instructions, simply are not, because they are not observed by the repair rename map. These logical registers represent "holes" in the repair rename map and their poison vectors cannot be referenced by an instruction's source registers. Fortunately, we know two things: (1) the poison vector generated by a CIDI instruction is all 0's because it is not CIDD with respect to any unresolved branch, and (2) a CIDI instruction *is* observed *once* (and only once) in either the speculative rename map (CIDI immediately) or repair rename map (CIDI eventually). So, when a source register of a CIDD instruction references a CIDI production for the first and only time (signaled by an all-0 poison vector in the rename map), a sticky bit ("CIDI_supplied") associated with the source register in the RXB is set to indicate that the source register's poison vector is by definition all 0s. Once CIDI_supplied=1, in future passes, an all-0 poison vector is used instead of referencing an absent poison vector in the repair rename map. Table 5 summarizes poisoning using the repair rename map.

The outcome of poisoning by the repair rename map indicates what to do with each instruction. For correct CD instructions from the instruction cache, the choices are: insert or do not insert into the RXB. For CIDD instructions from the RXB, the choices are: reinject only, insert (i.e., recirculate) only, reinject and insert, or discard. An instruction is inserted into the RXB if poisoning indicates that it is CIDD with respect to any unresolved branches. An instruction is reinjected into the pipeline if poisoning indicates that it is CIDD with respect to the mispredicted branch being serviced.

Table 5. Poisoning using the repair rename map.

	Source operand type	
	CIDI	**CIDD**
Action	Use an all-0 poison vector	Read poison vector from repair rename map

6.3.3 *Reinjecting CIDD instructions*

Only CIDD instructions from the RXB that are CIDD with respect to the branch being serviced are reinjected into the pipeline. These are re-renamed to bind physical registers and thereby facilitate re-execution.

CIDI instructions are absent from re-renaming, just as they were absent from poisoning. Now, additionally, CIDD instructions from the RXB that are not reinjected are also absent from re-renaming. The latter instructions are CIDD with respect to other branches but not with respect to the branch being serviced. They are tantamount to CIDI instructions with respect to the branch being serviced (“implicit” CIDI instructions), and need not be re-executed. As such, they are not re-allocated storage and do not participate in re-renaming.

When re-renaming a source register of a reinjected CIDD instruction, we need to determine if it depends on an explicit or implicit CIDI instruction (the two cases outlined above) versus a CD or reinjected CIDD instruction. If it depends on an explicit or implicit CIDI instruction, then the source value (if available) or source mapping from the RXB is used in lieu of re-renaming, because the repair rename map has a stale name. Otherwise, the correct mapping is obtained from the repair rename map. Table 6 summarizes re-renaming a source operand using the repair rename map.

Table 6. Renaming using the repair rename map.

	Source operand type		
	Explicitly CIDI	**CIDD but implicitly CIDI**	**CIDD**
Action	Read value or mapping from RXB	Read value or mapping from RXB	Read mapping from repair rename map

The source register depends on an explicit CIDI instruction if its CIDI_supplied bit in the RXB is set. The source register depends on an implicit CIDI instruction if its poison vector in the repair rename map does not have the current branch's bit set. Note, it is safe to reference the poison vector because all CIDD instructions in the RXB undergo poisoning. It is only unsafe to reference the poison vector in the case of explicit CIDI instructions, which is why the CIDI_supplied bit is checked first.

The reinjected CIDD instruction is allocated a new physical destination register and updates the repair rename map accordingly.

If a CIDD instruction is both inserted (i.e., recirculated) into the RXB and reinjected into the pipeline, its source registers may be updated in the RXB, analogous to what was described in Section 6.2.4. Specifically, when it redispatches, a re-renamed source register updates the corresponding source mapping in the RXB. When it reissues, it reads values from the physical register file for source registers that did not reuse values from the RXB. These new values replace corresponding source mappings in the RXB.

6.3.4 Merging repair/speculative rename maps

When RXB reconstruction is completed, the repair rename map is logically at the same point in the dynamic instruction stream as the speculative rename map. Some mappings in the speculative rename map have to be repaired using the repair rename map. Specifically, any speculative mapping whose poison vector has the branch’s bit set may be incorrect (it may have changed due to the control-flow adjustment). We simply copy the corresponding mapping from the repair rename map to the speculative rename map. All poison vectors in the repair rename map are copied.

Checkpoint maps are repaired the same way, as the repair rename map resequences through the RXB and reaches checkpoints along the way.

6.4 Writing source values into the RXB

When a CIDD instruction leaves the issue queue and reads its source values from the physical register file (or from the bypass network), it needs to access the RXB entry assigned to it to replace the stored source mappings with the actual source values. However, a CIDD instruction’s RXB entry may change its location as part of reconstructing the RXB during branch misprediction recovery. The RXB entry may reside in the Temp Buffer temporarily, only to be reinserted into a different RXB entry or to be discarded from the RXB altogether (no longer CIDD on any branch). In either case, the in-flight CIDD instructions still in the pipeline have stale RXB entry numbers (RXB tags) that need to be repaired, to prevent corrupting the RXB contents by way of updating the wrong RXB entries.

Furthermore, multiple instances of the same CIDD instruction may be in the pipeline concurrently, waiting for their opportunity to update the shared RXB entry. This situation may arise because an original dispatched instance has not issued by the time a branch misprediction starts servicing or because an instruction that is CIDD on multiple branches is injected multiple times due to servicing multiple branch mispredictions independently. In either case, only the last dispatched instance of a given CIDD instruction needs to update the RXB entry with source values, since the last instance has the most up-to-date source mappings that reflect the current state of the processor.

To overcome these challenges, a solution needs to be implemented that can fulfill two requirements:

1) Enable valid CIDD instructions in the pipeline to access their RXB entries even in the presence of RXB reconstruction (RXB-entry recirculation).
2) Invalidate the RXB tags of some CIDD instructions in the pipeline, in reaction to freeing some RXB entries or reinjecting duplicate copies of the CIDD instructions into the pipeline.

We propose using an indirection table (IT) in conjunction with the RXB to fulfill both of these requirements. The indirection table contains the actual RXB/TB mapping, a valid bit indicating if the mapping is valid or not, and a Temp Buffer bit indicating if the RXB entry is in the Temp Buffer. As CIDD instructions (either original CIDD or reinjected CIDD) dispatch into the pipeline, they are allocated an entry in the IT in addition to the RXB entry. The IT entry is initialized as being valid and contains the newly allocated RXB entry number. In-flight CIDD instructions carry with them only their IT entry number to access the RXB entry, instead of the actual RXB entry number (since it may become stale). This level of indirection allows us to insulate in-flight CIDD instructions from changes in the RXB.

Next, we show how the RXB's IT is managed under different situations:

- Scenario 1: dispatching a CIDD instruction for the first time:
    1) Allocate a new RXB entry and a new IT entry.
    2) Initialize the new IT entry to point to the new RXB entry.
    3) Initialize the new RXB entry's IT index to point to the new IT entry.
- Scenario 2: writing source values into the RXB/TB:
    1) Access IT entry.
    2) If valid bit is not set, free IT entry and discard the source values.
    3) If valid bit is set:
        a. Read out mapping and entry location bit (in RXB or in TB).
        b. Free the IT entry.
        c. Update the RXB or TB with the source values and clear the IT index stored in the RXB or TB entry.
- Scenario 3: moving entry from the RXB to the TB:
    1) If the RXB entry has a valid IT index, set corresponding IT entry's Temp Buffer bit and overwrite the mapping with the new location in the TB.
    2) If the RXB entry does not have a valid IT index, do nothing.
- Scenario 4: recirculating an RXB entry (moving entry from TB back to RXB) with no reinjecting:

1) If the TB entry has a valid IT index, clear corresponding IT entry's Temp Buffer bit and overwrite the mapping with the new location in the RXB.
2) If the TB entry does not have a valid IT index, do nothing.

- Scenario 5: recirculating an RXB entry (moving entry from TB back to RXB) while reinjecting a new instance of a CIDD instruction:
  1) If the TB entry has a valid IT index, clear corresponding *old* IT entry’s valid bit (this prevents old CIDD instances that are still in-flight from wrongly updating the RXB/TB). If the TB entry does not have a valid IT index, do nothing.
  2) Allocate a new IT entry for the new CIDD instruction instance being reinjected.
  3) Initialize the new IT entry to point to the existing RXB entry, i.e., the shifted location in the RXB.
  4) Overwrite the existing RXB entry’s IT index to point to the new IT entry.

The size of the RXB’s IT is equal to the maximum number of in-flight CIDD instructions (in the issue queue or in the backend pipeline). The maximum number of CIDD instructions in overall is equal to the size of the RXB. The maximum number of in-flight instructions is equal to the size of the issue queue plus the number of instructions in the backend pipeline (#backend pipeline stages * processor width). Hence, the size of the IT is the lesser number between the maximum number of CIDD instructions and the maximum number of in-flight instructions.

IT size = Minimum (

(RXB size),

(Issue queue size + (#backend pipeline stages * processor width))

)

The IT size presented above actually covers the worst-case scenario. However, in practice, only a small percentage of in-flight instructions tends to comprise CIDD instructions. Hence, the size of the IT can be much smaller.

IT size optimized = Minimum (

(RXB size),

(% CIDD * (Issue queue size + (#backend pipeline stages * processor width)))

)

6.5 Conventional recovery

If a branch misprediction is detected before the fetch unit has reached the branch's reconvergent point, then there is no need to transfer control to the repair rename map and RXB, as there are no CI instructions with respect to the branch yet. This scenario is easily detected by checking if the mispredicted branch has not yet popped the CFS (not reconverged). In this case, the speculative rename map is simply restored to the checkpoint corresponding to the mispredicted branch as in conventional recovery.

6.6 Servicing multiple branch mispredictions

TCI supports servicing new mispredictions concurrently with the one being serviced, if the new mispredictions are logically after the repair rename map. A new misprediction will begin servicing when the repair rename map logically reaches it, in a natural continuation of

RXB reconstruction. After fetching the correct CD instructions of the new misprediction, CIDD instructions of both the initial and new mispredictions are reinjected concurrently. If a new misprediction is logically before the repair rename map, we wait until the initial RXB reconstruction completes before servicing the new misprediction; however, an earlier misprediction that has not reconverged is serviced immediately via conventional recovery.

6.7 Store and load queues

Stores and loads issue out-of-order in the pipeline. The memory dependence predictor (e.g. store sets (Chrysos, et al., 1998)) is a speculative optimization to enable some loads to issue speculatively yet confidently. Ultimately, memory dependencies between loads and stores are enforced by the load/store queue (LSQ). A traditional LSQ needs to maintain order between all stores and loads for correct store-load forwarding and load violation detection. Hence, the order of the LSQ must be repaired when exploiting control independence to recover from a branch misprediction, since loads and stores may be removed and inserted in the middle of the window.

