Abstract

AL-OTOOM, MUAWYA MOHAMED. Preliminary Study of Trace-Cache-Based Control Independence Architecture. (Under the direction of Dr. Eric Rotenberg.)

Conventional superscalar processors recover from a mispredicted branch by squashing all instructions after the branch. While simple, this approach needlessly re-executes many future control-independent (CI) instructions after the branch's reconvergent point. Selective recovery is possible, but is complicated by the fact that some control-independent instructions must be singled out for re-execution, namely those that depend on data influenced by the mispredicted branch. That is, control-independent data-dependent (CIDD) instructions must be singled out for re-execution, thus avoiding needless re-execution of control-independent data-independent (CIDI) instructions.

To contrast different recovery models, we abstract the recovery process as constructing a "recovery sub-program" for repairing partially incorrect future state. In this conceptual framework, selective recovery constructs a shorter recovery sub-program than full recovery. In current selective recovery microarchitectures, the recovery sub-program is constructed on-the-fly after detecting a mispredicted branch, by sequencing through all CI instructions and singling out only the CIDD instructions among them. Not only is this discriminating approach complex, but the same recovery sub-program is repeatedly constructed every time this branch is mispredicted.

We propose constructing the recovery sub-program for each branch once and caching it for future use. In particular, traces of CIDD instructions are pre-constructed and stored in a

recovery trace cache. When a misprediction is detected, first, the branch's correct control-dependent instructions are fetched from the conventional instruction cache as usual. Then, at the reconvergent point, fetching simply switches from the instruction cache to the recovery trace cache. The appropriate recovery trace is fetched from the recovery trace cache at this time. In this way, fetching only the CIDD instructions is as simple as fetching all CI instructions from a conventional instruction cache. No explicit singling-out process is needed as this was done *a priori*, on the fill-side of the trace cache. Therefore, the recovery trace cache is efficient on multiple levels, combining the simplicity of full recovery with the performance of selective recovery.

This thesis explains the proposed trace-cache-based control independence architecture, at a high level. Preliminary studies are also presented, to project the potential of exploiting control independence as well as the effectiveness of a trace-cache-based approach in particular. The results include (i) breakdowns of retired dynamic instructions into different categories, based on their control and data dependences with respect to prior mispredicted branches, (ii) contributions of individual recovery traces to total CIDI instruction savings, and (iii) hit ratios of finite recovery trace caches.

PRELIMINARY STUDY OF TRACE-CACHE-BASED CONTROL INDEPENDENCE ARCHITECTURE

by

MUAWYA MOHAMED AL-OTOOM

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

COMPUTER ENGINEERING

Raleigh, NC

2006

Approved by:

____
Dr. Eric Rotenberg
Chair of Advisory Committee

______________________ ______________________
Dr. W. Rhett Davis Dr. Suleyman Sair

Dedication

To my loving parents …

To my love Zeinab …

To my beloved Jordan …

Biography

Muawya Al-Otoom was born in Amman, Jordan, on November 1981, the second son of Mohamed and Raisa Al-Otoom. Muawya spent the first seven years of his childhood in the capital Amman where he attended the first year of his high school at Dar El-Elem School. In 1988, Muawya moved with his family to Souf, his hometown, to join the rest of Al-Otoom family. In 1999, Muawya finished his high school education and joined Jordan University of Science and Technology (JUST) to pursue his undergraduate education in computer engineering. He received his Bachelor's degree in computer engineering from JUST in 2003 and worked there for one year as a teaching assistant. Motivated by an undergraduate project on Tomasulo's algorithm, Muawya decided to pursue his graduate studies in computer microarchitecture at North Carolina State University (NCSU) in 2004. He joined the research group of Dr. Eric Rotenberg in 2005 as a Master Student. Now he is continuing his graduate studies at NCSU toward the PhD degree under the direction of Dr. Eric Rotenberg. In 2006, Muawya was invited to be a member of the Phi Kappa Phi honor society in recognition of his academic achievements.

Acknowledgments

First, I would like to thank God for everything, for giving me health, family, friends, education and good life.

I want to thank my parents, Mohamed and Raisa Al-Otoom. I don’t know what I would have done without their continuous support. Despite, the long distance, I have never felt that I am far from them. Thanks to my father for being a great model I wanted to be like always. Thanks to my brothers, Muaz and Awni, and my sisters, Doa’a, Demah, and Amomah. Thanks to all of my uncles and aunts for their calls, and for making sure that I am doing fine.

Thanks to my love Zeinab, her continuous calls and support were great help while we were apart. Her trust in me helped me face all fears here and made me proud of her. I am waiting for the day when we will be together.

Special thanks to my advisor, mentor, and teacher Dr. Eric Rotenberg. Thank you for believing in me when nobody did. Thanks for being a good teacher through this year, and showing me how to be a good researcher. Thanks for always reminding me how to write good code by using the “Japanese Car Bumper” model. Thanks for tolerating my nagging during the submission period.

I would like to thank my friends and roommates here in Raleigh: Tareq Ghaith, my previous and current roommate, has always been a good brother; Mazen Kharbutli, my first friend in Raleigh and unofficial mentor during my first experience in research, Ali El-Haj-Mahmoud, my housemate and cubemate; Mahmoud Chehab (MC), my brother; Monther Al-Dwari, thanks for the 24-hour services upon coming to Raleigh; Khalid Gharaibeh and Hazim Al-Dwari, my Cup-O-Joes dudes.

I would like to thank all of my current and previous research group members: Ali El-Haj-Mahmoud, Aravindh Anantaraman, Ahmed Al-Zawawi, Vimal Reddy, Hashem Hashemi, Ravi Venkatesan, and Sailashri Parthasarathy. Special thanks to Ali and Aravindh for the 24-hour technical and non-technical support. Thanks to Ahmed for being our enthusiastic member responsible for the brainstorming us at our meetings and being the second criminal in the crime of control independence after Eric.

I would like to thank my committee members, Dr. W. Rhett Davis and Dr. Suleyman Sair, for their help in finishing up this work.

Thanks to Sandy Bronson, CESR administrative assistant, for making my life easier through taking care of all administrative and logistic issues.

I would like to thank all of my professors at JUST, especially Dr. Omer Al-Jarah, Dr. Abdullah Bataineh, and Dr. Sameer Bataineh.

Thanks to my friends during my undergrad studies and who are now here in the US, thanks for keeping in touch: Ghaith Matalkah, Rawad Haddad, Abdullah Khresha, Bashar Gharaibeh, Samer Al-Kiswani, Mohamed Kharashkah, Mwafag Al-Otoom, Mohamed Zbeidee, and Hakim Hussein.

Thanks to all my fellows in Souf: Sadeq, Omar, Hassan, Hamzeh, Zaid, and Tawfeeg. Thanks for keeping in touch during my stay here. Thanks for the good old days we spent in

Souf, especially “experiencing our survival skills at 3:00 AM driving with no windshield while it’s raining cats and dogs”. Thanks to my cousin and friend, Ahmed, who I can always depend on.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Researchers have proposed exploiting control independence to reduce the branch misprediction penalty in high-performance processors. Control independence architectures avoid needlessly re-executing future misprediction-independent instructions. This thesis presents a preliminary study for a *Trace-Cache-Based Control Independence Architecture*. My preliminary study discusses a possible microarchitecture at a high level and projects the potential of the microarchitecture in terms of the amount of misprediction-independent instruction savings and other results.

1.1 Motivation: Trend of Large Instruction Windows

Recent studies have suggested increasing the instruction window size [3][10][11][20][24][38] as one solution to hide the latency of cache-missed loads. By allowing future independent instructions after cache-missed loads to be fetched and executed, much of the long latency of cache misses can be hidden effectively. However, with the presence of branch mispredictions, forming large instruction windows will be extremely difficult. All of the instructions after the misprediction have to be squashed, and this will leave the instruction window with fewer instructions to execute in parallel with pending cache-misses.

Many solutions have been proposed [5][15][16][21][31][35][40] (discussed in the related work section) to deal with the problem of branch mispredictions. Some of these solutions try improve the accuracy of the branch predictor. Other solutions try to reduce the penalty of a branch misprediction. Control independence [2][7][8][12][19][31][32][35][36][41] is one solution that aims at reducing the penalty of a branch misprediction by not squashing all instructions after the branch, saving some of the good work that is independent of the branch itself.

1.2 Revisiting Control Independence

Dynamic instructions, in general, depend on prior branches through two types of dependences, control dependences and data dependences. As shown in Figure 1-1 instructions in both basic blocks A and B are considered to be control-dependent (*CD*) on the branch since fetching them is dependent on the direction taken by the branch. Basic block C is considered to be control-independent of the branch since it will be fetched regardless of the decision taken by the branch. The first instruction in basic block C is called the reconvergent point of the branch since it reconverges the control-flow of both paths of the branch. Although all instructions in basic block C are control-independent (*CI*) of the branch, some of them depend on values produced on either control-dependent path of the branch. For example, the consumption of R5 in basic block C, depends on the direction taken by the branch since it may use the production of R5 in basic block A or the production of R5 before the branch, as shown in Figure 1-1 these instructions are considered to be control-independent data-dependent (*CIDD*) on the branch. On the other hand, the rest of the instructions after the reconvergent point are both control and data independent of the branch

(*CIDI*). The benefit of exploiting control independence is not discarding CIDI instructions in the shadow of branch mispredictions.

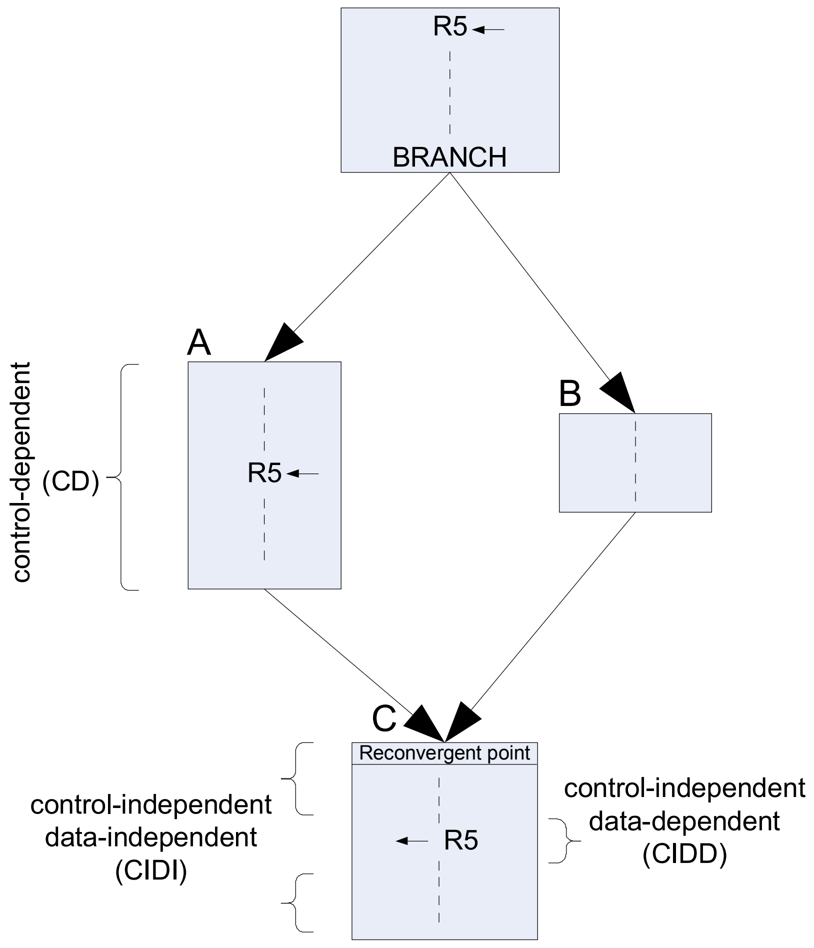


Figure 1-1: Control independence terminology.

A conventional superscalar processor [33] recovers from a branch misprediction by squashing all instructions after the misprediction and rolling back the architectural state. The state is rolled back either by walking backward from the last fetched instruction until the point of the branch misprediction, fixing the state and freeing resources incrementally, or by restoring the state instantaneously from a checkpoint at the mispredicted branch or at an earlier instruction. Although squashing all instructions after a branch misprediction will force the processor to re-execute many correct CIDI instructions unnecessarily, full squash recovery is simple to implement. That is, it is simplest to squash CD, CIDD, and CIDI instructions indiscriminately.

Processors that exploit control independence do not squash all instructions after the mispredicted branch and roll back the architectural state the same way a conventional superscalar processor does. Only the incorrect CD instructions are squashed or drained from pipeline, freeing their allocated resources accordingly. Then the correct CD instructions are fetched, thereby replacing the incorrect CD instructions that were removed. After repairing the branch's CD region, all CIDD instructions must be identified and re-issued again with new correct values. Note that the architectural state is not rolled back when the branch misprediction is detected. Therefore, when the misprediction is detected, the future state is partially incorrect. Holes in the state caused by the incorrect CD and CIDD instructions are fixed after fetching and executing the correct CD instructions and re-executing the CIDD instructions.

In general, any control independence architecture should be able to support two main requirements:

(i) Removing incorrect CD instructions from the middle of the instruction window and inserting correct CD instructions into the middle of the instruction window.

(ii) Selectively re-issuing CIDD instructions with new correct values.

1.2.1 Managing Instructions Insertion and Removal

Contemporary superscalar processors use a Reorder Buffer (ROB) [34] to maintain precise register state for recovery from exceptions and branch mispredictions. The ROB is a circular FIFO buffer containing all in-flight instructions in the window in program order. The FIFO ensures that instructions commit their results to the precise architectural state in program order, despite executing of instructions out-of-order. Unfortunately, the ROB is not compatible with control independence, because control independence requires inserting and removing instructions from the middle of the window, violating simple FIFO management of the ROB.

Checkpoint-based architectures [3][10][11][14][24][38] have been proposed as an alternative to ROB-based architectures. A checkpoint-based architecture does not use a ROB to maintain precise state for recovery. Instead, it maintains precise state by checkpointing the register state at coarse intervals. The fact that there is no FIFO of instructions in between checkpoints means there is no structural obstruction to prevent inserting/removing instructions between checkpoints. In other words, a checkpoint-based architecture provides a flexible instruction window, i.e., an expandable and collapsible window.

In addition to not presenting any structural obstruction for inserting/removing instructions from the middle of the window, other aspects associated with instruction insertion/removal are transparently handled by the checkpoint-based substrate:

- *Incorrect CD instructions must be removed from the window, and their allocated resources must be freed.* Checkpoint-based architectures manage physical register allocation and deallocation through usage counters [3]. There is no custom requirement to squash incorrect CD instructions and free their resources since it is performed naturally by the substrate. Once an instruction executes, whether correct or not, it will drain out of the pipeline and decrement usage counters of its source registers. Once the usage counter of a physical register reaches zero and it is not mapped to any logical register (in the rename map), it is freed autonomously. Summing up, incorrect CD instructions need not be explicitly identified and squashed, as all instructions eventually drain and physical registers are freed autonomously.