The LSQ can be repaired by leveraging the same reconstruction technique used to adjust the RXB after a branch misprediction. The modified LSQ would need its own TB to assist in shifting the loads and stores into their new locations. In addition, the modified LSQ would need its own indirection table (IT). The IT is necessary to insulate loads and stores in the pipeline from the LSQ repair process. Repairing the order of the LSQ in this way may degrade the performance of the system. The LSQ holds many more instructions than the compressed RXB, which could extend the branch misprediction recovery process. To overcome this problem, we need to reduce the time needed to repair the LSQ.

Fortunately, many processors implement the ordered LSQ as two separate ordered queues: a store queue (SQ) and a load queue (LQ). Order between the two queues is maintained using pointers. Each LQ entry knows which entry in the SQ it is logically after. Each SQ entry knows which entry in the LQ it is logically after. This separation allows repairing the LSQ twice as fast, since both the SQ and the LQ are repaired in parallel. However, this implementation also requires us to split both the TB and IT into two structures, one for the SQ and another for the LQ. The SQ indirection table (SQ-IT) insulates the LQ entries' pointers from changes during SQ reconstruction. Therefore, pointers in the LQ entries point to the SQ-IT instead of the SQ entries directly. This function is also carried out by the LQ indirection table (LQ-IT) to insulate the SQ entries' pointers from changes during LQ reconstruction.

The main reason for separating the SQ from the LQ is due to the different functions they perform. The SQ has two main functions: forwarding memory values to loads and committing stores to the cache in program order. On the other hand, the main function of the LQ is to detect memory dependence violations due to the possibility of a load receiving a wrong value. This separation leads to higher efficiency since we avoid accessing a large unified LSQ when accessing one of the smaller queues is sufficient. Note, the SQ needs to be fast (the store-load forwarding function) as it affects the performance of the processor; however, the LQ is only needed to guarantee correctness which is not as time sensitive.

We observe that to fulfill all the requirements of the LSQ, order only needs to be maintained between store instructions and between store and load instructions. Order between load instructions is not required. Relaxing the order between load instructions enables us to design a more efficient LSQ that is compatible with control independence. We propose using a partially ordered LSQ (POLSQ), which is the combination of an ordered SQ and an unordered LQ to achieve this objective. Figure 34 shows an example of how loads and stores would be organized in an ordered unified LSQ, a split ordered SQ and ordered LQ, and our new POLSQ. Notice that loads are only ordered with respect to stores in the POLSQ.

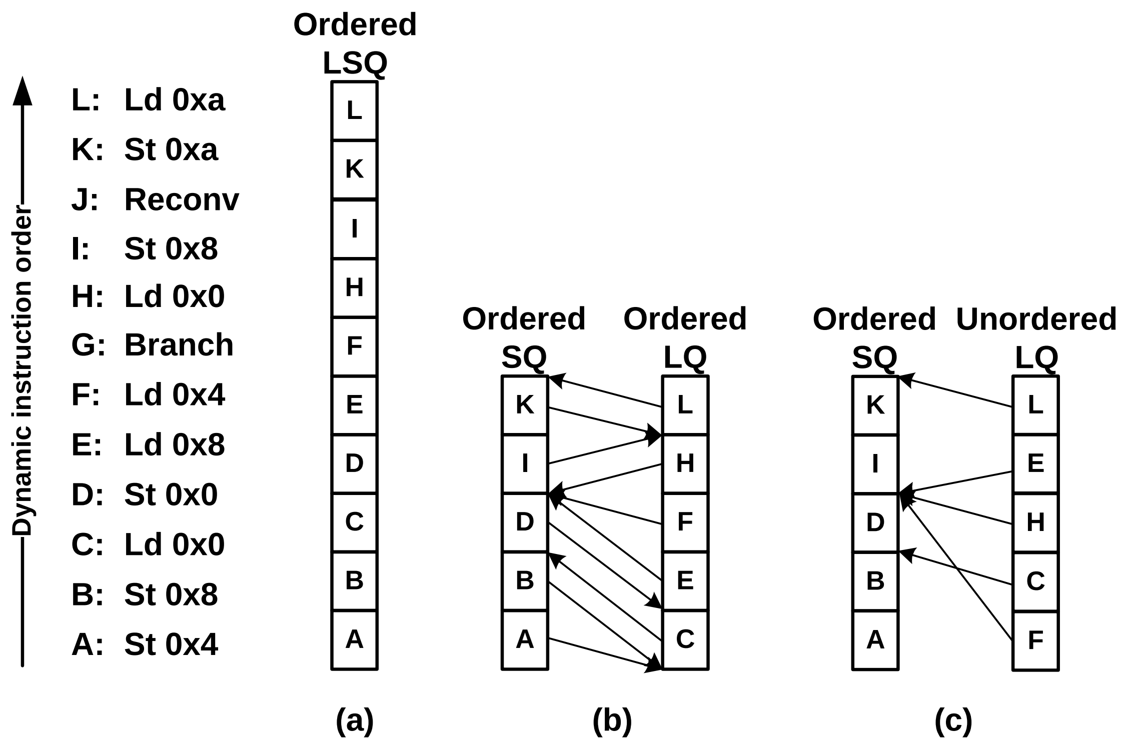


Figure 34. Example showing the relationship between stores and loads in the LSQ: (a) Ordered unified LSQ. (b) Ordered split SQ/LQ. (c) Partially ordered LSQ.

In the POLSQ, the SQ will preserve the correct program order required for both store-load data forwarding and committing stores to the cache in order. Hence, the SQ needs to be reconstructed during branch misprediction recovery. This is achieved via the store queue temporary buffer (SQ-TB) for recirculating the SQ entries and the store queue indirection table (SQ-IT) to insulate in-flight stores in the pipeline and pointers in the unordered LQ from SQ changes. Entries in the unordered LQ will only preserve order with respect to stores using the store pointers (SQ-IT index) in each LQ entry. The LQ no longer needs to be reconstructed after a branch misprediction since it is unordered. Therefore, it no longer needs the load queue temporary buffer (LQ-TB) or the load queue indirection table (LQ-IT).

Figure 35 shows the POLSQ structures in more detail. In addition, the content of the structures reflects the example in Figure 34. Elements with gray borders are additions to a conventional ordered split SQ/LQ design. Similar to the ordered split SQ/LQ design, both the unordered LQ and ordered SQ have an address CAM port for matching on addresses. Note, the address CAM port of the unordered LQ does not provide ordering information. Order is determined indirectly leveraging the SQ order. In the POLSQ, the unordered LQ also has an additional CAM port for matching on the SQ-IT index. This CAM port is used to free entries in the LQ either due to committing these instructions or due to squashing instructions as part of branch misprediction recovery. Notice, the SQ also contains the branch and reconvergent point instructions. These additional instructions are needed to be able to reconstruct the SQ correctly after a branch misprediction.

In the POLSQ, store-load forwarding is very similar to the conventional ordered split ordered SQ/LQ design. In both designs, the load instruction accesses the ordered SQ and performs a search on all stores logically before the load for a matching address. If one or more stores match the load’s address, then the value of the youngest store (closest to the load) is forwarded to the load. However, if no store address matches then the value is loaded from the data cache. POLSQ requires an additional step before performing store-load forwarding. The SQ-IT is accessed using the load’s SQ-IT index to find the store entry (SQ index) to search before. This extra step is a consequence of using the SQ-IT to insulate loads from the possible shifting of SQ entries.

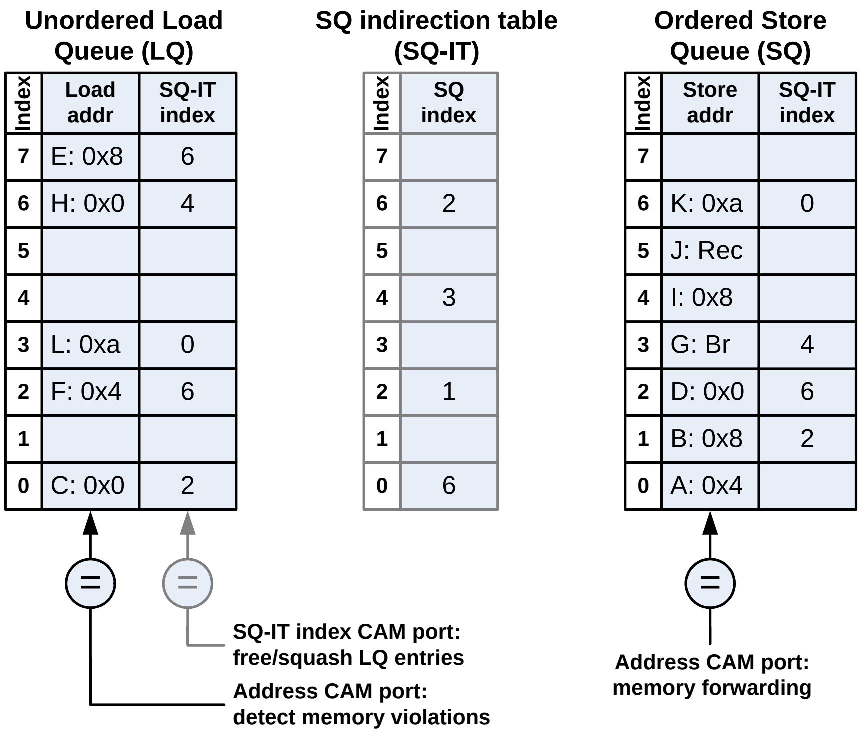


Figure 35. Structures of the POLSQ (excluding SQ-TB), with contents corresponding to the running example.

Figure 36 gives an example of store-load forwarding using POLSQ. When a load instruction (E in this example) computes its address, the store-load forwarding process is initiated. The first step involves accessing the SQ-IT with instruction E’s buffered index (SQ-IT index = 6). This produces the current SQ index that the load is logically after in the SQ (SQ index = 2). The second step uses the computed address of the load (0x8) to find stores with a matching address in the SQ, and uses the SQ index from step 1 to find the closest prior matching store instruction. In this example, we notice that two store instructions match on the address of the load (store B and store I), but only one of the stores is actually before the load's SQ index (store B). Hence, the load will be forwarded the value of the store instruction B.

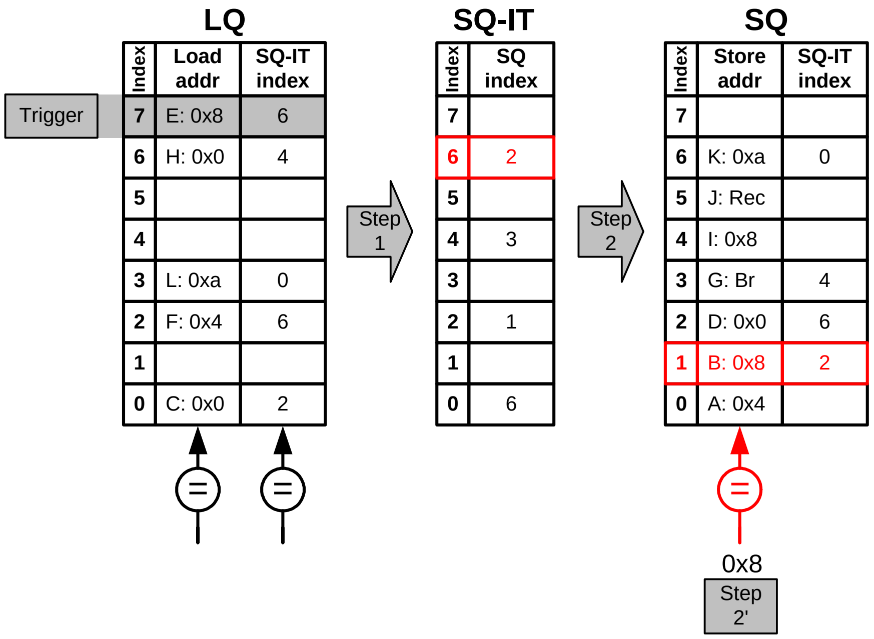


Figure 36. Store-load forwarding using the POLSQ.

Since loads execute speculatively with respect to prior unresolved stores, memory dependence violations may occur leading to incorrect program execution. Detecting and recovering from load violations is required for correct execution. Load violations are detected by comparing completed store addresses against the load queue. If the address of a completed store matches the address of any load in the LQ, then a violation may have occurred. However, since the LQ is unordered, additional steps are needed to verify a load violation has occurred. This is accomplished by comparing the order of the potentially offending loads to the order of the store. First, the potentially offending loads will access the SQ-IT, to determine their current logical positions in the SQ. The SQ indices are then used to confirm or dismiss the potential violations. Loads located before the store are false violations. However, loads located after the store are true violations needing recovery.

Figure 37 gives an example of detecting a memory dependence violation using the POLSQ. Assume that all loads in the LQ have executed speculatively. When store A computes its address, this triggers the need to check for possible memory dependence violations. The first step is to search the LQ for any load addresses that match the store's address (0x4). The search shows that load instruction F is a match and, hence, a potential violation. Next, we try to confirm the exception. The second step is to access the SQ-IT to determine the load's current location in the SQ (SQ index = 2). Finally, in the third step, we compare the load's SQ index with the store's SQ index. We find that the load instruction F is after the store instruction A. Therefore, the violation of load instruction F is confirmed and we need to recover to ensure correct execution.

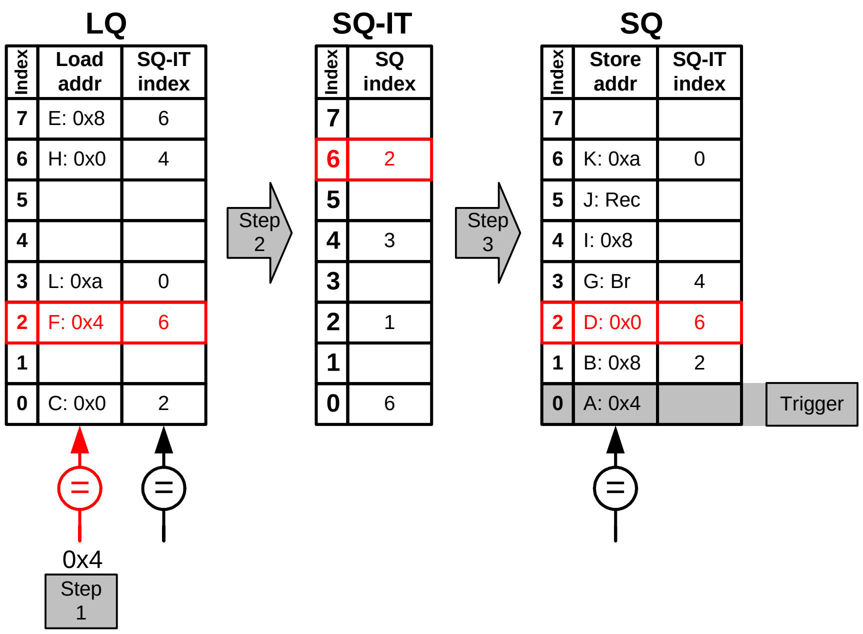


Figure 37. Detecting a memory dependence violation using the POLSQ.

POLSQ entries are freed at retirement or during branch misprediction recovery. The ordered SQ frees its entries in program order at retirement or from the middle of the SQ as part of the SQ reconstruction. On the other hand, the LQ is unordered and relies on the SQ to free its entries. When a SQ entry is freed (either at retirement or during branch misprediction recovery), the SQ entry's SQ-IT index is broadcasted to the LQ, freeing matching LQ entries. Note, when the SQ is empty (i.e., the window does not contain any branches, reconvergent points, or stores), dispatched loads are not inserted into the LQ. These loads are guaranteed not to cause load violations and require no store-load forwarding.

6.8 Branch-sets and CIDD loads

Loads issue speculatively and memory dependence violations are detected via the POLSQ. A memory dependence predictor (store-set predictor (Chrysos, et al., 1998)) is used to stall some loads when a predicted conflicting store is in the window. This reduces memory dependence violations.

A conventional store-set predictor works for stores currently in the window. Unfortunately, servicing a branch misprediction may change the stores in the window (remove incorrect CD stores, insert correct CD stores, or re-execute some CIDD stores) observed previously by the store-set predictor, possibly introducing memory dependence violations.

To reduce memory dependence violations caused by changed stores in the window, we introduce a new "branch-set" predictor. When the POLSQ detects a load violation, we determine the mispredicted branch that influenced the conflicting store (the store is CD or CIDD with respect to the branch). This is used to maintain branch-load dependence information (branch-sets), i.e., the load is considered CIDD with respect to the branch via memory dependencies. Hence, branches become proxies for potentially conflicting stores that they influence, whether the stores are currently in the window or not.

When a load is dispatched, the branch-set predictor will predict if there are or will be any potentially conflicting stores in the window – whether current CD stores, potential late CD stores, or current CIDD stores that may re-execute – and marks the load as being a CIDD load. Predicted CIDD loads and their dependents are then copied into the RXB like normal CIDD instructions. When a mispredicted branch is serviced, its CIDD loads will be re-injected into the pipeline, allowing them the opportunity to re-access the store-set predictor as part of memory disambiguation.

6.9 Load violation recovery

The processor must detect and recover from load violations. Load violations are detected by the POLSQ and indicate failures by the branch-set and store-set predictors.

Although CIDD loads are present in the RXB for an orthogonal reason (to allow them the opportunity to re-access the store-set predictor after a branch misprediction, as described in Section 6.8), violating CIDD loads are in the RXB and can take advantage of the RXB's selective recovery capabilities to efficiently recover from their violations. To do so, the violating CIDD loads are simply re-injected into the pipeline along with their dependent instructions. Notice that the RXB is a general selective re-execution mechanism that can cover any loads if we choose to insert arbitrary loads and their dependents into the RXB.

On the other hand, violating CIDI loads cannot selectively recover, since they are not present in the RXB. Recovering from CIDI load violations is delayed until retirement (to avoid servicing a speculative load violation), at which point the entire pipeline is flushed, the same as an exception.

6.10 Reconvergence predictor misinformation

The reconvergence predictor may provide a flawed reconvergent PC, incomplete IRS, or misleading CD path length for a branch. Inaccuracies are detected when fetching CD instructions of the branch. If inaccurate information is detected during the first pass through the CD region, it can be amended. If detected during the second pass (repairing mispredicted branch), it is handled by forgoing control independence. We call the latter “downgrades” (downgrade to conventional recovery). The frequency of downgrades is reported in results.

An incomplete IRS is detected by observing logical destination registers that are not specified in the IRS. If in the first pass, the IRS can be updated so that it is more accurate when poisoning begins at the reconvergent point. However, if in the second pass, we know that the branch’s CIDD instructions in the RXB are not sufficient: some needed CIDD instructions were not poisoned earlier due to the incomplete IRS.

Flaws in a branch’s predicted reconvergent PC or maximum CD path length are detectable when the branch does not reconverge within the maximum number of allowable CD instructions. If in the first pass, it is handled by popping the CFS top-of-stack, implicitly merging the branch with its encompassing branch which may have better luck reconverging. If in the second pass, the branch downgrades to conventional recovery.

Table 7. Benchmark statistics and Base/Perfect results.

Benchmarks	L2 load miss/1k instructions		Branch misp. /1k instructions		Base IPC				Perfect %IPC improvement			
					4-issue		8-issue		4-issue		8-issue	
	Base	TCI	Base	TCI	IQ32	IQ64	IQ32	IQ64	IQ32	IQ64	IQ32	IQ64
bzip2-program-ref	2.73	2.74	12.74	12.17	1.57	1.60	1.83	1.91	115%	124%	168%	208%
compress95-bigtest-ref	0.31	0.31	10.01	9.92	1.60	1.62	1.80	1.89	98%	120%	119%	171%
crafty-ref	0.06	0.06	5.67	6.17	2.41	2.43	3.11	3.33	55%	61%	81%	108%
gap-ref	0.99	1.04	2.18	2.27	2.86	2.95	3.62	3.96	20%	24%	26%	33%
gcc-expr-ref	0.11	0.12	4.99	5.60	2.36	2.38	3.02	3.13	46%	50%	66%	81%
go95-5stone21-ref	0.02	0.02	20.65	21.21	1.21	1.21	1.32	1.33	186%	205%	254%	342%
gzip-graphic-ref	0.73	0.73	10.42	10.56	1.63	1.64	1.89	1.94	102%	113%	128%	178%
ijpeg95-specmun-ref	0.63	0.63	4.67	4.83	2.51	2.54	3.37	3.59	42%	44%	66%	76%
li95-ref	0.00	0.00	5.24	6.42	2.42	2.45	3.05	3.22	52%	59%	79%	96%
mcf-ref	128.13	128.87	5.02	4.75	0.10	0.10	0.10	0.11	1%	2%	1%	1%
parser-ref	0.04	0.04	7.69	7.66	1.75	1.85	1.99	2.19	53%	71%	61%	90%
perlbmk-diffmail-ref	0.04	0.04	2.34	2.40	2.97	3.00	4.21	4.45	25%	28%	38%	46%
twolf-ref	0.02	0.03	13.43	16.59	1.36	1.41	1.49	1.58	86%	116%	101%	149%
vortex-two-ref	0.97	0.99	0.29	0.30	3.54	3.63	5.18	5.66	3%	3%	4%	5%
vpr-route-ref	5.48	6.91	9.98	9.62	1.18	1.24	1.32	1.44	73%	95%	73%	97%