- *Correct CD instructions must link to their producers correctly.* The correct CD instructions are fetched out-of-order. The checkpoint taken at the mispredicted branch can be used to rename these instructions, ensuring that they link to the right producers in the middle of the window.

1.2.2 Handling Selective Re-Issue

Proposed control independence architectures handle the selective re-issue problem either by saving all CI instructions in a separate buffer or fetching them one more time from the instruction cache. These approaches are inefficient since the CI instructions include both CIDI and CIDD instructions. Using either approach requires re-sequencing through all CI instructions and singling out only the CIDD instructions for selective re-issuing. Having to re-sequence through all CI instructions just to identify CIDD instructions reduces the benefit of exploiting control independence.

Another problem with these previous approaches is that they require repeatedly constructing the same set of CIDD instructions, every time the same branch misprediction occurs.

We propose a novel approach, in which “compressed” recovery traces of CIDD instructions are pre-constructed before resolving mispredictions and stored in a recovery trace cache. So, when a branch misprediction occurs, a compressed trace of pre-identified CIDD instructions will be fetched for recovery. This approach performs better because it avoids fetching all the CI instructions just to single out CIDD instructions. Moreover, it is simple and more efficient because it only constructs traces of CIDD instructions once, and then repeatedly reuses the traces for many recurrences of the same misprediction.

As discussed previously, although full squashing after a mispredicted branch discards many correctly executed instructions, it is very simple and appealing for designers since it requires only rolling back the architectural state to the point of the misprediction and re-directing the fetch engine to start fetching instructions from the correct path. In this thesis, I will suggest a trace-cache-based control independence architecture that provides the illusion of full squashing (hence preserving the simplicity of conventional recovery) without actually discarding correctly executed instructions. This will be achieved by fetching correct CD instructions from the instruction cache, as usual, and then simply switching the fetch unit to fetch compressed CIDD traces from the recovery trace cache.

1.3 The “Big Picture”

This thesis proposes a misprediction recovery approach that combines the simplicity of full recovery and the performance of selective recovery. This is illustrated in Figure 1-2 at a high

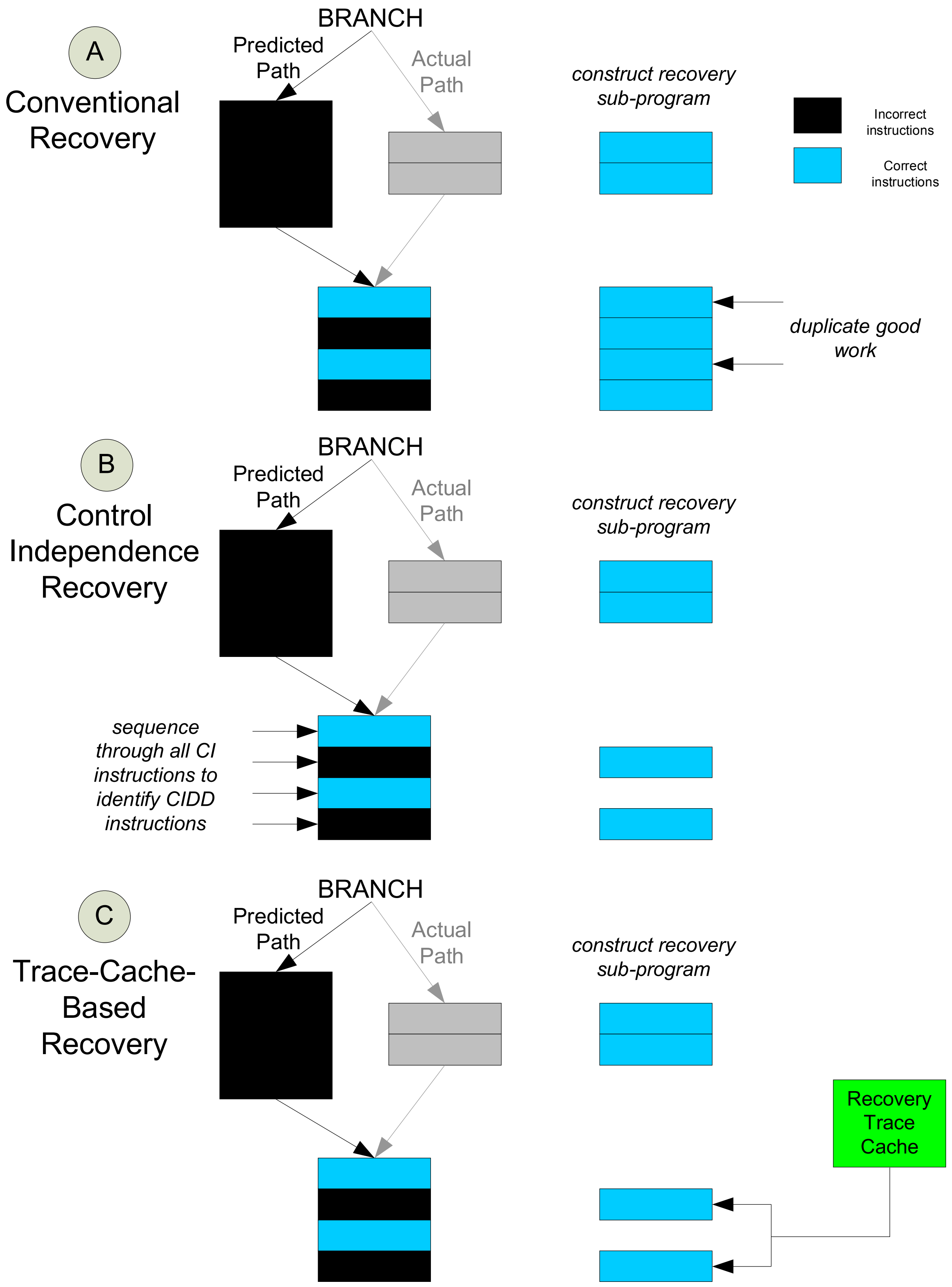

A
Conventional Recovery
BRANCH
Predicted Path
Actual Path
construct recovery sub-program
Incorrect instructions
Correct instructions
duplicate good work
B
Control Independence Recovery
BRANCH
Predicted Path
Actual Path
construct recovery sub-program
sequence through all CI instructions to identify CIDD instructions
C
Trace-Cache-Based Recovery
BRANCH
Predicted Path
Actual Path
construct recovery sub-program
Recovery Trace Cache

Figure 1-2: The "big picture".

level. The process of recovering from a branch misprediction is abstracted as constructing a "recovery sub-program". The recovery sub-program is a set of instructions that, when executed, repairs the future architectural state of the processor. This abstraction provides a conceptual framework for contrasting different recovery models.

Figure 1-2 shows how different paradigms construct a recovery sub-program for a single mispredicted branch, to repair the future architectural state corresponding to the end of the control flow graph (CFG) shown in the figure. Instructions colored in black are incorrect instructions that need to be replaced or re-executed, specifically incorrect CD instructions and CIDD instructions, respectively. Instructions colored in blue are correct instructions, comprised of (i) CIDI instructions and (ii) instructions in the recovery sub-program.

Part A of the figure shows how a conventional superscalar processor recovers from the mispredicted branch. Conceptually, it constructs a recovery sub-program that includes all correct CD instructions and all CI instructions (CIDD and CIDI) after the mispredicted branch. The recovery sub-program is easy to construct since it does not require discriminating CIDD and CIDI instructions. However, the recovery sub-program duplicates the work done by correct CIDI instructions as shown.

Part B of the figure shows how a conventional control independence architecture recovers from the same misprediction, in terms of constructing a recovery sub-program. A shorter recovery sub-program is constructed that includes the correct CD instructions and only CIDD instructions among the CI instructions. While the recovery sub-program is more efficient (shorter) by virtue of not duplicating correct CIDI instructions, constructing it is more complex and somewhat inefficient. Constructing the recovery sub-program requires sequencing through all CI instructions to identify CIDD instructions as shown in the figure. Moreover, the same recovery sub-program must be reconstructed every time the branch is mispredicted.

Part C illustrates the paradigm proposed in this thesis, a trace-cache-based approach for exploiting control independence. It constructs the same compressed recovery sub-program as a conventional control independence architecture. But, rather than repeatedly constructing the same recovery sub-program every time the branch is mispredicted, CIDD instructions are efficiently supplied by the recovery trace cache. The new approach is similar to approach A, in that it fetches the recovery sub-program from the instruction cache (CD region) and the recovery trace cache (compressed CI region consisting of only CIDD instructions), similar to fetching the whole program from the instruction cache. Yet the new approach is similar to approach B, in that it does not duplicate good work performed by CIDI instructions. Summing up, the new approach combines the simplicity of full recovery and the performance of selective recovery.

1.4 Contributions

The main contributions of this thesis are:

- *Preliminary analysis of CIDD traces in a recovery trace cache.* Although a detailed cycle-level timing simulator was not implemented for measuring performance improvement of the target microarchitecture directly, we measure the potential number of preserved CIDI instructions in the shadow of mispredictions using the recovery trace cache. CIDI instruction savings are measured using both unbounded trace caches and multiple finite-sized trace cache configurations. Moreover, we characterize the contributions of individual recovery traces to the total amount of CIDI savings. The hit ratios of different trace cache configurations are also provided. The combined studies give a preliminary indication of performance improvement.

- *Trace-Cache-Based Control Independence Architecture*. This thesis proposes the idea of a trace-cache-based control independence architecture and discusses it at a high level. However, a full microarchitecture design is left for future work. What distinguishes this architecture from other proposed control independence architectures is that it provides the illusion of simple full-squash recovery while still reducing the amount of unnecessary re-execution of misprediction-independent instructions.

Chapter 2

Target Future Microarchitecture

This chapter explains a possible microarchitecture for Trace-Cache-Based Control Independence, at a high level. Figure 2-1 shows a high-level view of a superscalar pipeline [33] modified for trace-cache-based control independence. In the figure, pathways and components are labeled with corresponding sections of this chapter that describe them. Section 2.1 describes the reconvergence predictor, Section 2.2 describes the recovery trace, pre-construction, and Section 2.3 and 2.4 describes preparing for recovery and recovery respectively.

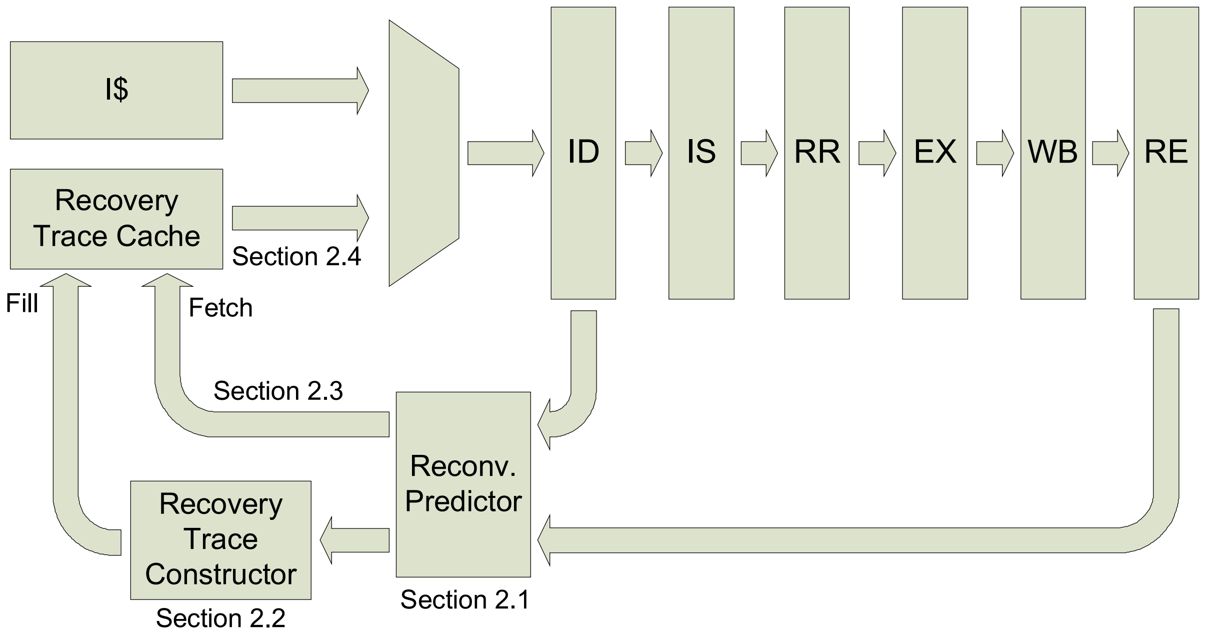


Figure 2-1: High-level view of possible trace-cache-based control independence microarchitecture.

2.1 Reconvergent Point and Influenced Register Set (IRS) Predictor

To exploit control independence with respect to a branch misprediction, two pieces of information are needed about the CD region of the mispredicted branch. (i) The program counter (PC) of the reconvergent point, which identifies the first instruction in the CI region. (ii) The Influenced Register Set (IRS), which is the set of logical registers that are written to along one or more CD paths of the branch. These two pieces of information are used to identify CIDD instructions when pre-constructing traces. All CI instructions, starting from the reconvergent point, that directly or indirectly depend on logical registers specified in the IRS will be singled out as CIDD instructions. This is explained in section 2.2.

The reconvergent points and IRSs of branches can be determined statically using a compiler or dynamically using a hardware predictor. We borrow the Dynamic Reconvergence Predictor proposed by Collins, Tullsen, and Wang [9] augmented with IRS predictions as done by Al-Zawawi, Reddy, and Rotenberg [4]. The predictor is accessed at two different points in the pipeline, (i) when beginning construction of the CIDD recovery trace for a branch at the retirement stage (Section 2.2), and (ii) when preparing for possible recovery of a branch at the dispatch stage (Section 2.3). In either case, the predictor is indexed using the PC of the branch. The predictor supplies the predicted reconvergent PC and a predicted IRS for the branch. (The predictor may also supply other information for guiding selective application of control independence, such as the confidence of the reconvergent PC and IRS predictions and the length of the CD region of the branch [4].)

2.2 Pre-constructing CIDD Recovery Traces

Recovery traces of CIDD instructions are pre-constructed by monitoring the retired dynamic instruction stream at the retirement stage of the pipeline. Figure 2-2 shows the overall process divided into a sequence of four steps (labeled with block arrows 1 through 4).

The first step is performed when a branch is retired. The branch's PC is used to index the reconvergent point predictor. The predictor supplies the predicted reconvergent PC and IRS for the branch. The predicted reconvergent PC is then held, as shown in Figure 2-2, for detecting reconvergence of the CD region among instructions that follow. The IRS is also held, for later use by the trace constructor to identify CIDD instructions among future CI instructions, as shown in Figure 2-2.