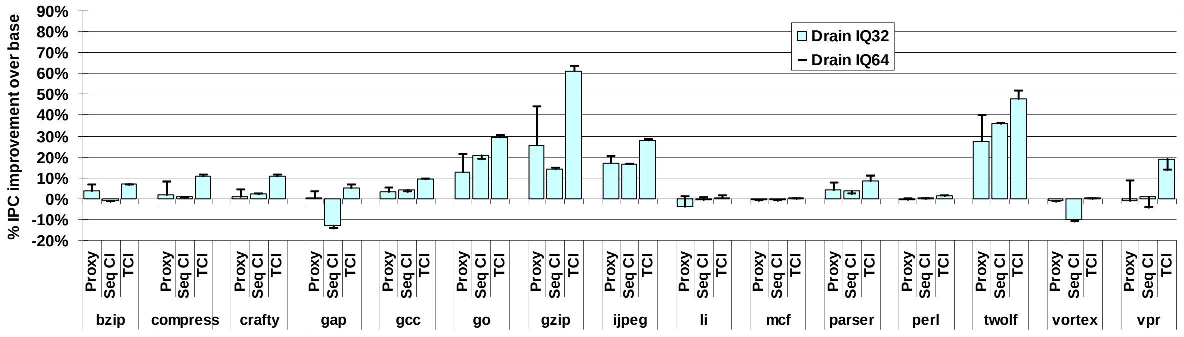


Figure 38. Performance improvement for 4-issue pipeline.

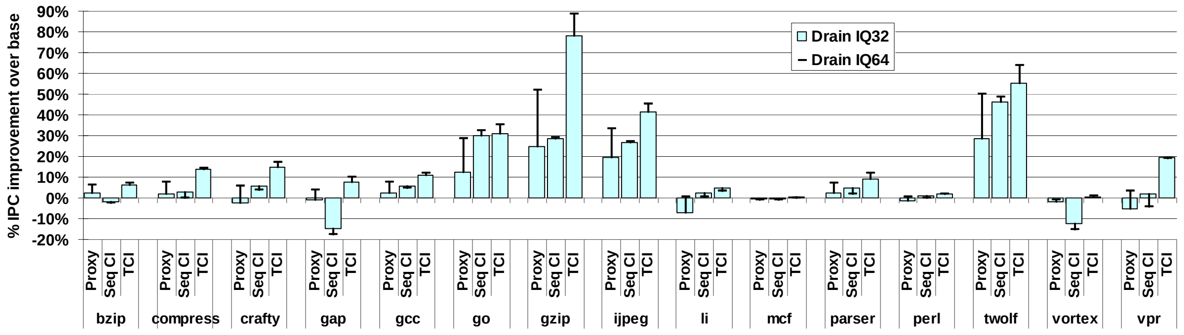


Figure 39. Performance improvement for 8-issue pipeline.

6.11 Results

We present performance results for five models: *Base*, *Proxy*, *Seq CI*, *TCI*, and *Perfect* (the baseline with perfect branch prediction). *Proxy*, *Seq CI*, and *TCI* leverage the *Drain IQ* re-execution substrate (see Section 5.2.3). Table 7 shows the IPCs for *Base* for 4-issue and 8-issue pipelines with 32-entry and 64-entry issue queues. IPC improvement of *Perfect* over *Base* is also shown in Table 7.

6.11.1 Performance and analysis

Figure 38 shows the performance improvement of the various models over *Base*, for 4-issue pipelines with 32-entry and 64-entry issue queues. The 64-entry issue queue results are shown as error bars with respect to the 32-entry bars. *TCI* improves IPC by up to 61% (64%) over *Base* with a 32-entry (64-entry) issue queue. The average IPC improvement of *TCI* over *Base*, across all benchmarks, is 16% for both issue queue sizes.

Figure 39 shows corresponding IPC improvements over *Base* for 8-issue pipelines. The maximum improvement of *TCI* over *Base* increases to 78% (88%) for a 32-entry (64-entry) issue queue, as the opportunity cost of mispredictions is higher for the wider pipeline. On average, *TCI* achieves 20% (22%) IPC improvement over *Base* for a 32-entry (64-entry) issue queue.

TCI consistently and significantly outperforms *Seq CI*, making clear that resequencing all CI instructions after a misprediction does not fully capitalize on control independence opportunity. Furthermore, as a consequence of limiting the window to the size of the RXB, *Seq CI* degrades performance on some benchmarks with respect to the ROB-free *Base*.

Proxy is not resource efficient. As seen in Figure 38 and Figure 39, for the 32-entry issue queue, *TCI* outperforms *Proxy* in all benchmarks. In some benchmarks (e.g., li, vpr), *Proxy* degrades with respect to *Base* as a result of issue queue pressure caused by proxy and CIDD instructions. The average gain for *Proxy* drops from 11% to 6% on a 4-issue pipeline when the issue queue size is reduced from 64 to 32. In contrast, *TCI* and *Seq CI* are less sensitive to the issue queue size.

To understand the performance improvements of *TCI*, we refer to measurements in Table 7 (L2 load misses per 1000 instructions, branch mispredictions per 1000 instructions) and Figure 40. The latter provides a breakdown of branch mispredictions. Some mispredictions are not covered because they have a maximum CD path length that exceeds our chosen threshold of 256 (Non-CI Br) or they resolve before reconverging. For some mispredictions, control independence is attempted (CI Br) but it fails due to downgrade scenarios, two of which are (i) incomplete IRS (IRS downgrade) and (ii) exceed temp buffer (TB downgrade) thereby preventing RXB expansion. Control independence cannot be exploited in these cases. Due to this, in some benchmarks where branch misprediction rates are fairly high, *Perfect* shows great promise but *TCI* cannot exploit enough control independence resulting in more modest performance gains (e.g., bzip, compress).

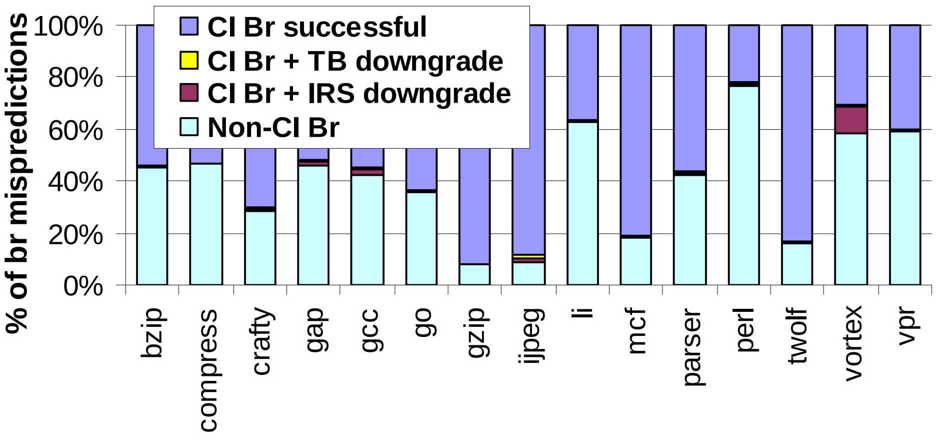


Figure 40. Breakdown of branch mispredictions.

To not artificially favor misprediction-tolerance, we chose the high quality perceptron predictor (Jimenez, et al., 2001). Notice in Table 7 branch misprediction rates for *TCI* are typically higher than for *Base*. This is mainly due to gaps in global history (branches in mispredicted CD regions are omitted from global history used by future branches). We found the perceptron predictor to be relatively more resilient to history gaps than gshare. Further, *TCI* can tolerate some extra mispredictions.

We analyze the 64-entry issue queue results by grouping benchmarks based on branch misprediction rates (Table 7) and control independence coverage (CI coverage) (Figure 40):

- Group A (bzip, compress, go, gzip, twolf, and vpr): High misprediction frequency (9 to 21/1K inst.). Gzip and twolf post significant speedups due to high CI coverage (92% and 83%): 64% and 52% on 4-issue, and 88% and 64% on 8-issue. Go posts a medium speedup: 30% for 4-issue and 35% for 8-issue. Though it has the highest branch misprediction frequency, benefits are limited by medium CI coverage (64%), leaving about 7.6 mispredictions uncovered per 1000 instructions. For bzip, compress, and vpr, CI coverage is moderate (54%, 54%, and 40%), leading to moderate speedups: 7%, 11%, and 14% for 4-issue, and 7%, 14%, and 19% for 8-issue.

- Group B (crafty, gcc, ijpeg, li, and parser): Moderate misprediction frequency (4 to 8/1K inst.). For crafty, gcc, ijpeg, and parser, CI coverage is medium to high (55%-88%), yielding modest speedups: 11%, 10%, 28%, and 11% on 4-issue, and 17%, 12%, 45% and 12% on 8-issue. Li shows low speedups (1-3%) due to its low CI coverage (37%). In li, most mispredicted branches resolve before fetching their reconvergent points.
- Group C (gap, perl, and vortex): Low misprediction frequency (less than 3/1K inst.). Group C does not benefit from *TCI* due to excellent accuracy in the simulated regions, yielding performance close to *Perfect*.
- Group D (mcf): Moderate misprediction frequency, but very high L2 miss rate. For mcf, the simulated region is dominated by a high frequency of serialized L2 misses, as shown in the second column of Table 7. Despite high CI coverage (81%), the penalty of branch mispredictions is masked since they occur in the shadow of L2 misses. This is confirmed by the negligible gains for *Perfect*.

6.11.2 Instruction breakdown

Figure 41 characterizes retired instructions in the context of branch mispredictions. SBM ("shadow of branch misprediction") refers to control independent instructions that are logically in the window when a prior misprediction is detected. (In *TCI*, these are preserved whereas *Base* squashes and re-fetches them.) In contrast, instructions before mispredictions or instructions fetched after a misprediction has initiated servicing, are not considered to be in the shadow of a branch misprediction (Non-SBM). SBM instructions represent control independence opportunity, Non-SBM do not.

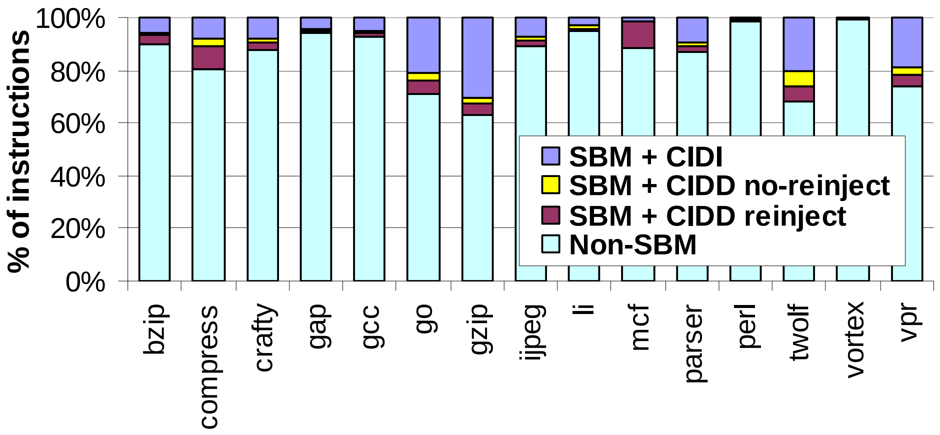


Figure 41. Breakdown of all instructions.

SBM instructions are broken down further into those that were inserted into the RXB (CIDD) and those that were not (CIDI). Among those that were inserted into the RXB, we indicate if they had to be reinjected (CIDD reinject) or not (CIDD no-reinject). SBM+CIDD reinject occurs when the instruction is CIDD with respect to the mispredicted branch (must re-execute). SBM+CIDD no-reinject occurs when the instruction is not CIDD with respect to the mispredicted branch, but rather a different correctly predicted branch. Thus, SBM+CIDD no-reinject is tantamount to SBM+CIDI with respect to the misprediction.

Summing up, the top two classes in Figure 41 (SBM+CIDD no-reinject, SBM+CIDI) represent savings compared to conventional (full) recovery. Benchmarks in Group A and Group B have the largest percentages of these misprediction-independent instructions (7%-33% for Group A and 4%-11% for Group B). Their speedups in Figure 38 and Figure 39 correlate well with their percentages of saved instructions.

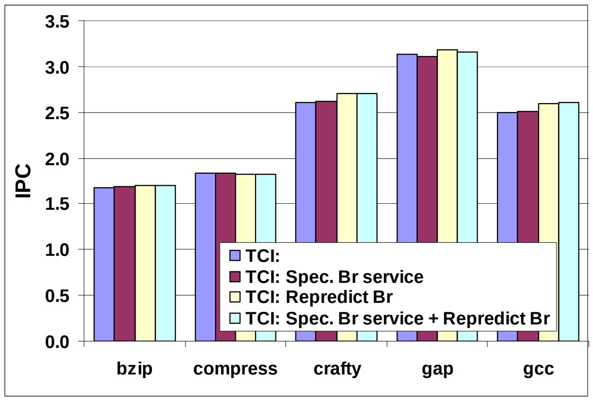
6.11.3 Branch prediction and branch misprediction servicing policies

When using conventional branch recovery, all CI instructions after the branch are squashed are re-fetched. During re-fetching, the CI branches are re-predicted with a repaired global history register (GHR) that reflects the corrected branch and new branches in its correct CD region. This process may overturn some initial predictions, eliminating some branch mispredictions in the re-fetched CI region. On the other hand, control independence implementations keep CI instructions in the window after detecting a branch misprediction; hence, CI branches do not get to be predicted with the repaired GHR. This may introduce additional branch mispredictions in control independence implementations compared to conventional processors. To avoid these additional branch mispredictions, we choose a branch predictor that is somewhat tolerant of an incomplete GHR. In our studies, we have found that the perceptron branch predictor is superior to the gshare branch predictor in its tolerance to the imperfect GHR. Hence, we incorporate the perceptron branch predictor in all our runs. In addition, while reconstructing the RXB, we re-predict branches that have not been truly resolved yet (i.e., branches in the RXB). This allows us to mimic the branch re-prediction process in conventional processors and detect possible mispredictions early (Rotenberg, et al., 1999).

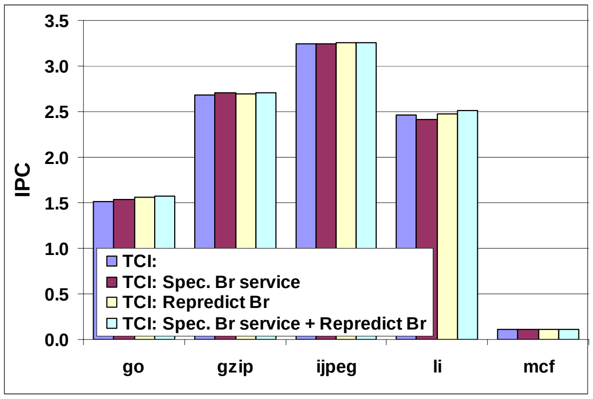
In conventional processors, mispredicted branches are serviced immediately once their outcomes are known. This leads to the highest performing implementation. This is not always true in control independence implementations. Since CI branches may execute multiple times with incorrect source operands from earlier branch mispredictions, it may be better to wait until the source operands are truly stable (CIDI) before servicing the confirmed branch misprediction. Hence, we are faced with either allowing branches to execute speculatively with whatever data they have, with the threat of false mispredictions (Rotenberg, et al., 1999), or delaying their execution until we can verify that their sources are correct (the poison vector is clear).

In TCI, we found that the best combination happens to be (1) allowing branches to execute speculatively and (2) only re-predicting branches that have not executed or that have executed with unconfirmed data. In Figure 42, we present results for four models. The first does not service branches speculatively and does not re-predict branches at all (TCI:). The second only allows branches to be serviced speculatively (TCI: Spec. Br service). The third only re-predicts branches that have not executed or have executed with unconfirmed data (TCI: Repredict Br). The fourth services branches speculatively and re-predicts branches that have not executed or that executed with unconfirmed data (TCI: Spec. Br service + Repredict Br).

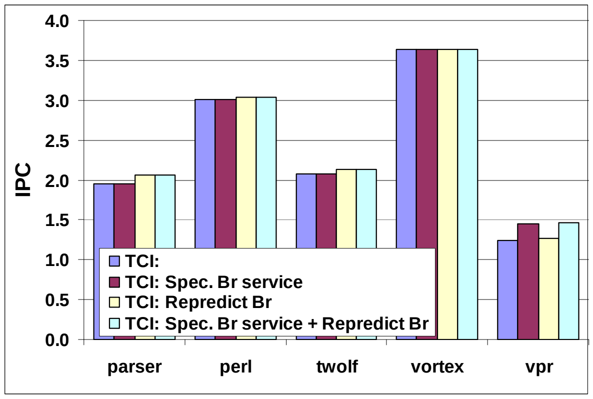
Servicing branches speculatively improves performance most of the time, as can be seen in the harmonic mean IPC presented in Figure 42(d). However, it degrades performance in some benchmarks, such as gap and li. Branch re-prediction improves performance in all benchmarks except compress, where there is a slight degradation. Using both these techniques together further improves performance. The two techniques can be combined, resulting in higher performance than using either technique separately.



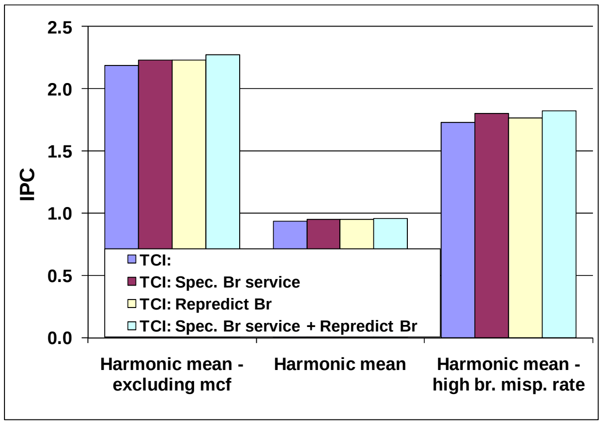
(a)



(b)



(c)



(d)

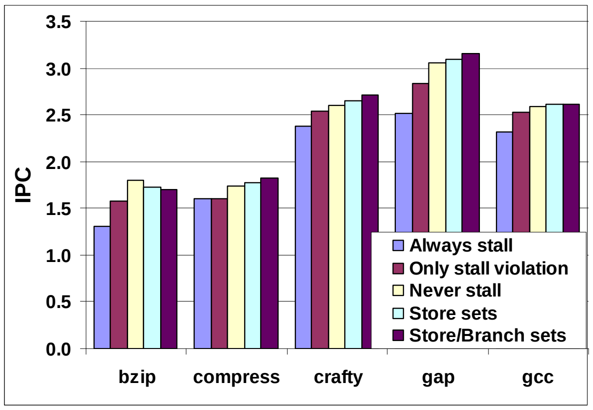
Figure 42. Branch prediction and branch misprediction servicing policies.

6.11.4 Memory dependence predictor

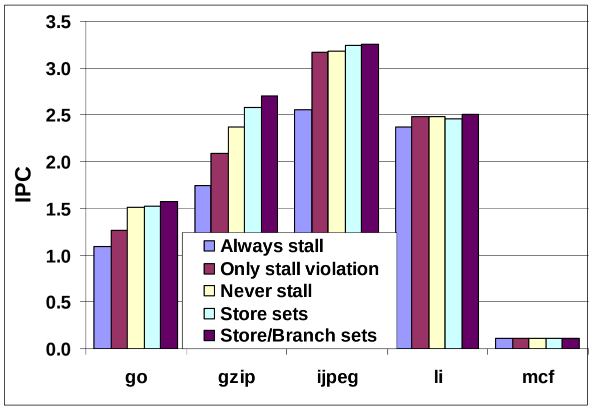
In TCI and other control independence implementations, predicting memory dependencies faces additional challenges. Unlike processors with conventional branch misprediction recovery, control independence implementations have to deal with holes in the middle of the window caused by branch mispredictions. These holes may introduce wrong store instructions from the wrong CD region or delay correct stores from the correct CD region. Normal memory dependence predictors are not designed to deal with these holes in the instruction window.

In this section, we investigate different memory dependence predictors and their impact on the performance of TCI. The first predictor is conservative and chooses to stall all load instructions until all prior stores have computed their addresses (Always stall). The second predictor allows loads to bypass unresolved stores, but if any load causes a memory violation, then, for the remainder of execution, this load will have to stall until all prior stores have computed their addresses (Only stall violation). The third predictor is very aggressive and allows all loads to bypass unresolved stores freely. The next two predictors are based on the store-set predictor that monitors relationships between stores and loads. The basic store-set predictor is studied (Store sets), as well as a modified store-set predictor that adds branches to the store0set predictor (Store/Branch sets). The intuition behind this modification is that branches act as proxies for stores in their CD region. Hence, a branch can give future loads hints on which stores may be fetched in the future, if this happens to be a mispredicted branch.