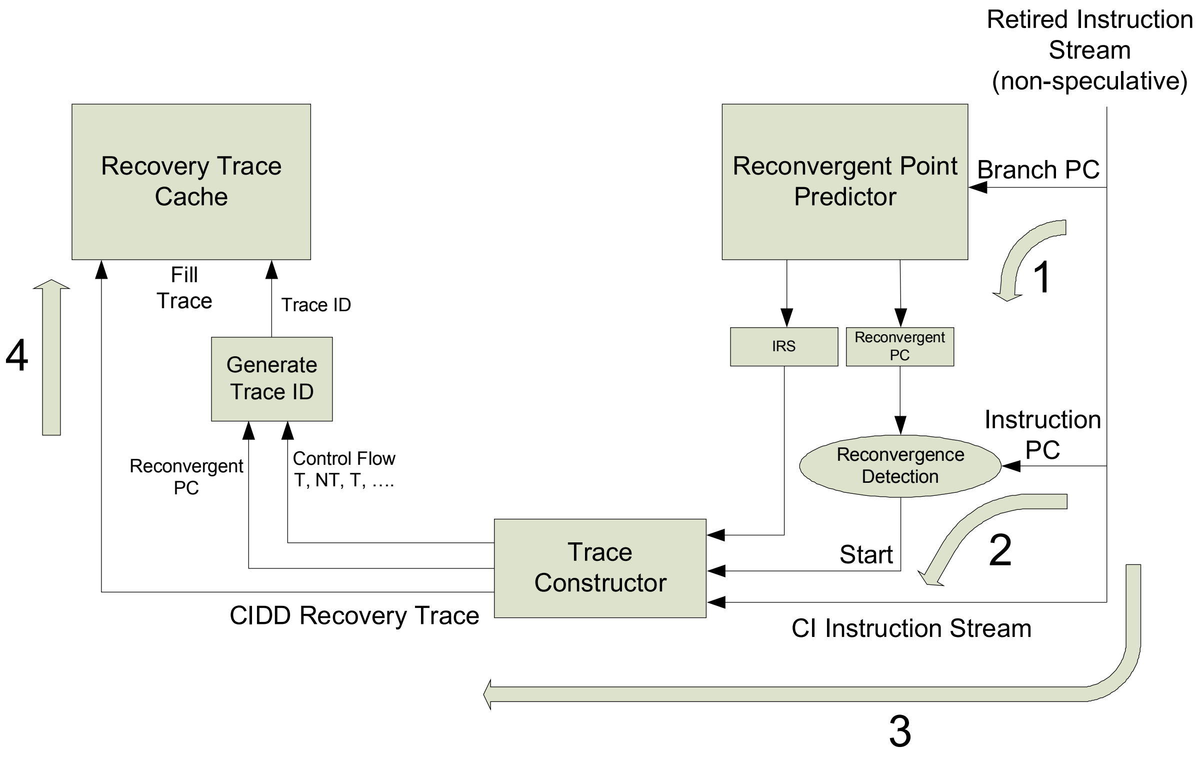


Figure 2-2: Pre-constructing CIDD recovery traces.

In the second step, retired instructions' PCs are checked against the reconvergent point that was set aside. Reconvergence of the CD region is detected when the next retired instruction PC matches the reconvergent PC. All instructions including and after the reconvergent point are considered to be CI with respect to the branch.

In the third step, after reconvergence is detected, CI instructions are examined. The trace constructor uses the IRS that was set aside previously to single out CIDD instructions among CI instructions. A CI instruction is CIDD if it depends directly or indirectly on registers potentially written to within the branch's CD region, as specified by the IRS. Direct and indirect CIDD instructions are easily identified using a logical register table of "poison" bits where logical registers specified in the IRS are initially poisoned [4]. An instruction is CIDD if any of its sources is poisoned. An instruction marks its poison status in the table entry corresponding to its logical destination register, propagating poison status to its dependents.

The identified CIDD instructions are grouped together to form the recovery trace. The trace constructor decides when to end the CIDD trace, based on the trace selection policy [28][29]. For the purpose of this study, we use two trace selection policies. In the first policy, a trace is ended when the CI region (encompassing both CIDI and CIDD instructions) reaches a maximum bound. The second policy also uses a maximum bound, but may prematurely end the trace at the first indirect branch. Notice that the bound is with respect to "logical trace length" which reflects the uncompressed CI region, as opposed to using a bound on the "physical trace length" which reflects the number of CIDD instructions only (compressed trace length).

During step 3, in addition to collecting the CIDD instructions, the trace constructor collects the branch outcomes (e.g., T, NT, T …) that characterize the control-flow of the CI region. The reconvergent PC and the sequence of branch outcomes through the CI region uniquely identify the recovery trace. We call this a trace ID [28][29]. The trace ID is used to index into the recovery trace cache for storing the constructed trace (assuming it is not already in the trace cache).

Finally, in step 4, when construction of the CIDD recovery trace has finished and the trace ID is now available, the trace cache is indexed using the trace ID and the new trace is stored into the corresponding location in the trace cache.

2.3 Preparing for Recovery

To recover from a branch misprediction, the recovery trace cache must be accessed to obtain the CIDD recovery trace corresponding to the current CI region fetched after the branch's reconvergent point. To do the access, some preparation is needed before the branch executes, i.e., before recovery. In particular, the trace ID needs to be generated so that the trace cache can be accessed later. Figure 2-3 shows the process of preparing for recovery, i.e., generating the trace ID corresponding to the CI region in the window. When the branch is dispatched, the reconvergent point predictor is accessed using the branch's PC. The predictor supplies the predicted reconvergent PC that terminates the CD region of the branch. After this, fetching proceeds as usual, until the reconvergent point is fetched. After reconvergence is detected, the branch predictions in the CI region are accumulated as control independent instructions are fetched. By combining the reconvergent PC and the cumulative branch predictions, a predicted trace ID is generated.

If the branch was in fact mispredicted, then the misprediction is detected when the branch executes. At this time, the predicted trace ID is used to access the trace cache, and the recovery trace is supplied if the trace cache hits. Selective recovery, in the case of a trace cache hit, is described in Section 2.4. In the case of a trace cache miss the processor falls back to conventional full-squash recovery.

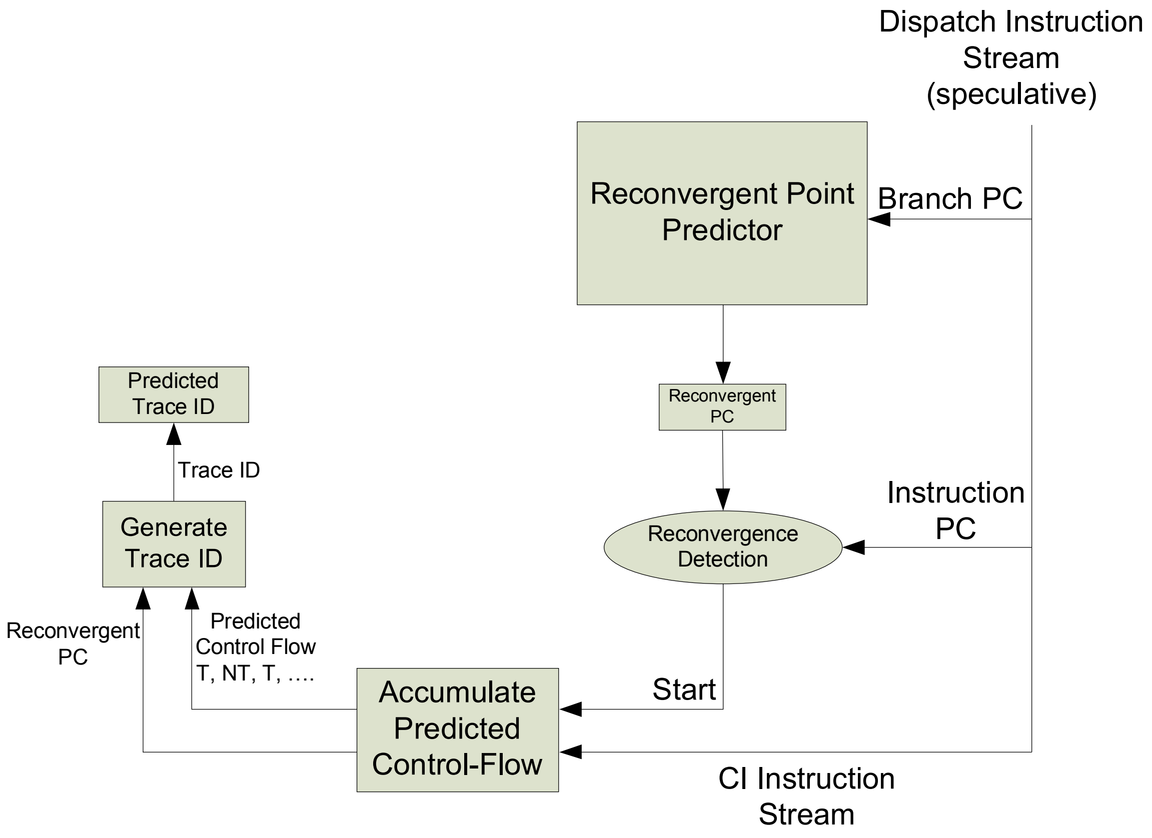


Figure 2-3: Preparing for recovery.

2.4 Recovery

2.4.1 Repairing the CD Region

As discussed previously, a checkpoint-based architecture provides a flexible substrate for inserting/removing instructions from the middle of the window. Whether before or after a branch misprediction is detected, corresponding incorrect CD instructions finish as usual and their physical registers are freed autonomously, except those registers tied to any checkpoint or still mapped in the rename map table.

When the misprediction is detected, the fetch unit begins fetching correct CD instructions. A second rename map, called the repair rename map, is initialized from the branch's checkpoint, and is used to rename correct CD instructions. Fetching correct CD instructions stops when the reconvergent point is reached. The reconvergent PC was retrieved during the recovery preparation step, as explained in Section 2.3.

2.4.2 Fetching and Executing the CIDD Recovery Traces

Similar to incorrect CD instructions, incorrect CIDD instructions finish as usual and any non-checkpointed and unmapped physical registers free autonomously. After repairing the CD region, the predicted trace ID generated during earlier preparations (Section 2.3) is used to access the trace cache for fetching the CIDD recovery trace, as shown in Figure 2-4. As CIDD instructions are fetched, they are renamed using the repair rename map which now reflects the corrected CD region.

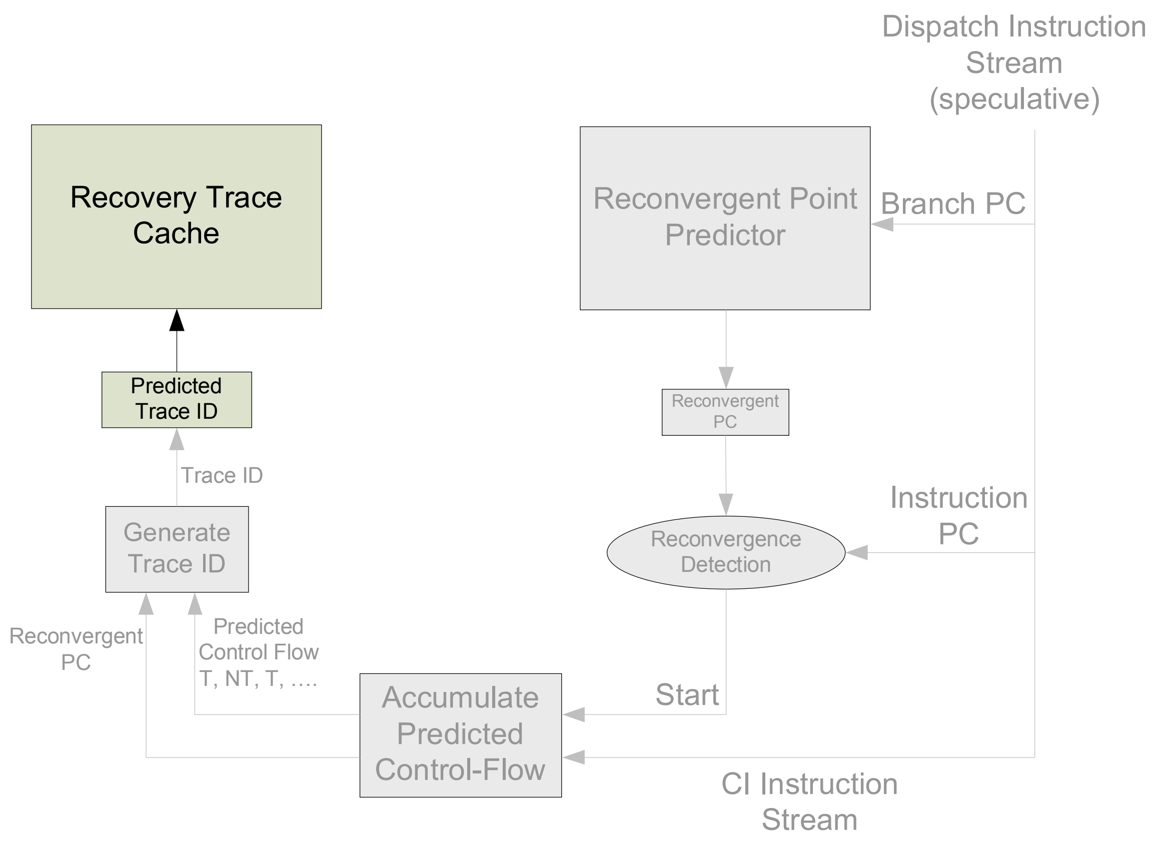


Figure 2-4: Repairing the CI region using the CIDD recovery trace from the trace cache.

To execute correctly, the trace's CIDD instructions must be linked to their correct producers. With respect to the re-dispatched CIDD instructions, there are three types of values, must be made available: (i) values produced by instructions before the branch or within the branch's CD region, (ii) values produced by other CIDD instructions in the trace, and (iii) values produced by CIDI instructions. Figure 2-5 shows the three types of data dependences explained through an example.

For register values produced before or within the CD region and consumed by a CIDD instruction in the recovery trace (dependence 1 in the figure), the recovery trace uses the repair rename map, which now reflects correct producers up to the reconvergent point.