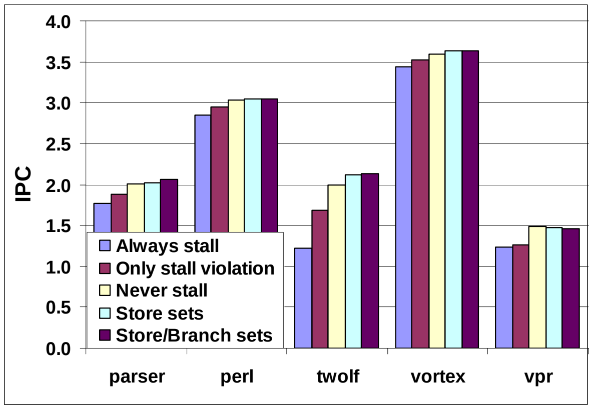
Figure 43 shows IPC results of the TCI architecture with different memory dependence predictors. Figure 43(a)-(c) give the results of individual benchmarks, whereas Figure 43(d) shows the harmonic mean IPC results. Notice that predictors based on the store-set predictor outperform the basic predictors in all benchmarks except bzip, li, and vpr. We also notice that the branch-set enhancement further improves on the performance of the base store-set predictor, on average. For individual benchmark results, we notice that all benchmarks except bzip and vpr see an improvement.



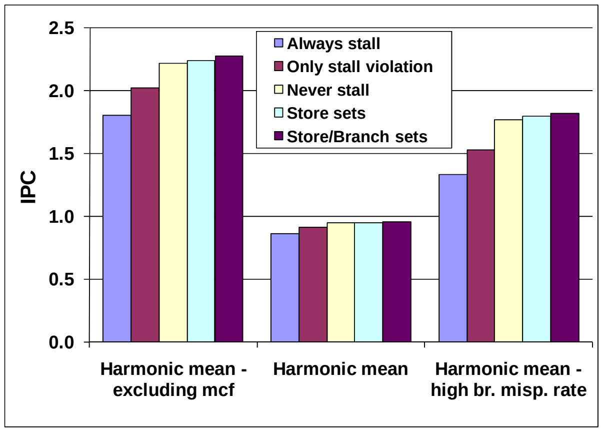
(a)



(b)



(c)



(d)

Figure 43. Memory dependence predictors.

6.11.5 Sensitivity to the number of checkpoints and poison vector bits

TCI leverages the checkpoint tags to identify branches covered by control independence. The more checkpoints allocated to the processor, the higher the branch misprediction coverage and, hence, higher potential performance.

Figure 44 shows the IPC results of the TCI architecture with 4, 8, 16, and 32 checkpoints. Figure 44(a)-(c) give the results of individual benchmarks, whereas Figure 43(d) shows the harmonic mean IPC results. The results show that increasing the number of checkpoints helps performance. In addition, the performance starts to saturate with 16 checkpoints. In vpr, we notice that TCI with 16 checkpoints outperforms TCI with 32 checkpoints. Analyzing the data gathered from the two runs, the reason for this degradation is a higher branch misprediction rate. By covering additional branch mispredictions in vpr, we worsen the accuracy of the branch predictor (possibly due to corrupting the GHR more with 32 checkpoints then with 16 checkpoints).

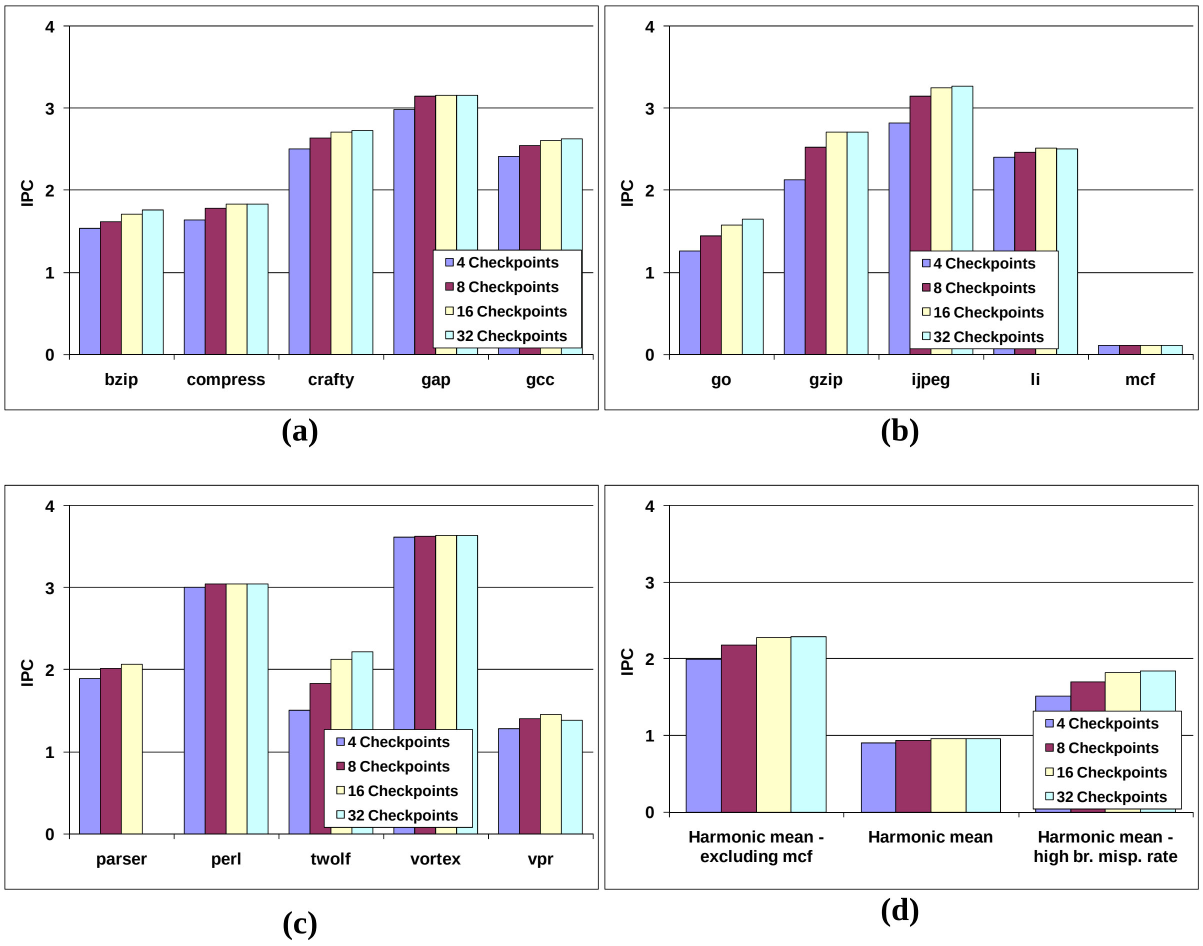


Figure 44. TCI IPC results with varying number of checkpoints.

6.11.6 Sensitivity to the number of CFSs and to IRS optimizations

TCI leverages the CFS to detect the reconvergence of branches. Initially, the leading sequencer pushes the reconvergent PCs of fetched branches onto the CFS. When the reconvergent PC of a branch is fetched, the CFS detects reconvergence and the process of copying the CIDD instructions into the RXB is commenced. During branch misprediction recovery, the newly fetched CD region may itself contain internal control-flow. To be able to detect inner reconvergent points, we leverage a second CFS that is devoted to the repair rename map. Without this second CFS, the internal branches would need to rely on the original reconvergent point, which may not be optimal since it is a distant reconvergent point for these branches.

In this section, we first investigate the effect of using the CFS only with the speculative rename map versus allocating an additional CFS for the repair rename map. Second, we look at the effect of using IRS optimizations discussed in Section 4.3.1.4 on performance.

Figure 45 shows the IPC results of the TCI architecture with one or two CFSs. For both models, we study the effect of using an unoptimized IRS or an optimized IRS. Figure 45(a)-(c) give the results of individual benchmarks, whereas Figure 45(d) shows the harmonic mean IPC results. From the results, we can conclude that, on average, both multiple CFSs and an optimized IRS improve the performance of TCI. However, some benchmarks show different trends. For example, twolf only benefits from using multiple CFSs and not from using the optimized IRS. On the other hand, gap only benefits from using an optimized IRS.

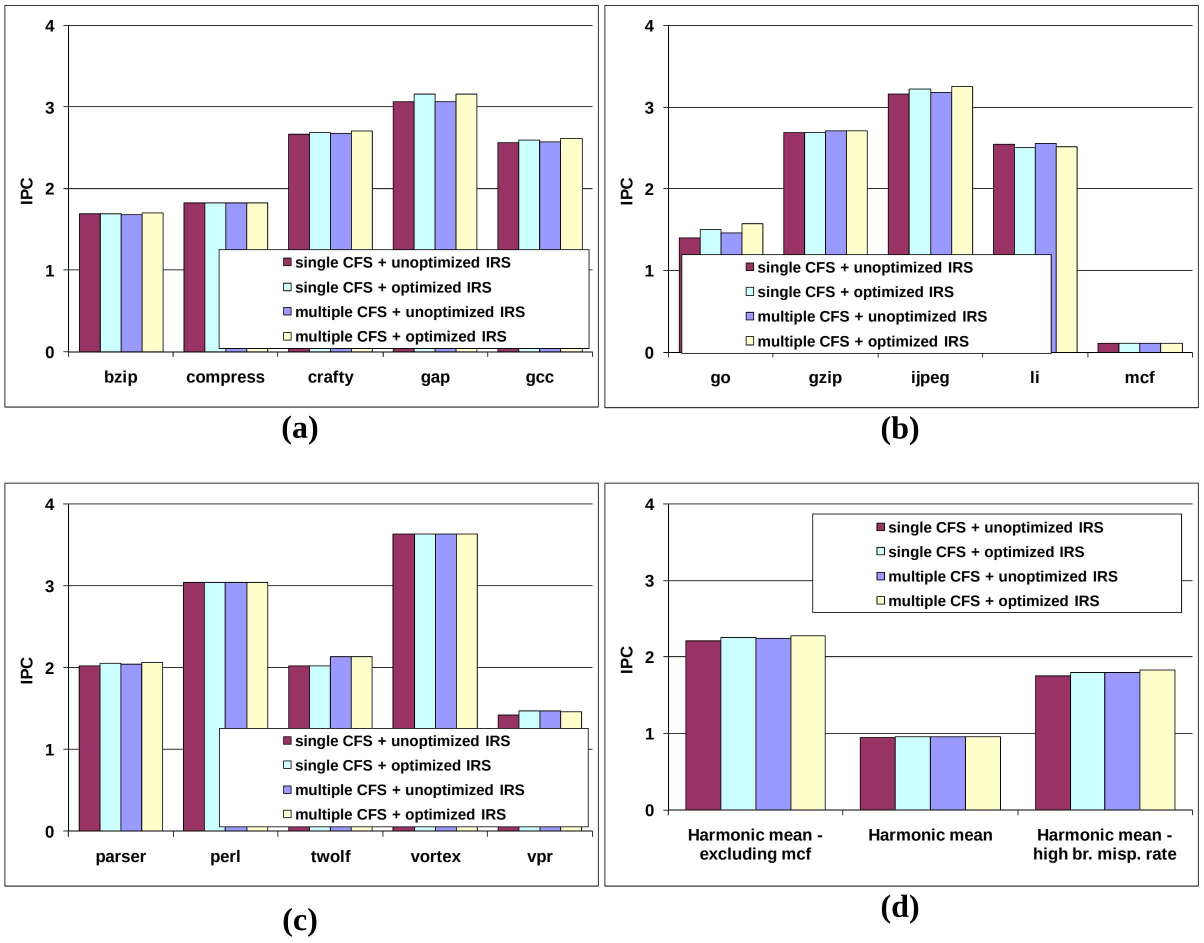


Figure 45. TCI IPC results with a single or multiple CFSs while using an optimized or unoptimized IRS.

6.12 Additional related work

We already compared and contrasted TCI with the following control independence architectures in Chapter 5 and, in the interest of space, that discussion is not repeated here: speculative multithreading architectures such as Multiscalar (Sohi, et al., 1995) and DMT (Akkary, et al., 1998), trace processors (Rotenberg, et al., 1999), and superscalar based implementations including instruction reuse (Sodani, et al., 1997), dual ROBs (Chou, et al., 1999), Skipper (Cher, et al., 2001), exact convergence (Gandhi, et al., 2004), and a generic implementation (Rotenberg, et al., 1999).

ReSlice (Sarangi, et al., 2005) uses slice re-execution to selectively recover from data misspeculation. Correct repair is guaranteed by checking for sufficient slice conditions. In general, ReSlice is designed for any data misspeculation handling including control-flow influenced data misspeculation, but it was studied only for thread-level speculation (TLS). ReSlice aborts slice re-execution if there are branches (whether in the slice or not) that change the slice's instructions. As we illustrated with the example in Section 6.3.1 of two co-mingled CIDD slices, RXB reconstruction allows slices to change, moreover, the co-mingled slices can resequence in any order, with correct results.

The continual flow pipeline (CFP) (Srinivasan, et al., 2004) is related to our work in that CFP takes an analogous approach for releasing resources of L2 miss dependent instructions. However, CFP does not exploit control independence.

Chapter 7

Summary and future work

In this thesis, we investigated two main claims. First, we compared different branch misprediction tolerance techniques qualitatively and quantitatively, including multipath, static/dynamic predication, skip-based control independence (CI-skip), and speculation-based control independence (CI-speculate). As a result of this comparison, we claim that CI-speculate is the best-performing branch misprediction tolerance technique on a processor with realistic resources. The chief reason is that CI-speculate does not penalize correctly predicted branches and, hence, complements the branch predictor.

Second, we analyzed the strengths and weaknesses of different CI-speculate models, including *Proxy*, *Seq CI*, and *Seq CIDD*, in the context of a high-performance checkpoint-based substrate compatible with control independence. We showed that *Seq CIDD* combines the resource efficiency of *Seq CI* and the bandwidth efficiency of *Proxy*, while eliminating their disadvantages.

Finally, we presented Transparent Control Independence (TCI), a new microarchitecture that implements the *Seq CIDD* model. By proactively identifying CIDD instructions in preparation for a branch misprediction, TCI only re-renames CIDD instructions and only reallocates resources to CIDD instructions when recovery is needed. Hence, TCI yields a highly streamlined pipeline that quickly recycles resources based on conventional speculation, enabling a large window with small cycle-critical resources, and prevents many mispredictions from disrupting this large window by fetching a condensed and self-sufficient recovery program.

This thesis has investigated the TCI microarchitecture in detail. However, many interesting directions are left for future work.

- Overcome CI-speculate and CI-skip limitations: CI-speculate is limited by the wasted bandwidth/resources consumed by the incorrect CD instructions. On the other hand, CI-skip does not waste any bandwidth/resources on incorrect CD instructions, at the expense of penalizing many correctly predicted branches. It is worthwhile investigating new architectures that overcome both CI-speculate's and CI-skip's limitations.
- Control independence aware compiler: Control independence can only tolerate mispredicted branches with reconvergent points. There remains a subset of mispredicted branches with no reconvergent points (no reconvergence information or very large CD region). In addition, the performance of control independence is dependent on the quality of the CI region (ratio of CIDI instructions to CIDD instructions). The compiler can potentially increase the performance of control independence by generating favorable code.
- Scalable performance: Branch mispredictions limit performance scalability of control independence based superscalar processors by wasting resources and bandwidth on incorrect CD instructions. Scaling the window size and issue width does not result in proportional performance scaling. Implementing TCI on top of Thread-level Speculation (TLS) may achieve scalability because TLS allocates some resources and bandwidth to future CI instructions while still allocating resources and bandwidth near the commit point (balancing CI-skip and CI-speculate). A drawback

of TLS is the inability to repair branches that have speculatively retired long ago, and that depend on misspeculated loads. TCI can solve this by adding control-flow slices to load recovery slices, e.g., overcoming weaknesses of ReSlice (Sarangi, et al., 2005). TCI would provide load-induced control-flow misspeculation tolerance within a large speculative thread and TLS would balance resources and bandwidth fairly between near and distant threads. We believe this would lead to scalable performance with resource scaling.

- Additional uses of the recovery program: One of the interesting contributions of the TCI microarchitecture is the ability to repair the processor’s state after misspeculation using a self-sufficient recovery program. This concept could be extended to repair other types of speculation. In particular, I would like to investigate the ability to speculate past page faults and then leverage a self-sufficient recovery program to recover from any misspeculation.

Chapter 8

Bibliography

Ahuja Pritpal S [et al.] Multipath execution: opportunities and limits [Conference] // ICS ’98: Proceedings of the 12th international conference on Supercomputing. - New York, NY, USA : ACM Press, 1998. - pp. 101-108.

Akkary Haitham and Driscoll Michael A A dynamic multithreading processor [Conference] // MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture. - Los Alamitos, CA, USA : IEEE Computer Society Press, 1998. - pp. 226-236.

Akkary Haitham, Rajwar Ravi and Srinivasan Srikanth T Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors [Conference] // MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. - Washington, DC, USA : IEEE Computer Society, 2003. - p. 423.

Allen J R [et al.] Conversion of control dependence to data dependence [Conference] // POPL ’83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. - New York, NY, USA : ACM Press, 1983. - pp. 177-189.

Al-Zawawi Ahmed S [et al.] Transparent control independence (TCI) [Journal] // SIGARCH Comput. Archit. News. - New York, NY, USA : ACM Press, 2007. - 2 : Vol. 35. - pp. 448-459.

Ando Hideki [et al.] Unconstrained speculative execution with predicated state buffering [Conference] // ISCA ’95: Proceedings of the 22nd annual international symposium on Computer architecture. - New York, NY, USA : ACM Press, 1995. - pp. 126-137.

Bondi James O, Nanda Ashwini K and Dutta Simonjit Integrating a misprediction recovery cache (MRC) into a superscalar pipeline [Conference] // MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture. - Washington, DC, USA : IEEE Computer Society, 1996. - pp. 14-23.

Burger Doug, Austin Todd M and Bennett Steve Evaluating Future Microprocessors: The SimpleScalar Tool Set [Book]. - 1996.

Cher Chen-Yong and Vijaykumar T N Skipper: a microarchitecture for exploiting control-flow independence [Conference] // MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture. - Washington, DC, USA : IEEE Computer Society, 2001. - pp. 4-15.

Chou Yuan, Fung Jason and Shen John Paul Reducing branch misprediction penalties via dynamic control independence detection [Conference] // ICS ’99: Proceedings of the 13th international conference on Supercomputing. - New York, NY, USA : ACM Press, 1999. - pp. 109-118.

Chrysos George Z and Emer Joel S Memory dependence prediction using store sets [Conference] // ISCA ’98: Proceedings of the 25th annual international symposium on Computer architecture. - Washington, DC, USA : IEEE Computer Society, 1998. - pp. 142-153.

Collins Jamison D, Tullsen Dean M and Wang Hong Control Flow Optimization Via Dynamic Reconvergence Prediction [Conference] // MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture. - Washington, DC, USA : IEEE Computer Society, 2004. - pp. 129-140.

Cristal A [et al.] Large Virtual ROBs by Processor Checkpointing [Journal]. - 2002.

Cristal Adri&#225 [et al.] Toward kilo-instruction processors [Journal] // ACM Trans.Archit.Code Optim.. - 2004. - 4 : Vol. 1. - pp. 389-417.

Cristal Adrian [et al.] Out-of-Order Commit Processors [Journal] // hpca. - 2004. - Vol. 00. - p. 48.

Dubey Pradeep K [et al.] Single-program speculative multithreading (SPSM) architecture: compiler-assisted fine-grained multithreading [Conference] // PACT ’95: Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques. - Manchester, UK, UK : IFIP Working Group on Algol, 1995. - pp. 109-121.

Franklin Manoj and Sohi Gurindar S. ARB: A Hardware Mechanism for Dynamic Reordering of Memory [Journal] // IEEE Trans.Computers. - 1996. - 5 : Vol. 45. - pp. 552-571.

Franklin Manoj The multiscalar architecture [Book]. - 1993.

Gandhi Amit [et al.] Scalable Load and Store Processing in Latency Tolerant Processors [Journal] // isca. - 2005. - Vol. 00. - pp. 446-457.

Gandhi Amit, Akkary Haitham and Srinivasan Srikanth T Reducing Branch Misprediction Penalty via Selective Branch Recovery [Journal] // hpca. - 2004. - Vol. 00. - p. 254.

Gross Thomas R and Hennessy John L Optimizing delayed branches [Journal]. - 1982. - pp. 114-120.

Heil T and Smith J Selective dual path execution [Book]. - 1996.

Hennessy John [et al.] Hardware/software tradeoffs for increased performance [Journal]. - 1982. - pp. 2-11.