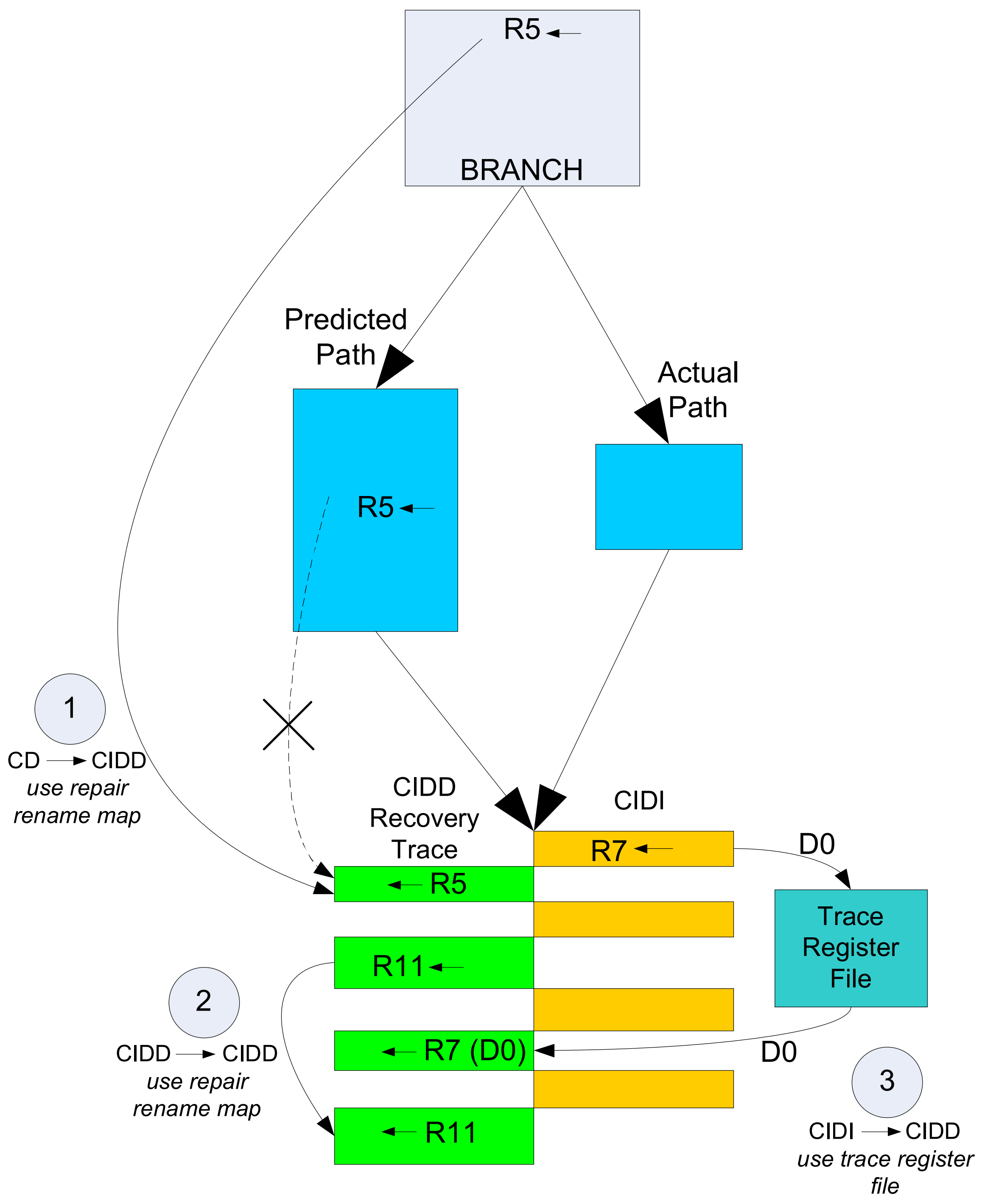


Figure 2-5: Three types of data dependences among CIDD instructions in the recovery trace.

In the example, the repair rename map (which was initialized from the branch's checkpoint) indicates that the correct producer of R5 is the one before the branch instruction.

Register values produced by CIDD instructions in the recovery trace and consumed by other CIDD instructions in the recovery trace (dependence 2 in the figure) are handled as usual by the repair rename map. Producers in the recovery trace update the repair rename map and consumers use these mappings. In the figure, the production and consumption of R11, localized within the recovery trace, is an example of the second type of dependence.

Register values produced by CIDI instructions and consumed by CIDD instructions in the recovery trace (dependence 3 in the figure) pose a more difficult problem. The problem with this type of dependence is that CIDI instructions may have already executed and their physical registers may have been freed via the aggressive register reclamation policy [3]. Therefore, these values must be saved in some alternative storage area and the CIDD instructions in the recovery trace must know how to link to them correctly in the event of a branch misprediction.

One solution is to write these values to a separate register file, one register file dedicated to each potential recovery trace in the window. We call these *trace register file*. CIDD instructions in a recovery trace are pre-renamed [30][37][39] such that source registers dependent on CIDI instructions (external to the trace) are linked to specific physical registers in the trace register file. During preparation for recovery (Section 2.3), once reconvergence is detected, all CI instructions not only write their values to the conventional physical register file, as usual, but also to a larger trace register file. The particular destination physical register in the trace register file is based on the distance of the producer CI instruction from the reconvergent point. So, in the example of Figure 2-5, the first CI instruction writes its value for R7 into the conventional physical register file (using its conventional physical register name), but it also writes this value to D0 in trace register file. D0 is selected based on the distance of the CI producer instruction from the reconvergent point, which is zero in the example. CIDD instructions in the recovery trace that consume this version of R7 link to the D0 value via pre-renaming, done during trace pre-construction. All CIDD instructions in the recovery trace that consume this version of R7 are pre-renamed to access D0 int trace register file.

Chapter 3

Related Work

3.1 Predication

Predication [17][21][22][27] attempts to reduce the misprediction penalty of difficult-to-predict forward branches. A difficult-to-predict branch is typically identified via profiling. Mispredictions are avoided altogether for this branch, by removing the branch and combining its control-dependent paths into one straight-line block. The branch is replaced with just the predicate calculation part of the original branch. To achieve selection of instructions from only one of the original paths, instructions from various paths are qualified by exclusive predicates.

At run-time, all paths of the original branch are fetched since the compiler merged them into a single straight-line block. After predicates are resolved, instructions from the correct path execute and instructions from other paths are discarded. In some predication schemes, instructions from all paths start executing before predicates are known. Only instructions with true predicates commit their values to the architectural state, whereas instructions with false predicates nullify their results.

Scheduling all paths and nullifying results of incorrect paths solves the problem of inserting/removing instructions from the middle of the window. In other words, predication removes control-flow and window management complexity associated with this control-flow.

The performance advantage of predication is that correct CD instructions are fetched and maybe executed before resolving the branch condition and without predicting the branch condition. However, this performance advantage must be weighed against two performance disadvantages. The first disadvantage is that predication wastes fetch bandwidth and maybe execution bandwidth with respect to conventional speculation, since both selected and non-selected instructions are scheduled. The second disadvantage is that CIDD instructions after the predicated region are delayed with respect to conventional speculation. CIDD instructions cannot execute speculatively since their source values are not forwarded from the predicated region until predicates resolve. This squanders performance with respect to conventional speculation, when speculation is correct.

In contrast to predication, microarchitectures that exploit control independence fully capitalize on correct branch predictions by speculatively forwarding data to CIDD instructions. Thus the benefits of conventional speculation are preserved. The advantage with respect to conventional speculation is not re-executing CIDI instructions after a branch misprediction.

3.2 Multipath Execution

Multipath execution [1][13][18][40] speculatively executes both paths of branches. To prune the total number of simultaneous paths, multipath execution is typically only applied to hard-to-predict branches as identified by a confidence estimator [15]. When an unconfidently predicted branch is encountered, a speculative thread is spawned to fetch and execute the non-predicted path in parallel with the predicted path. When the branch is resolved, the incorrect thread is squashed. Different policies have been proposed for controlling the number and quality of spawned threads. For example, TME [40] spawns alternate threads only from the predicted path and SDPE [13] limits the number of simultaneous threads to only two.

Multipath execution reduces the penalty of a mispredicted branch because the prediction is hedged by also fetching and executing the non-predicted path. However, multipath execution degrades performance with respect to conventional speculation when the branch prediction is correct, since the correct path competes for fetch and execution bandwidth with the incorrect path.

3.3 Control Independence Techniques

Multiscalar Processors [36] and other speculative multithreading architectures [2][23][26] divide a sequential program into multiple speculatively-parallel tasks. The multiple tasks are fetched and executed in parallel. In Multiscalar, a branch misprediction in one task does not squash future tasks if the future tasks are control independent of the mispredicted branch. However, if the register forwarding policy is to speculatively forward branch-influenced values among tasks, then a branch misprediction in one task may squash future control-independent tasks. The reason is that Multiscalar does not implement selective recovery with respect to data-flow speculation. Thus the original register forwarding policy is to forward inter-task register values only when branches influencing these values are resolved. This has the downside of delaying CIDD instructions in future tasks until branches in previous tasks resolve.

Dynamic Multithreading [2] introduces selective recovery capabilities in the context of speculative multithreading. DMT creates speculative threads at the return points of functions and exit points of loops. Thus, DMT implicitly exploits control independence at these coarse-grain reconvergent points. To facilitate selective re-execution of only CIDD instructions after these reconvergent points, all instructions executed by a thread must be kept in a trace buffer (there is one trace buffer per speculative thread). Selective re-execution of CIDD instructions is performed by re-sequencing through all the speculative instructions in the trace buffer and only instructions that depend on a misprediction get re-dispatched into the pipeline. In contrast, our trace-cache-based approach does not need to re-sequence through all the CI instructions just to single out the CIDD instructions. Moreover, constructing the "recovery sub-program" is done only once, on the fill-side of the recovery trace cache.

Rotenberg and Smith advocate using trace processors for efficiently exploiting control independence [32]. Fine-grain control independence (FGCI) is defined for exploiting control independence with respect to arbitrary nested `if-else` constructs. FGCI targets relatively short CD regions that fit within a single trace, thus, exploiting FGCI only requires replacing the trace in the affected PE. They also define coarse-grain control independence (CGCI) for other branches that cannot be covered with FGCI. CGCI uses global reconvergent points to cover many branches. Global reconvergent points include the backward edges of loops, loop exit points, and function return points. In the case of CGCI, multiple CD traces may need to be inserted/removed from the middle of the window. To achieve this, the trace processor's processing elements (PEs) are managed as a linked list for arbitrary trace insertion/removal. For both FGCI and CGCI, after repairing the CD trace(s), all CI traces are re-dispatched to identify and re-issue direct CIDD instructions. Then, indirect CIDD instructions re-issue automatically via the existing issue logic. This is possible because all traces remain in the PEs until retirement, ready to re-issue instructions as needed. The fact that all CI traces must be redispatched, places trace processors among the class of control independence architectures that must resequence through all CI instructions to identify CIDD instructions.

The instruction reuse buffer [35] records the result values of instructions along with the source values that produced the results. The reuse buffer is consulted when an instruction is dispatched. If the instruction hits in the reuse buffer and its current source values match the previous source values in the reuse buffer, the result value in the reuse buffer can be reused. In this case, the instruction bypasses the execution core and writes its reusable value to the register file. Instruction reuse implicitly exploits control independence in the form of squash reuse. All instructions after a mispredicted branch are squashed. Nonetheless, if a squashed CIDI instruction executed and wrote its result value to the reuse buffer before the mispredicted branch resolved, then the CIDI instruction is not re-executed when it is re-fetched. However, all CI instructions still need to be re-fetched even if not all of them need to be re-executed. This places squash reuse among control independence architectures that must resequence through all CI instructions to single out CIDD instructions. The dual-ROB approach proposed by Chou, Fung, and Shen features a secondary ROB that serves as a FIFO squash reuse buffer [8].

In the Skipper microarchitecture [7], if a candidate branch is unconfidently predicted, the processor skips fetching of the branch's CD region. In this case, the fetch unit is redirected to the CI region early. When a CD region is skipped, the processor creates gaps in the middle of the ROB and the load/store queues to allow skipped CD instructions to be filled in after resolving the branch. The size of the gap is the maximum length among all paths (similar to how trace processors pad traces for FGCI branches). As mentioned, the CI region is fetched early, before the skipped CD region. Among the CI instructions, only CIDI instructions can execute before the branch resolves. Similar to predication, CIDD instructions are delayed because they depend on values that are influenced by the deferred CD region.

Gandhi, Akkary, and Srinivasan target only mispredicted branches that exhibit exact convergence [12]. In exact convergence, the correct target of the mispredicted branch is the reconvergent point itself. Thus, repairing the mispredicted branch does not require inserting any instructions in the middle of the window, greatly simplifying exploiting of control independence. However, the effects of incorrect CD instructions must be reversed. They propose converting each incorrect CD instruction into a move instruction, that copies the value from the physical register that was unmapped by the instruction to the instruction's own physical register. When the CD instructions re-execute, this time as move instructions, CIDD instructions re-execute. This requires buffering all CI instructions, either in the issue queue or in separate replay buffers. They suggest implementing selective re-issuing by one of three means: (1) all instructions remain in the issue queue until they are non-speculative, or (2) all CI instructions are placed in a replay buffer and are scanned to single out CIDD instructions for re-injection into the pipeline, or (3) all CI instructions are placed in a replay buffer and dependence bit vectors are used to identify CIDD instructions directly without scanning.

Zilles, Emer, and Sohi propose exploiting control independence in the context of exception handling [41]. They propose executing the exception handler in a separate hardware context and not squashing instructions after the excepting instruction. All instructions after the excepting instruction are CI instructions, and many of these are CIDI instructions.

This thesis is heavily influenced by a precursor project [4] that leverages the Continual Flow Pipeline (CFP) [38] as a convenient substrate for tolerating branch mispredictions. The CFP-CI microarchitecture [4] essentially constructs CIDD traces on-the-fly within CFP's slice data buffer (SDB). When a mispredicted branch is detected, the correct CD instructions are fetched from the instruction cache followed by the branch's CIDD instructions from the SDB. The SDB and recovery trace cache play the same role of supplying compressed recovery traces. The key distinction is that my approach does not repeatedly construct recovery traces. Rather, each unique recovery trace is pre-constructed once and cached. A trace-cache-based approach potentially simplifies exploiting control independence compared to an SDB-based approach.

3.4 Exploiting Caches for Faster Misprediction Recovery

Bondi, Nanda, and Dutta proposed the Misprediction Recovery Cache (MRC) [5] to reduce the pipeline refill penalty following a mispredicted branch. The technique is particularly beneficial in the context of CISC ISAs (e.g., x86), which often require many fetch and decode pipeline stages. When a mispredicted branch is detected and the fetch unit is redirected to the branch's correct target, a trace of decoded correct-path instructions is accumulated in the MRC. If the same misprediction recurs in the future, the MRC is consulted in parallel with redirecting the fetch unit. If there is a hit in the MRC, it supplies the already decoded instructions, significantly reducing or eliminating the deep pipeline refill penalty. The MRC does not exploit control independence because all instructions after the mispredicted branch are still re-fetched and re-executed. Nonetheless, the time required to re-fetch and re-decode both CD and CI instructions is reduced.

Chapter 4

Evaluation Methodology

4.1 Simulation Environment

I developed a custom trace-driven simulator, using the Simplescalar toolset [6]. The simulator generates three types of results:

(i) The breakdown of all retired instructions, that shows the percentages of CIDI instructions, CIDD instructions, CD instructions, and remaining instructions that are not in the shadow of mispredictions.

(ii) The contribution of each unique recovery trace to the total number of saved CIDI instructions.

(iii) The hit ratios of recovery trace caches.

Timing and hardware resources are not modeled, except for the recovery trace cache. The recovery trace cache is modeled in order to measure its hit ratio and how hit ratio affects actual CIDI instruction savings.

Although a trace-driven simulator is used, the simulator still fetches instructions on the wrong paths of mispredicted branches, specifically within the CI regions of branches. A predicted recovery trace, which is based on branch predictions in the CI region, may itself contain partially incorrect control-flow. Incorrect CD instructions corresponding to other mispredictions within the CI region of the branch being serviced, are not counted among the branch's CIDI and CIDD instructions. Modeling wrong control-flow within predicted recovery traces will more accurately reflect the number of retired CIDI and CIDD instructions with respect to the mispredicted branch being serviced.

4.2 Description of Results Generated by the Simulator

4.2.1 Breakdown of Retired Instructions

The first set of results, that will be presented in the results chapter, is a breakdown of retired dynamic instructions into different categories. As shown in Figure 4-1, dynamic instructions are first divided into two categories. Dynamic instructions that were fetched after one or more unresolved mispredicted branches in the window, are considered to be in the shadow of a branch misprediction. These instructions are targeted for savings by any control independence architecture. Instructions that were fetched when there were no prior unresolved mispredicted branches are not considered to be in the shadow of a branch misprediction. As such, these instructions do not represent opportunity for applying control independence, as there are no prior mispredictions.

Dynamic instructions in the shadows of branch mispredictions are further divided into CD instructions and CI instructions. CI instructions are further broken down into those that are logically part of traces that either hit or miss in the trace cache. CI instructions that hit in the trace cache are then classified as either data independent (DI) or data dependent (DD) with respect to the prior branch misprediction.

On the right-hand side of Figure 4-1 are shown the labels used in the results section to convey each category: “not in misp. shadow”, “CD”, “CIDD”, and “CIDI”. The CIDI category represents the savings, in terms of not needlessly re-executing CIDI instructions. Only CIDD instructions are re-executed efficiently using the recovery trace cache.

Note that the “CD” label refers to both truly CD instructions as well as CI instructions that are not covered by the trace cache due to a trace miss. In the case of a miss, the proposed architecture cannot discriminate between CI and CD instructions, that is, all instructions in the shadow of the misprediction are deemed CD (even the CI instructions). In other words, on a trace cache miss, conventional full-squash recovery is used thereby handling CI instructions as CD instructions.

A subtle point is that an instruction may be in the shadows of multiple branch mispredictions. Note that all measures are with respect to the last misprediction prior to an instruction. Thus, for example, CIDI consists of instructions that were CIDI at least with respect to the last prior misprediction and maybe more.

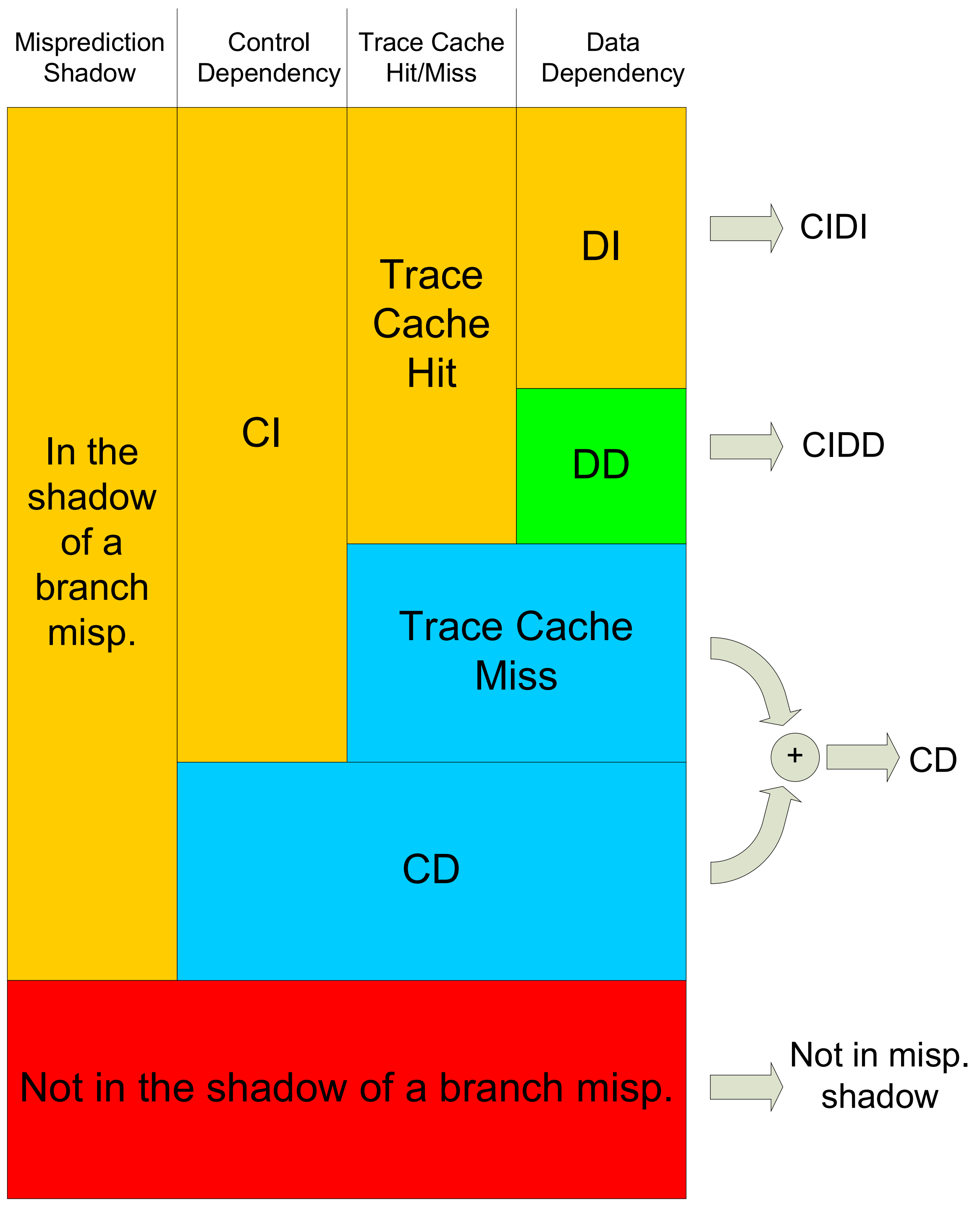


Figure 4-1: Breakdown of retired instructions.

The above breakdown is misleading in the context of a mispredicted branch that has a very long incorrect CD path. The mispredicted branch is likely to be resolved before the fetch unit even reaches the reconvergent point along the long incorrect path. In this case, the CI instructions are not truly in the shadow of a misprediction (not fetched yet) and as such it is misleading to classify them as either CIDI or CIDD.

To more closely represent a mispredicted branch with a long incorrect CD path, its CI instructions are not marked as CIDD or CIDI with respect to *this* branch, rather they are marked as "not in misp. shadow" since they will most likely be fetched after the mispredicted branch is resolved. We define a long incorrect CD path to be 128 or more instructions. The 128-instruction threshold approximates the number of incorrect CD instructions that can be fetched in a 4-issue processor assuming a fetch-to-resolve branch misprediction penalty of 32 cycles.

4.2.2 Studying Locality: Contributions of Individual Recovery Traces

In the previous section, the CIDI category represents the total number of instruction savings due to exploiting control independence. A moderately sized trace cache will only be effective if relatively few unique recovery traces (e.g., 10%) contribute most of the CIDI instruction savings (e.g., 90%). Accordingly, the second set of results characterizes the contributions of individual recovery traces to the total number of CIDI instructions. The contribution of a unique recovery trace is the number of CIDI instructions after the reconvergent point that are not re-executed when the recovery trace is used, multiplied by the number of times the trace is used for recovery. After determining individual contributions, traces are sorted in descending order from highest to lowest contribution.

4.2.3 Trace Cache Hit Ratio

The actual achievable CIDI instruction savings depends on the trace cache hit ratio. The last set of results presents hit ratios, for different trace cache configurations.

4.3 Benchmarks

Table 4-1 shows the benchmarks used for this study. Ten of the SPEC2K integer benchmarks are simulated. The remaining two SPEC2K integer benchmarks (eon and crafty) did not compile using the Simplescalar gcc compiler because of compatibility issues. None of the SPEC2K floating-point benchmarks are included since they have low misprediction rates, hence there is little need for exploiting control independence. The ref inputs are used for all benchmarks, with specific parameters shown in Table 4-1.

For each benchmark, the first one billion instructions are skipped without warming the recovery trace cache or the branch predictor. The next 100 million instructions are simulated.

Table 4-1 also shows the branch misprediction rates for the benchmarks using a gshare branch predictor [25], with a 64K-entry pattern history table and a 4K-entry branch target buffer.

Table 4-1: SPEC2K benchmarks.

Benchmark	Input Dataset	Conditional Branch Misprediction Rate
bzip2	input.program 58	0.5%
gap	-l ./lib.gap –q –m 64M ref.in	3.0%
gcc	expr.i –O3 –o expr.s	4.7%
gzip	input.program 16	9.5%
mcf	inp.in	3.8%
parser	2.1.dict –batch ref.in	4.2%
perlbmk	-I./lib splitmail.pl 850 5 19 18 1500	0.6%
twolf	ref	11.2%
vortex	lendian1.raw	1.1%
vpr	net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2	8.4%

Chapter 5

Results and Analysis

In this chapter, I will present the results of my study. Four types of results are presented. First, retired dynamic instructions are broken down into the four categories that were discussed in Section 4.2.1, for an unbounded recovery trace cache. Second, contributions of individual recovery traces to total CIDI instruction savings are presented. Third, recovery trace cache hit ratios and retired dynamic instruction breakdowns are shown for different finite-size trace cache configurations. Finally, an example of control independence from the twolf benchmark is studied in depth.

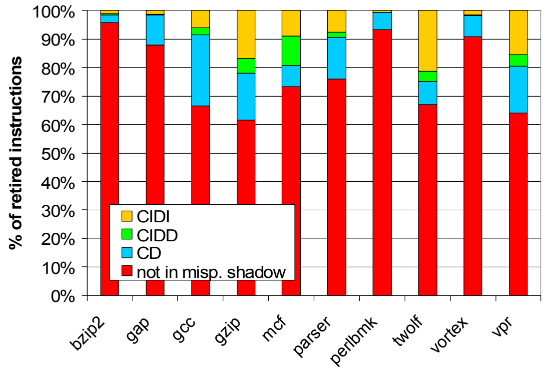
5.1 Breakdown of Retired Dynamic Instructions

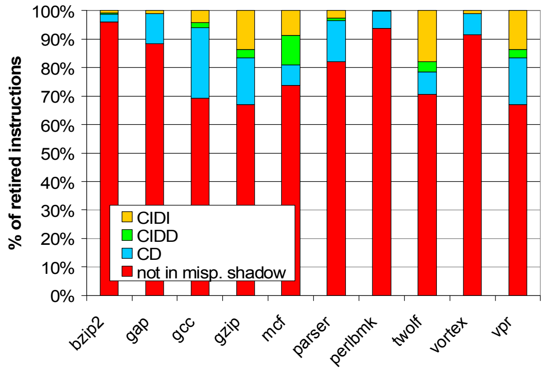
As explained in Chapter 4, all retired dynamic instructions are classified into one of four categories, according to their control and data dependences with respect to prior branch mispredictions. For this set of results, an unbounded trace cache is used to study the intrinsic behavior of benchmarks. The logical trace length and the trace selection policy are varied.

The logical trace length is the total number of CI instructions after the reconvergent point, from which CIDI savings will be exploited. For example, if the logical trace length is 64 instructions, this means that the physical trace length (CIDD instructions only) will vary from zero to 64. When the physical trace length is zero, there are no CIDD instructions in the trace and all CI instructions after the reconvergent point are CIDI instructions (100% savings). On the other hand, if the physical trace length is 64 instructions, then all CI instructions after the reconvergent point are CIDD instructions and there are no CIDI instructions (0% savings).

Figure 5-1, Figure 5-2, and Figure 5-3 show the breakdown of retired dynamic instructions for an unbounded trace cache, with logical trace lengths of 32, 64, and 128 instructions, respectively. For each figure, two trace selection policies are shown. The default policy ends traces at the maximum logical trace length (32, 64, or 128 instructions). The modified policy ends traces at either the maximum logical trace length or the first indirect branch.

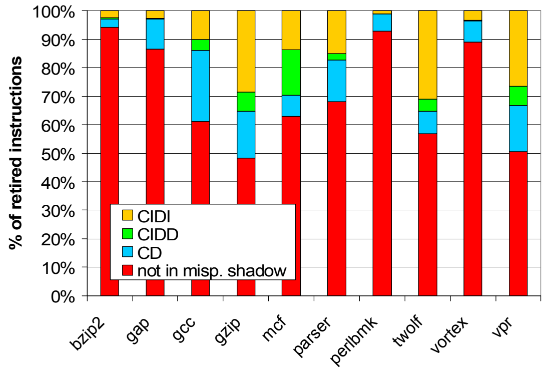
First, I will discuss CIDI savings for a specific logical trace length and trace selection policy. Then I will explain how CIDI savings change with different logical trace lengths and selection policies. For the case of a logical trace length of 64 instructions and default trace selection policy, CIDI savings vary among benchmarks from 1% (perlbmk) to 31% (twolf). Referring to Table 4-1, notice that twolf has the highest branch misprediction rate of 11.2% while perlbmk has the very low misprediction rate of 0.6%. Based on misprediction rate, there is more opportunity to exploit control independence in twolf than perlbmk. Twolf shows a large percentage of instructions in the shadow of mispredictions (43%) whereas perlbmk has a much smaller percentage (7%). Twolf achieves 31% CIDI savings among 43% of instructions in the shadows of mispredictions, indicating a large amount of control independence.

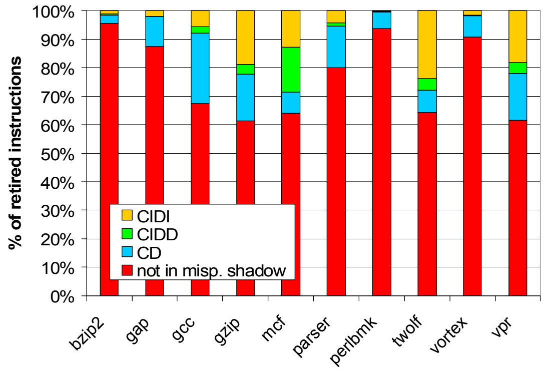




(a) default trace selection policy (b) modified trace selection policy

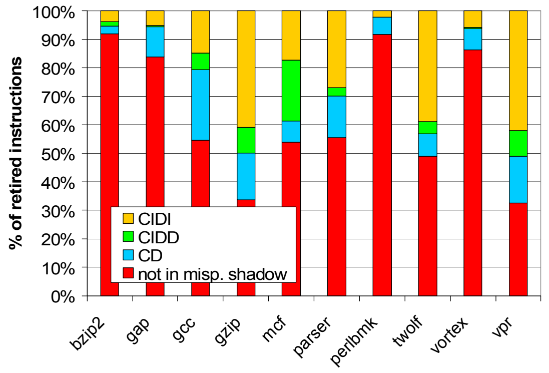
Figure 5-1: Unbounded trace cache, logical trace length = 32 instructions.