Hennessy John [et al.] MIPS: A microprocessor architecture [Conference] // MICRO 15: Proceedings of the 15th annual workshop on Microprogramming. - Piscataway, NJ, USA : IEEE Press, 1982. - pp. 17-22.

Hilton Andrew D and Roth Amir Ginger: control independence using tag rewriting [Journal] // SIGARCH Comput. Archit. News. - New York, NY, USA : ACM Press, 2007. - 2 : Vol. 35. - pp. 436-447.

Hwu W W and Patt Y N Checkpoint repair for out-of-order execution machines [Conference] // ISCA ’87: Proceedings of the 14th annual international symposium on Computer architecture. - New York, NY, USA : ACM Press, 1987. - pp. 18-26.

Jacobsen Erik, Rotenberg Eric and Smith J E Assigning confidence to conditional branch predictions [Conference] // MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture. - Washington, DC, USA : IEEE Computer Society, 1996. - pp. 142-152.

Jimenez Daniel A and Lin Calvin Dynamic Branch Prediction with Perceptrons [Conference] // HPCA. - 2001. - pp. 197-206.

Karkhanis T and Smith J A day in the life of a data cache miss [Book]. - 2002.

Kim Hyesoon [et al.] Diverge-Merge Processor (DMP): Dynamic Predicated Execution of Complex Control-Flow Graphs Based on Frequently Executed Paths [Conference] // MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. - Washington, DC, USA : IEEE Computer Society, 2006. - pp. 53-64.

Kim Hyesoon [et al.] Profile-assisted Compiler Support for Dynamic Predication in Diverge-Merge Processors [Conference] // CGO ’07: Proceedings of the International Symposium on Code Generation and Optimization. - Washington, DC, USA : IEEE Computer Society, 2007. - pp. 367-378.

Kim Hyesoon [et al.] Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution [Conference] // MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture. - Washington, DC, USA : IEEE Computer Society, 2005. - pp. 43-54.

Klauser A [et al.] Dynamic Hammock Predication for Non-Predicated Instruction Set Architectures [Conference] // PACT ’98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques. - Washington, DC, USA : IEEE Computer Society, 1998. - p. 278.

Klauser Artur, Paithankar Abhijit and Grunwald Dirk Selective eager execution on the PolyPath architecture [Conference] // ISCA ’98: Proceedings of the 25th annual international symposium on Computer architecture. - Washington, DC, USA : IEEE Computer Society, 1998. - pp. 250-259.

Lam Monica S and Wilson Robert P Limits of control flow on parallelism [Conference] // ISCA ’92: Proceedings of the 19th annual international symposium on Computer architecture. - New York, NY, USA : ACM Press, 1992. - pp. 46-57.

Lebeck Alvin R [et al.] A large, fast instruction window for tolerating cache misses [Conference] // ISCA ’02: Proceedings of the 29th annual international symposium on Computer architecture. - Washington, DC, USA : IEEE Computer Society, 2002. - pp. 59-70.

Lee Johnny K and Smith Alan Jay Branch prediction strategies and branch target buffer design [Journal]. - 1995. - pp. 83-99.

Lipasti Mikko Herman Value locality and speculative execution [Book]. - 1998.

Mahlke Scott A [et al.] A comparison of full and partial predicated execution support for ILP processors [Conference] // ISCA ’95: Proceedings of the 22nd annual international symposium on Computer architecture. - New York, NY, USA : ACM Press, 1995. - pp. 138-150.

Marcuello Pedro [et al.] Speculative multithreaded processors [Conference] // ICS ’98: Proceedings of the 12th international conference on Supercomputing. - New York, NY, USA : ACM Press, 1998. - pp. 77-84.

Martinez Jose F [et al.] Cherry: checkpointed early resource recycling in out-of-order microprocessors [Conference] // MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture. - Los Alamitos, CA, USA : IEEE Computer Society Press, 2002. - pp. 3-14.

McFarling S and Hennesey J Reducing the cost of branches [Conference] // ISCA ’86: Proceedings of the 13th annual international symposium on Computer architecture. - Los Alamitos, CA, USA : IEEE Computer Society Press, 1986. - pp. 396-403.

Mirapuri Sunil, Woodacre Michael and Vasseghi Nader The Mips R4000 Processor [Journal] // IEEE Micro. - 1992. - 2 : Vol. 12. - pp. 10-22.

Morano D [et al.] Realizing high IPC through a scalable memory-latency tolerant multipath microarchitecture [Journal] // SIGARCH Comput.Archit.News. - 2003. - 1 : Vol. 31. - pp. 16-25.

Moudgill Mayan, Pingali Keshav and Vassiliadis Stamatis Register renaming and dynamic speculation: an alternative approach [Conference] // MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture. - Los Alamitos, CA, USA : IEEE Computer Society Press, 1993. - pp. 202-213.

Mutlu Onur [et al.] On Reusing the Results of Pre-Executed Instructions in a Runahead Execution Processor [Journal] // IEEE Comput.Archit.Lett.. - 2005. - 1 : Vol. 4. - p. 2.

Olukotun Kunle [et al.] The case for a single-chip multiprocessor [Conference] // ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems. - New York, NY, USA : ACM Press, 1996. - pp. 2-11.

Palacharla Subbarao, Jouppi Norman P and Smith J E Complexity-effective superscalar processors [Journal]. - 1997. - pp. 206-218.

Park I, Ooi Chong Liang and Vijaykumar T N Reducing Design Complexity of the Load/Store Queue [Conference] // MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. - Washington, DC, USA : IEEE Computer Society, 2003. - p. 411.

Path-based Hardware Loop Prediction [Conference] // 4th International Conference on Control, Virtual Instrumention and Digital Systems. - Mexico City, Mexico : [s.n.], 2002. - pp. 29-38.

Patterson David A and Sequin Carlo H RISC I: A Reduced Instruction Set VLSI Computer [Conference] // ISCA ’81: Proceedings of the 8th annual symposium on Computer Architecture. - Los Alamitos, CA, USA : IEEE Computer Society Press, 1981. - pp. 443-457.

Raasch Steven E, Binkert Nathan L and Reinhardt Steven K A scalable instruction queue design using dependence chains [Conference] // ISCA ’02: Proceedings of the 29th annual international symposium on Computer architecture. - Washington, DC, USA : IEEE Computer Society, 2002. - pp. 318-329.

Riseman E M and Foster C C The Inhibition of Potential Parallelism by Conditional Jumps [Journal] // IEEE Trans.Computers. - 1972. - 12 : Vols. C-21. - pp. 1405-1411.

Rotenberg E, Jacobson Q and Smith J A Study of Control Independence in Superscalar Processors [Conference] // HPCA ’99: Proceedings of the 5th International Symposium on High Performance Computer Architecture. - Washington, DC, USA : IEEE Computer Society, 1999. - p. 115.

Rotenberg Eric and Smith Jim Control independence in trace processors [Conference] // MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture. - Washington, DC, USA : IEEE Computer Society, 1999. - pp. 4-15.

Sarangi Smruti R, Liu Josep Torrellas and Zhou Yuanyuan ReSlice: Selective Re-Execution of Long-Retired Misspeculated Instructions Using Forward Slicing [Conference] // MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture. - Washington, DC, USA : IEEE Computer Society, 2005. - pp. 257-270.

Sethumadhavan Simha [et al.] Scalable Hardware Memory Disambiguation for High ILP Processors [Conference] // MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. - Washington, DC, USA : IEEE Computer Society, 2003. - p. 399.

Sherwood Timothy [et al.] Automatically characterizing large scale program behavior [Journal] // SIGOPS Oper.Syst.Rev.. - 2002. - 5 : Vol. 36. - pp. 45-57.

Smith Aaron [et al.] Dataflow Predication [Conference] // MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. - Washington, DC, USA : IEEE Computer Society, 2006. - pp. 89-102.

Smith James E A study of branch prediction strategies [Conference] // ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture. - Los Alamitos, CA, USA : IEEE Computer Society Press, 1981. - pp. 135-148.

Sodani Avinash and Sohi Gurindar S Dynamic instruction reuse [Conference] // ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture. - New York, NY, USA : ACM Press, 1997. - pp. 194-205.

Sohi Gurindar S, Breach Scott E and Vijaykumar T N Multiscalar processors [Conference] // ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture. - New York, NY, USA : ACM Press, 1995. - pp. 414-425.

Srinivasan Srikanth T [et al.] Continual flow pipelines [Conference] // ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems. - New York, NY, USA : ACM Press, 2004. - pp. 107-119.

Steffan J and Mowry T The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization [Conference] // HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture. - Washington, DC, USA : IEEE Computer Society, 1998. - p. 2.

Tendler Joel M, Dodson J Steve and and J S POWER4 system microarchitecture. [Journal] // IBM Journal of Research and Development. - 2002. - 1 : Vol. 46. - pp. 5-26.

Tsai Jenn-Yuan and Yew Pen-Chung The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation [Conference] // PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques. - Washington, DC, USA : IEEE Computer Society, 1996. - p. 35.

Tyson Gary Scott The effects of predicated execution on branch prediction [Journal]. - 1994. - pp. 196-206.

Tyson Gary, Lick Kelsey and Farrens Matthew Limited Dual Path Execution [Journal]. - 1997.

Uht Augustus K [et al.] Levo - A Scalable Processor With High IPC [Journal] // J. Instruction-Level Parallelism. - 2003. - Vol. 5.

Uht Augustus K, Sindagi Vijay and Hall Kelley Disjoint eager execution: an optimal form of speculative execution [Conference] // MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture. - Los Alamitos, CA, USA : IEEE Computer Society Press, 1995. - pp. 313-325.

Vijaykumar T N and Sohi Gurindar S Task selection for a multiscalar processor [Conference] // MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture. - Los Alamitos, CA, USA : IEEE Computer Society Press, 1998. - pp. 81-92.

Vintan Lucian N [et al.] An alternative to branch prediction: pre-computed branches [Journal] // SIGARCH Comput.Archit.News. - 2003. - 3 : Vol. 31. - pp. 20-29.

Wallace Steven, Calder Brad and Tullsen Dean M Threaded multiple path execution [Conference] // ISCA ’98: Proceedings of the 25th annual international symposium on Computer architecture. - Washington, DC, USA : IEEE Computer Society, 1998. - pp. 238-249.

Wallace Steven, Tullsen Dean M and Calder Brad Instruction Recycling on a Multiple-Path Processor [Conference] // HPCA ’99: Proceedings of the 5th International Symposium on High Performance Computer Architecture. - Washington, DC, USA : IEEE Computer Society, 1999. - p. 44.