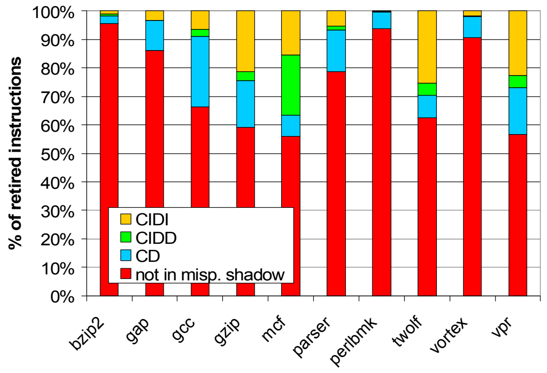




(a) default trace selection policy (b) modified trace selection policy

Figure 5-2: Unbounded trace cache, logical trace length = 64 instructions.





(a) default trace selection policy (b) modified trace selection policy

Figure 5-3: Unbounded trace cache, logical trace length = 128 instructions.

Using the modified trace selection policy (end at indirect branch) reduces CIDI savings. For example, in twolf, CIDI savings reduce from 31% to 24%. This difference is due to the fact that traces are smaller for the second policy. This shrinks the CI region from which the processor can extract CIDI savings. This also increases the percentage of instructions that are not in the shadow of branch mispredictions, since ending traces earlier appears to move some instructions out of the shadow of branch mispredictions (they are squashed and considered again later). Increasing the logical trace length tends to increase CIDI savings for the same reason. Increasing trace length appears to extend the shadows of branch mispredictions. For example, for vpr, increasing trace length from 64 to 128 increases CIDI savings from 26% to 41% (corresponding to 49% and 67% of instructions in misprediction shadows, respectively).

5.2 Contribution of Traces to CIDI Savings

This section shows the contributions of individual recovery traces to the total amount of CIDI instruction savings. For each unique trace used for recovery, the number of CIDI instructions is recorded and multiplied by how many times the trace was used for recovery. This product is the trace's individual contribution to overall CIDI savings. Traces are then sorted in descending order based on their contributions. In the graphs that follow, the x-axis shows individual traces sorted based on their contributions and the y-axis shows the cumulative contribution of the traces with respect to the total savings. Figure 5-4 to Figure 5-13 show the contribution graphs for the ten benchmarks. The logical trace length is 64 instructions and the default trace selection policy is used.

For the trace cache to perform well (high hit ratio), high locality is needed. High locality exists if relatively few traces (e.g., 20% of all traces) contribute a high percentage of the total savings (e.g., 80% of total savings).

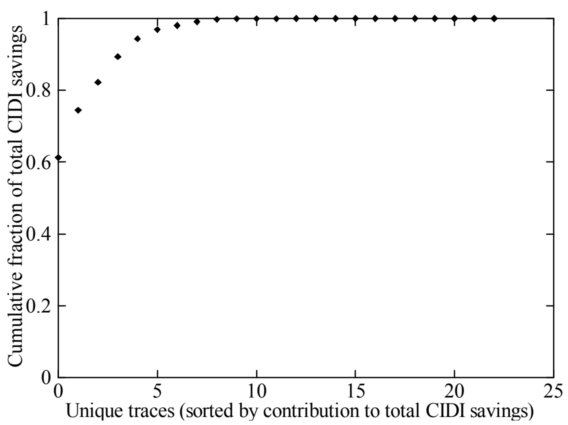


Figure 5-4: Contributions of unique traces (bzip2).

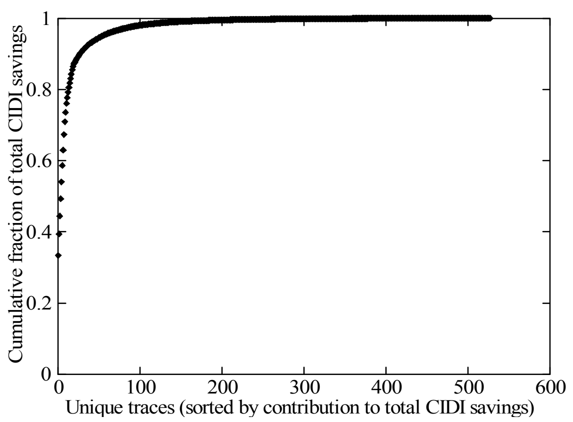


Figure 5-5: Contributions of unique traces (gap).

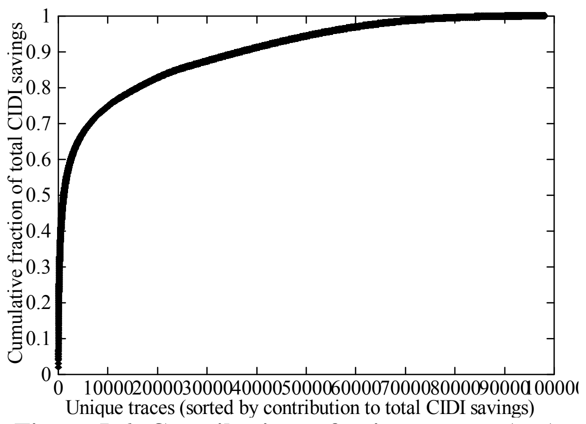


Figure 5-6: Contributions of unique traces (gcc).

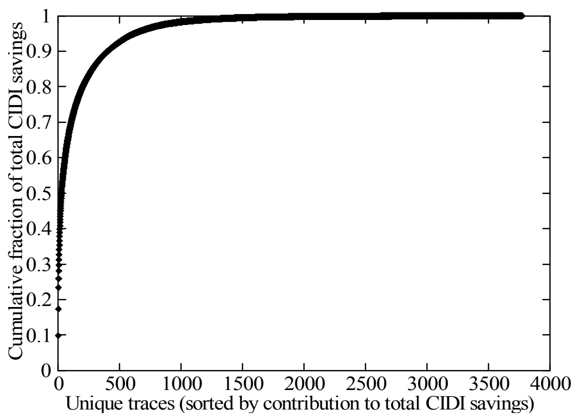


Figure 5-7: Contributions of unique traces (gzip).

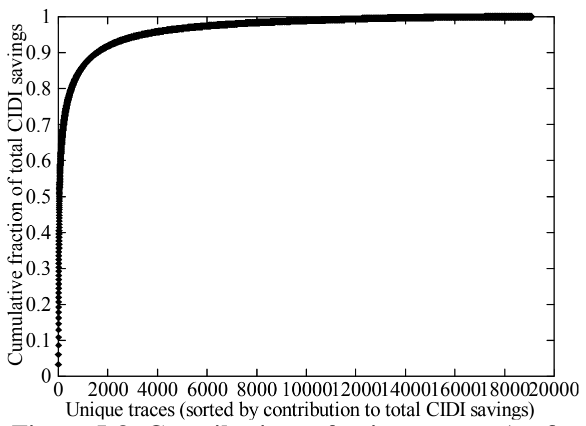


Figure 5-8: Contributions of unique traces (mcf).

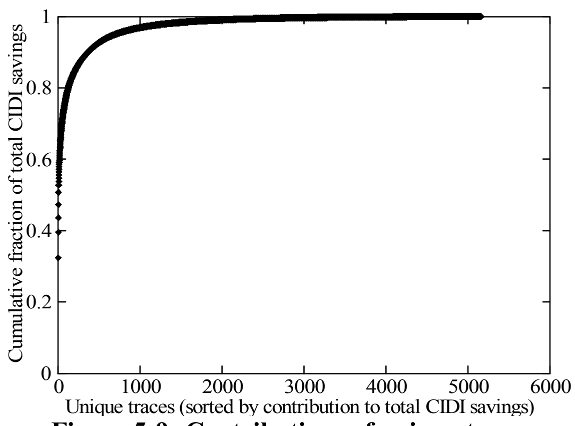


Figure 5-9: Contributions of unique traces (parser).

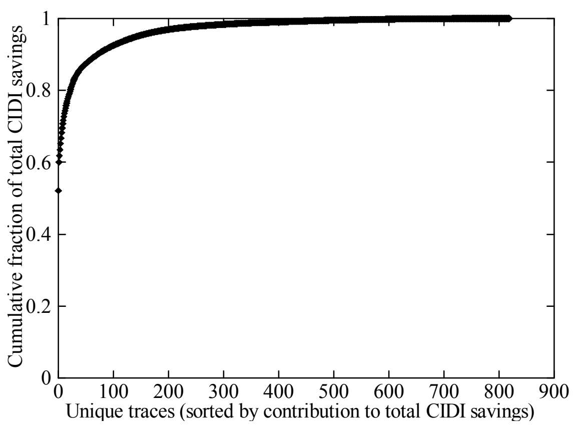


Figure 5-10: Contributions of unique traces (perlbmk).

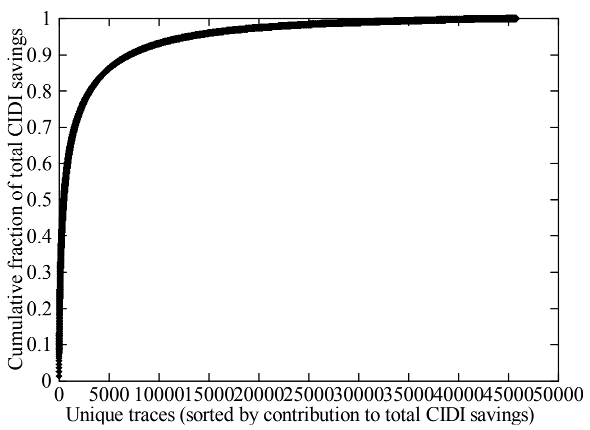


Figure 5-11: Contributions of unique traces (twolf).

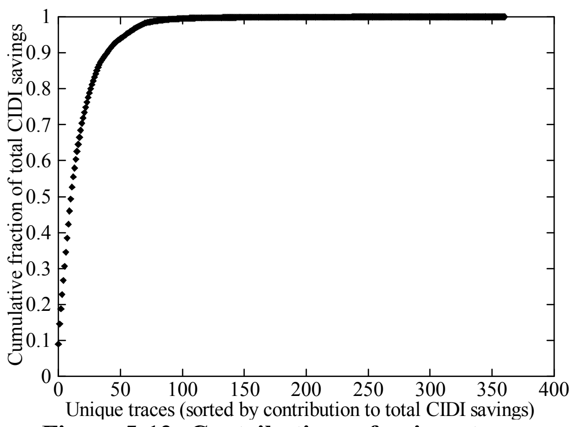


Figure 5-12: Contributions of unique traces (vortex).

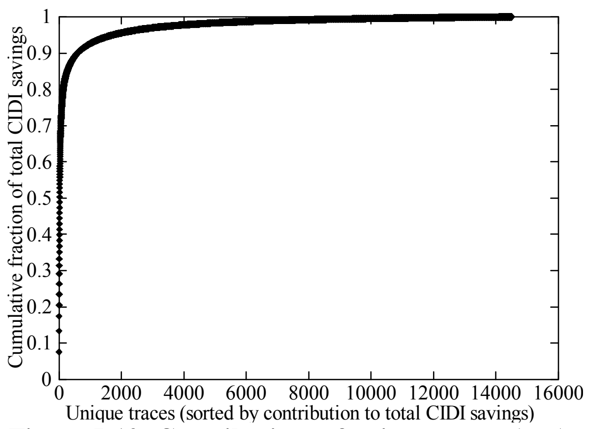


Figure 5-13: Contributions of unique traces (vpr).

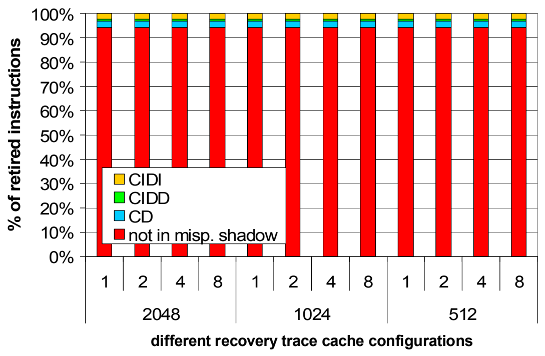
The results show that there is a high degree of locality in all benchmarks. Nonetheless, the number of top-contributing traces needed to achieve 80% savings varies among the benchmarks, ranging from 3 traces for bzip2 to 16,000 traces for gcc. Thus, the required trace cache capacity likely varies among the benchmarks.

These graphs can be used as a guide for performance only if the percentage of CIDI instructions among retired instructions is substantial. For example, for bzip2 and perlbmk, although only 3 and 23 traces are needed to achieve 80% of total CIDI savings, the percentage of CIDI instructions among all retired instructions is very small anyway (2.32% and 1.1%, respectively).

Analysis of twolf is more meaningful since, unlike bzip2 and perlbmk, it has a high percentage of CIDI instructions among all retired instructions (31%). Twolf achieves 80% of all CIDI savings with the top 3,000 unique traces, which constitute only 6.7% of all the unique traces in twolf. This gives some indication of the required trace cache capacity for twolf.

5.3 Trace Cache Hit Ratio

Figure 5-14 to Figure 5-23 show the behavior of the ten benchmarks with various finite-sized trace cache configurations. For each benchmark, the graph on the left shows the breakdown of retired dynamic instructions and the graph on the right shows trace cache hit ratios. For both graphs, the x-axis shows different trace cache configurations. The number of trace cache lines is varied from 2048 lines to 512 lines. For each such configuration, set-associativity is varied. For example, for the 2048-line configuration, 1 is a direct-mapped trace cache with 2048 sets, whereas 8 is an 8-way set-associative trace cache with 256 sets. In both cases the number of lines is 2048. The hit ratio is the fraction of times the trace cache hits when a trace is required for recovery.



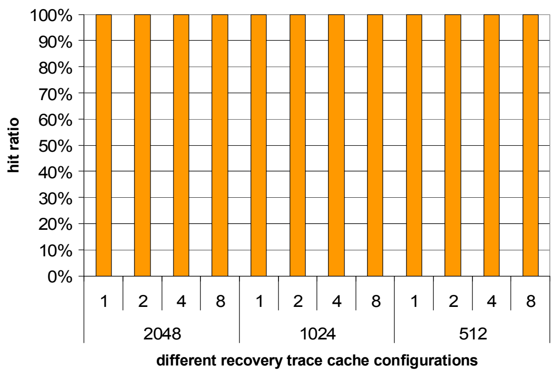
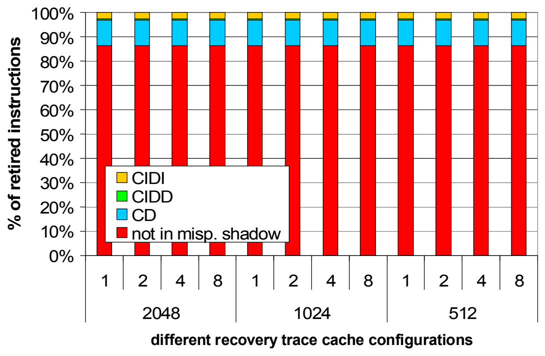


Figure 5-14: Recovery trace cache results for bzip2: instruction breakdown (left), hit ratios (right).



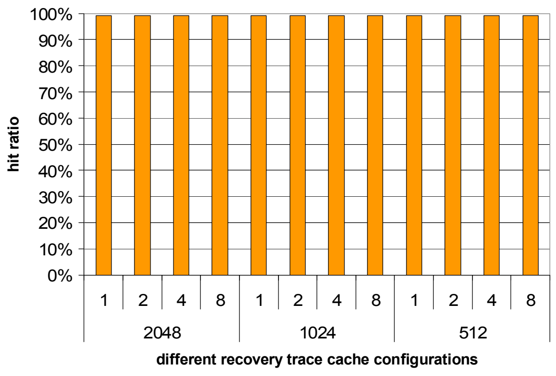
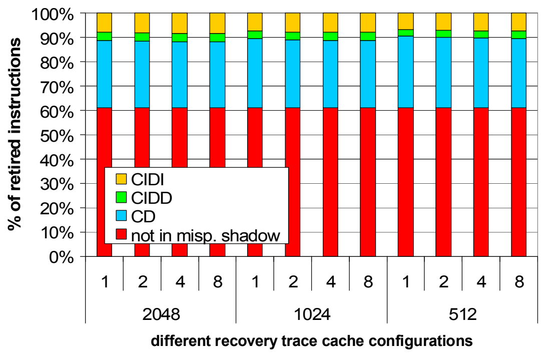


Figure 5-15: Recovery trace cache results for gap: instruction breakdown (left), hit ratios (right).



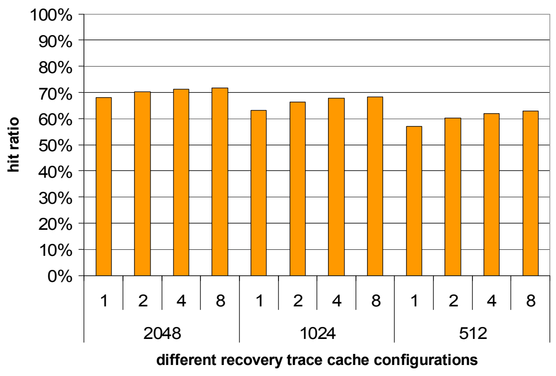
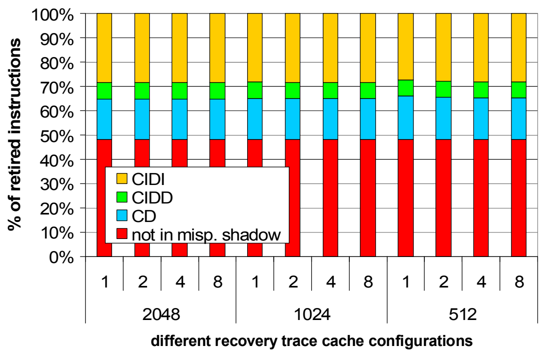


Figure 5-16: Recovery trace cache results for gcc: instruction breakdown (left), hit ratios (right).



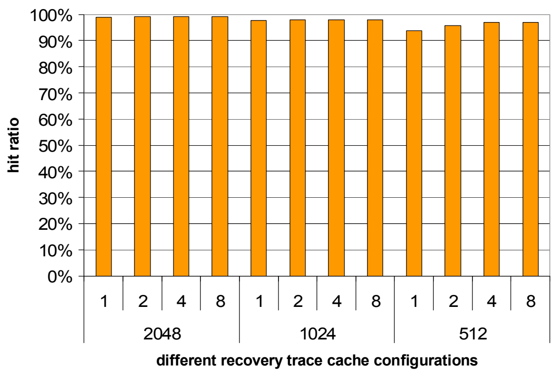
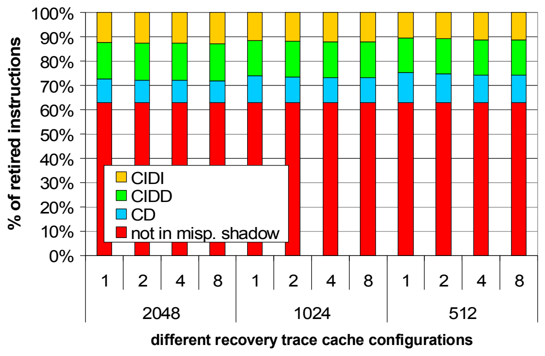


Figure 5-17: Recovery trace cache results for gzip: instruction breakdown (left), hit ratios (right).



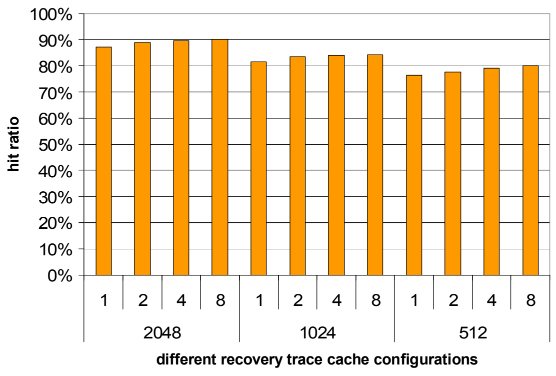
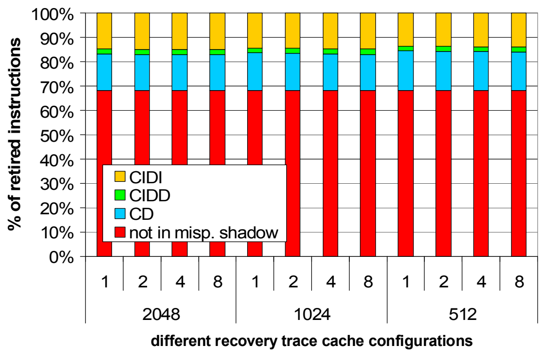


Figure 5-18: Recovery trace cache results for mcf: instruction breakdown (left), hit ratios (right).



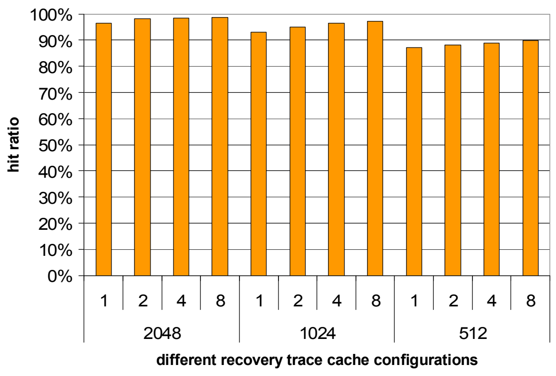
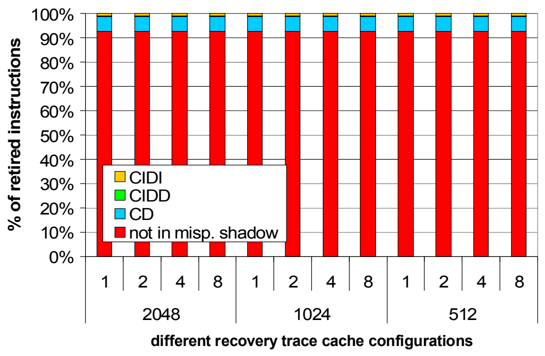


Figure 5-19: Recovery trace cache results for parser: instruction breakdown (left), hit ratios (right).



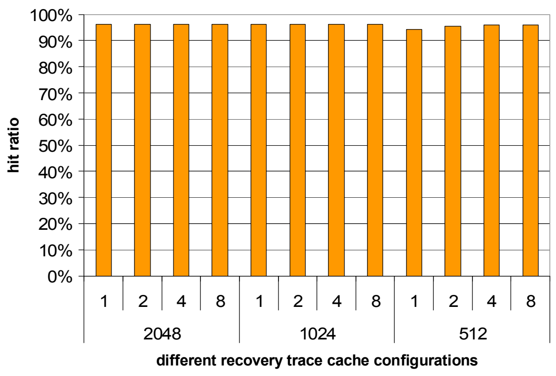
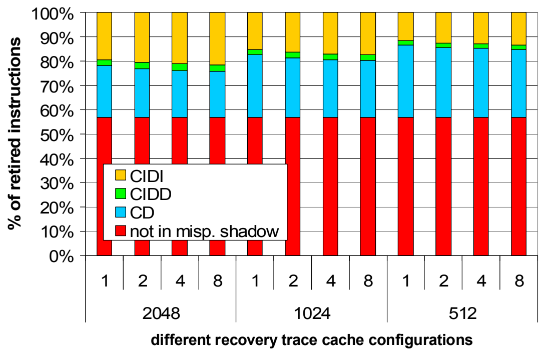


Figure 5-20: Recovery trace cache results for perlbmk: instruction breakdown (left), hit ratios (right).



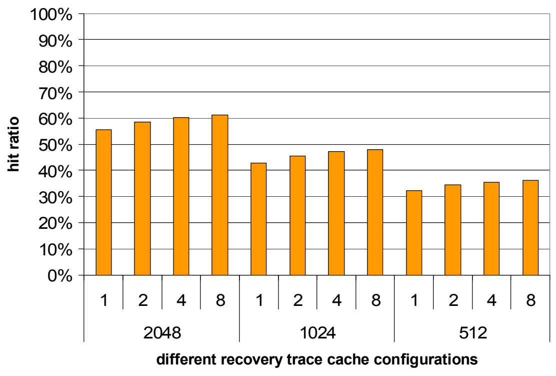
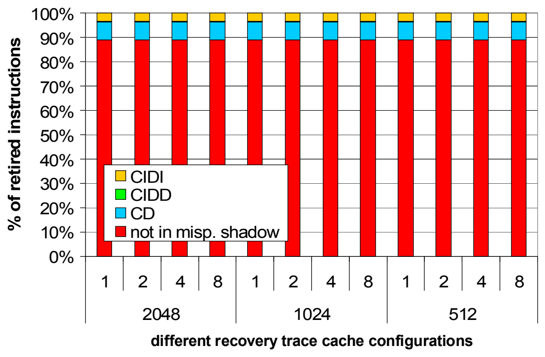


Figure 5-21: Recovery trace cache results for twolf: instruction breakdown (left), hit ratios (right).



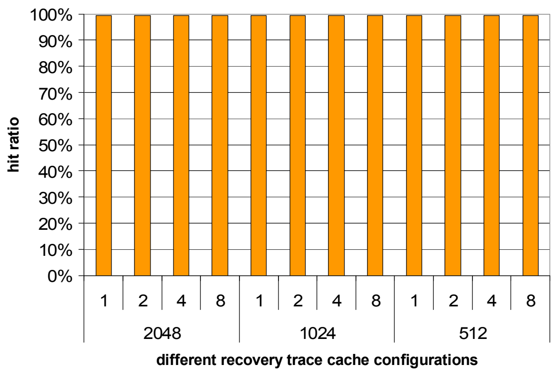
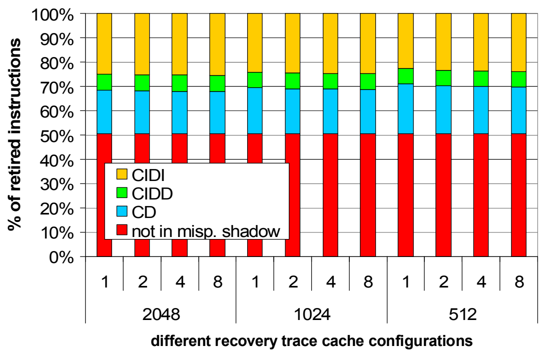


Figure 5-22: Recovery trace cache results for vortex: instruction breakdown (left), hit ratios (right).



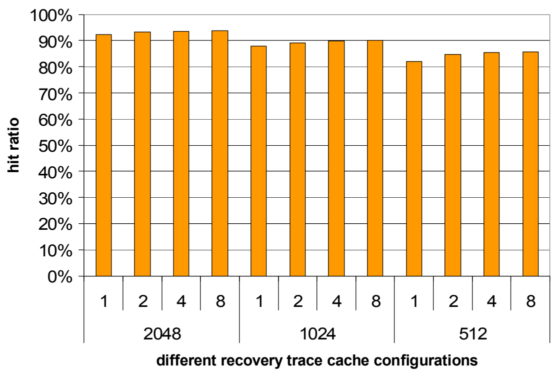


Figure 5-23: Recovery trace cache results for vpr: instruction breakdown (left), hit ratios (right).

The first thing to notice in the breakdown graphs is that the number of instructions that are not in the shadow of a branch misprediction does not change with trace cache configuration since it does not depend on whether or not recovery traces hit or miss in the cache.

Next, decreasing the size and/or associativity of the cache decreases the number of CIDI and CIDD instructions. This is due to the fact that, if a trace misses in the cache, then all instructions after the branch misprediction will be considered CD on the branch. This was explained in Section 4.2.1: the processor resorts to full-squash recovery if a recovery trace is not available. Notice that the decrease in CIDI/CIDD corresponds to an increase in CD. For benchmarks that construct many unique recovery traces, increasing cache size and/or associativity increases the percentage of CIDI instruction savings. This trend is noticeable for gcc, mcf, parser, twolf, and vpr. In contrast, bzip2, gap, gzip, perlbmk, and vortex construct fewer unique recovery traces, thus increasing cache size and/or associativity does not yield a noticeable increase in CIDI instruction savings.

Trace cache hit ratios follow the same trend. Hit ratios are only affected by size and/or associativity, for those benchmarks that require many recovery traces. For example, gcc, mcf, parser, twolf, and vpr require many traces. These benchmarks show moderate to high sensitivity to changing cache size and associativity.

5.4 Measuring Memory Violations

All previous results assume branches only influence CI instructions through register dependences. Yet, branches also influence CI instructions through memory dependences. We assume loads issue speculatively. To account for branch-influenced memory dependences, we measure the frequency of possible mispredicted CI loads. A CI load may get a wrong value if there is a prior store to the same address in the branch's CD region (because the store may be fetched late), or if there is a prior CIDD store to the same address (because the store must re-execute).

Figure 5-24 shows a breakdown of retired load instructions into three categories: (i) "depend on CD store" shows the percent of loads that depend on CD stores, (ii) "depend on CIDD store" shows the percent of loads that depend on CIDD stores, (iii) "depend on CIDI/other store" shows the percent of loads that depend on CIDI stores or stores before the branch. The first two categories represent mispredicted loads.

The figure shows that, for all benchmarks, almost all load instructions depend on branch-independent store instructions ("depend on CIDI/other store"). For gzip, 2% of load instructions depend on store instructions within the CD region. Some benchmarks (e.g., gzip, parser, twolf, and vpr) show small percentages of load instructions that depend on CIDD stores. Interestingly, these are the same benchmarks that show the most CIDI savings, so naturally they experience some mispredicted loads.

We conclude that load violations are likely to be infrequent in an actual implementation. Therefore, it may suffice to handle load violations like exceptions (flush pipeline and restart). Repeatedly mispredicted loads and their dependents can be included in CIDD recovery traces, for efficient selective recovery of the mispredicted loads.

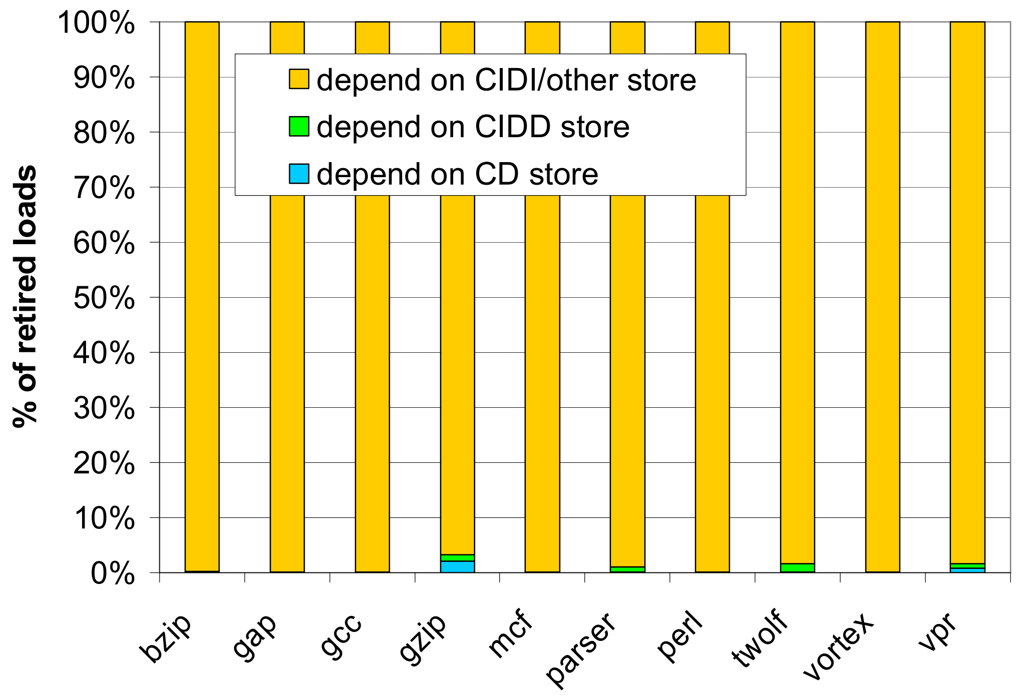


Figure 5-24: Memory violations.

5.5 Control Independence Example: Top Trace in Twolf

This section presents and discusses an example of control independence taken from twolf, one benchmark that shows a high percentage of CIDI instruction savings. The logical trace length is 64 and default trace selection is used. The recovery trace that yields the highest contribution to total CIDI savings was selected for study in depth. This recovery trace contributes 1.2% of the total CIDI savings. Among 64 CI instructions between the reconvergent point and the end of the trace, 62 instructions are CIDI, that is, the recovery trace contains only 2 CIDD instructions. This trace was used for recovery 12,562 times.

Figure 5-25 shows the C code from twolf that contains the mispredicted branch that uses the recovery trace. It is the branch corresponding to the `if-else` statement. This branch mispredicted 12,735 times. We noticed that the compiler transformed this `if-else` statement into an `if` statement, as shown in Figure 5-26. Thus, the CI region of the mispredicted branch is a subset of the `XPICK_INT` function, as can be seen in the figure. The value of `temp` is passed to the function through argument register `a2`, which means there are only register dependences between the CD and the CI regions, and no memory dependences. The function body of `XPICK_INT` is shown in Figure 5-27. The function contains another `if-else` branch. The `else` is always taken, and the branch predictor predicts the `else` path with more than 99.9% accuracy. As mentioned above, the trace has only 2 CIDD instructions. One of the instructions is a move of the value in `a2` to `s0`. The other instruction is the branch instruction that compares the value of `s0` with zero, which corresponds to the `if(c < 0)` statement. Most of the CIDI instruction savings come from the entrance code of the `XPICK_INT` function and subsequent instructions inside the `PICK_INT` function. It so happens that the CI region ends after the return from `PICK_INT` and just before the branch corresponding to the `while (d == c)` statement.

```
bblock = blk;
bycenter = bblckptr->bycenter;

if (bblock == ablock)
{
    bxcenter = XPICK_INT(l, r, axcenter) ;
}
else
{
    bxcenter = XPICK_INT(l, r, 0) ;
}
```

Figure 5-25: Control independence example

```
bblock = blk;
bycenter = bblckptr->bycenter;

int temp = 0;
if (bblock == ablock)
{
    temp = axcenter;
}

bxcenter = XPICK_INT (l, r, temp);
```

Figure 5-26: Transformed code.

```
XPICK_INT (int l, int u, int c)
{
    int d;

    if (c < 0)
    {
        return (-c);
    }
    else
    {
        do
        {
            d = PICK_INT(l, u);
        } while (d == c);

        return (d);
    }
}
```

Figure 5-27: CI region.

Chapter 6

Summary and Future Work

Conventional superscalar processors use full recovery to recover from branch mispredictions. While simple, full recovery needlessly re-executes many future CIDI instructions. On the other hand, selective recovery is possible, but is complicated by the fact that it requires sequencing through all CI instructions to single out CIDD instructions for re-execution.

In this thesis, I conceptualize the recovery process as constructing a “recovery sub-program” for repairing partially incorrect future state. Selective recovery constructs a reduced recovery sub-program that consists of only CD and CIDD instructions, and not CIDI instructions. Unfortunately, constructing the reduced recovery sub-program on-the-fly is complex. To compound the problem, the same recovery sub-program is repeatedly constructed, every time the corresponding branch is mispredicted.

I propose a trace-cache-based technique, where the CIDD component of the recovery sub-program for each branch is pre-constructed once and cached in a recovery trace cache for future use. When a misprediction is detected, first the branch's correct CD instructions are fetched from the instruction cache as usual and then its CIDD trace is fetched from the recovery trace cache. With the trace cache, fetching only CIDD instructions is as simple as fetching all CI instructions from a conventional instruction cache, as it does not require explicitly singling out CIDD instructions. Distilling CIDD instructions was done a priori, on the fill-side of the trace cache. Therefore, the recovery trace cache is efficient on multiple levels, combining the simplicity of full recovery with the performance of selective recovery.

This thesis explains the proposed trace-cache-based control independence architecture, at a high level. Preliminary studies are also presented, to project the potential of exploiting control independence and the viability of a trace-cache-based approach in particular.

The four sets of results presented in this thesis are summarized below.

(i) Retired dynamic instructions are broken down into different categories, based on their control and data dependences with respect to prior mispredicted branches. Breakdowns are first provided in the context of unbounded trace caches to understand the intrinsic behavior of benchmarks. The results indicate there is significant opportunity to exploit control independence in benchmarks with severe mispredictions. For a logical trace length of 64 instructions and default trace selection, the percentage of CIDI instructions (i.e., the percentage of dynamic instructions in the shadows of mispredictions that are not needlessly re-executed) ranges from 1% to 31%, and is 13% on average.

(ii) Unique recovery traces are characterized in terms of their individual contributions to total CIDI instruction savings. The results show that, for benchmarks with high percentages of CIDI savings (i.e., benchmarks for which exploiting control independence is worthwhile), only a moderate number of unique traces is required to achieve most of the CIDI savings potential.

(iii) Recovery trace cache hit ratios and retired instruction breakdowns are characterized for various recovery trace cache configurations. Among benchmarks with significant CIDI savings in the case of an unbounded trace cache, twolf has the lowest hit ratios, ranging from 32% (512 lines, direct-mapped) to 61% (2,048 lines, 8-way set-associative). For twolf, the 512-line direct-mapped trace cache and 2,048-line 8-way trace cache capture 37% and 69% of total CIDI savings, respectively.

(iv) Measurements of possible mispredicted CI loads, caused by correct CD stores that are fetched late or CIDD stores that re-execute, show that load violations are infrequent and unlikely to be a performance limiter in an actual implementation.

This thesis is a preliminary study of a trace-cache-based control independence architecture. In future work, a detailed and comprehensive microarchitecture must be designed, implemented in a cycle-level simulator, and evaluated for performance and other metrics. The results provided in this study are encouraging and justify pursuing a trace-cache-based control independence architecture.

At least two aspects of the microarchitecture are not elaborated in this thesis and are left for future work.

The first aspect deals with checkpoint placement. A recovery trace only provides selective recovery within the scope of the logical trace length. Thus, the fetch unit and processor state must be rolled back logically to the endpoint of the recovery trace. This requires pre-placing a checkpoint logically at the end of the trace (or logically before the end of the trace, the closer the better). Fortunately, the recovery preparation phase provides a convenient timeframe for pre-placement of checkpoints.

The second aspect deals with repairing mappings in checkpoints that are logically between the start and end of a recovery trace. New mappings in the repair rename map, that are live at a checkpoint's logical position in the dynamic instruction stream, must be merged into the checkpoint. I believe key information can be deduced during trace pre-construction, that greatly simplifies determining which mappings must be merged.

In addition to fleshing out the microarchitecture, performance optimizations will be explored in future work. For example, while the recovery trace cache presented in this thesis gives equal weight to all recovery traces, caching should favor highly compressed recovery traces, i.e., recovery traces with very few CIDD instructions. These recovery traces correspond to mispredicted branches for which there are many misprediction-independent instructions, in which case selective recovery is substantially distinct from full recovery. Even moderately compressed recovery traces should be favored, if they make significant individual contributions to total CIDI savings. It may be that a very small recovery trace cache can exploit most of the control independence opportunity, if only high-payoff recovery traces are cached. This hypothesis will be tested in future work.

Bibliography

[1] P. Ahuja, K. Skadron, M. Martonosi, and D. Clark. Multipath Execution: Opportunities and Limits. In *Proc. of the 12th Annual International Conference on Supercomputing*, pages 101-108, July 1998.

[2] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *Proc. of the 31st Annual International Symposium on Microarchitecture*, pages 226-236, Dec. 1998.

[3] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 423-434, Dec. 2003.

[4] A. S. Al-Zawawi, V. Reddy, E. Rotenberg. An Efficient Control Independence Architecture. Unpublished Report. In submission.

[5] J. O. Bondi, A. K. Nanda, and S. Dutta. Integrating a Misprediction Recovery Cache (MRC) into a Superscalar Pipeline. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 14–23, Dec. 1996.

[6] D. Burger, T. M. Austin, and S. Bennett. The Simplescalar Toolset, Version 2. Tech. Report CS-TR-1997-1342, CS Dept., Univ. of Wisconsin-Madison, July 1997.

[7] C. Cher and T.N. Vijaykumar. Skipper: A Microarchitecture for Exploiting Control-Flow Independence. In *Proceedings of the 34thd Annual International Symposium on Microarchitecture*, pages 4-15, Nov. 2001.

[8] Y. Chou, J. Fung, and J. Shen. Reducing Branch Misprediction Penalties via Dynamic Control Independence Detection. In *Proc. of the Annual International Conference on SuperComputing*, pages 109-118, June 1999.

[9] J. D. Collins, D. M. Tullsen, and H. Wang. Control Flow Optimizations via Dynamic Reconvergence Prediction. In *Proc. of the 37th Annual International Symposium on Microarchitecture,* pages 129-140, Dec. 2004.

[10] A. Cristal, D. Ortega, J. Llosa, M. Valero. Kilo-instruction Processors. In *Proc. of the 5th International Symposium on High Performance Computing*, pages 10-25, Oct. 2003.

[11] A. Cristal, M. Valero, J. Llosa, and A. González. Large virtual ROBs by processor checkpointing. Technical Report UPC-DAC-2002.

[12] A. Gandhi, H. Akkary, and S. Srinivasan. Reducing Branch Misprediction Penalty via Selective Recovery. In *Proc. of the 10th Annual International Symposium on High Performance Computer Architecture*, pages 254-265, Dec. 2004.

[13] T.H. Heil and J.E. Smith. Selective Dual Path Execution. Technical Report, University of Wisconsin-Madison, ECE, Nov. 1997.

[14] W. W. Hwu and Y. N. Patt. Checkpoint Repair for Out-Of-Order Execution Machines. In *Proc. of the 14th Annual International Symposium on Computer Architecture*, pages 18-26, July 1987.

[15] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning Confidence to Conditional Branch Predictions. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 142-152, Dec. 1996.

[16] D. A. Jim´enez and C. Lin. Dynamic Branch Prediction with Perceptrons. In *Proc. of the 7th Annual International Symposium on High Performance Computer Architecture*, pages 197-206, Jan. 2001.

[17] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic Hammock Predication for Nonpredicated Instruction Set Architectures. In *Proc. of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 278-285, Oct. 1998.

[18] A. Klauser, A. Paithankar and D. Grunwald. Selective Eager Execution on the PolyPath Architecture. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 250-259, June 1998.

[19] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, May 1992.

[20] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 59-70, May 2002.

[21] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu. Characterizing the Impact of Predicated Execution on Branch Prediction. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 217-227, Dec. 1994.

[22] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In

Proc. of the 22nd Annual International Symposium on Computer Architecture, pages 138–149, May 1995.

[23] P. Marcuello, A. González, J. Tubella. Speculative Multithreaded Processors. In *Proc. of the 12th Annual International Conference on Supercomputing*, pages 77-84, July 1998.

[24] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 3-14, Nov. 2002.

[25] S. McFarling, Combining Branch Predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[26] I. Park, B. Falsafi, and T. N. Vijaykumar. Implicitly – Multithreaded Processors. In *Proc. of the 30th Annual International Symposium on Computer Architecture,* pages 39-50, June 2003.

[27] D. N. Pnevmatikatos and G. S. Sohi. Guarded Execution and Branch Prediction in Dynamic ILP Processors. In *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 120-129, April 1994.

[28] E. Rotenberg, S. Bennett, and J. E. Smith. A Trace Cache Microarchitecture and Evaluation. *IEEE Transactions on Computers*, Special Issue on Cache Memory, 48(2): pages 111-120, Feb. 1999.

[29] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 24-34, Dec. 1996.

[30] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace Processors. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 138-148, Dec. 1997.

[31] E. Rotenberg, Q. Jacobson, and J. E. Smith. A Study of Control Independence in Superscalar Processors. In *Proc. of the5th International Symposium on High-Performance Computer Architecture,* pages 115-124. Jan. 1999.

[32] E. Rotenberg and J. E. Smith. Control Independence in Trace Processors. In *Proc. of the 32nd Annual International Symposium on Microarchitecture,* pages 4-15, Nov. 1999.

[33] J. E. Smith and G. S. Sohi. The Microarchitecture of Superscalar Processors. In *Proc. IEEE*, volume 83, pages 1609-1624, Dec. 1995.

[34] J. E. Smith and A. R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proc. of the 12th Annual International Symposium on Computer Architecture*, pages 36-44, June 1985.

[35] A. Sodani and G. Sohi. Dynamic Instruction Reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 194–205, June 1997.

[36] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 414-425. June 1995.

[37] E. Sprangle and Y. Patt. Facilitating Superscalar Processing via a Combined Static/Dynamic Register Renaming Scheme. *In Proc. of the 27th International Symposium on Microarchitecture*, pages 143-147, Dec. 1994.

[38] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, 107-119, Oct. 2004.

[39] S. Vajapeyam and T. Mitra. Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 1-12, June 1997.

[40] S. Wallace, B. Calder, and D. Tullsen. Threaded Multiple Path Execution. In *25th Annual International Symposium on Computer Architecture*, pages 238-249. June 1998.

[41] C.B. Zilles, J.S. Emer, G.S. Sohi. The Use of Multithreading for Exception Handling. In *Proc. of the 32nd Annual International Symposium on Microarchitecture*, pages 219-229, Nov. 1999.