ABSTRACT

GALLIHER, WILLIAM PATRICK. Squashed Branch Reuse. (Under the direction of Dr. Eric Rotenberg, committee chair).

In an out-of-order superscalar processor, when a mispredicted branch initiates recovery, all younger instructions are squashed. This includes squashing younger branches, some of which may have executed and been discovered as mispredicted branches themselves. In this case, the recovery initiated by the older branch can be turned into an opportunity for higher performance, by reusing the results of squashed branches as accurate predictions when they are refetched. Squashed Branch Reuse (SBR) exploits this opportunity by saving the results of younger branches at the time of a squash, and selectively reusing these results for more accurate branch prediction. Furthermore, SBR is implemented entirely within the fetch unit, allowing it to be used easily alongside any branch predictor. SBR then turns the cost of a squash into an opportunity for more accurate branch predictions. Looked at another way, with SBR and under the right circumstances of concurrent branches, the processor may pay only one misprediction penalty for multiple mispredictions by the branch predictor.

Squashed Branch Reuse

by
William P. Galliher

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, North Carolina

2015

APPROVED BY:

Dr. Eric Rotenberg Committee Chair	Dr. Huiyang Zhou
Dr. Greg Byrd	

DEDICATION

This thesis is dedicated to my parents, who faithfully encouraged me and supported me all throughout my educational career. Without their guidance, love, and support, I certainly would not have reached this point. Thank you.

BIOGRAPHY

William Patrick Galliher is an Accelerated Master's Program student in the Electrical and Computer Engineering Department at North Carolina State University. Prior to pursuing a Master's degree, William earned a B.S. in Computer Engineering and a B.S. in Electrical Engineering from North Carolina State University. During that time, he had an internship with Intel, and pursued research following up branch prediction methods with Dr. Eric Rotenberg. His research focuses on processor architecture, namely branch predictors. Upon completion of his Master's degree, William plans to continue his education through pursuing a PhD.

ACKNOWLEDGMENTS

I would like to thank my family, my two parents and three brothers. Their support, encouragement, and guidance has been immeasurable, but most of all, their continued love has kept me going through the hardest parts of my life.

I would like to thank my adviser, Dr. Eric Rotenberg, for his guidance, teaching, and continual encouragement. He has taught me the methods of research, and shaped my development in academia through the opportunities he has given me.

I would like to thank my advisory committee members, Dr. Huiyang Zhou and Dr. Gregory Byrd, for their time, effort, and service. Their willingness to volunteer those aspects leaves me immensely grateful.

I would also like to give my utmost gratitude to Elliott Forbes. His advice, from my first year in pursuing my B.S., has cleared the way for many of the opportunities in my academic career. From the first programming project during freshman summer, to the urging to take Dr. Rotenberg’s architecture class, and even to this day with small meetings to discuss research, defense, and each other’s lives, I do not think I could have asked for a better friend through these five years.

This research was supported by NSF grant CCF-0916481 and an ECE fellowship. Any opinions, findings, and conclusions or recommendations expressed herein are those of the author and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Despite impressive improvements in branch predictors over several decades, certain classes of data dependent branches have relatively high misprediction rates [1]. In an out-of-order superscalar processor, when a mispredicted branch initiates recovery, all younger instructions are squashed. This includes squashing younger branches, some of which may have executed and been discovered as mispredicted branches themselves. In this case, the recovery initiated by the older branch can be turned into an opportunity for higher performance, by reusing the results of squashed branches as accurate predictions when they are refetched. Squashed Branch Reuse (SBR) exploits this opportunity by saving the results of younger branches at the time of a squash, and selectively reusing these results for more accurate branch prediction. Furthermore, SBR is implemented entirely within the fetch unit, allowing it to be used easily alongside any branch predictor. SBR then turns the cost of a squash into an opportunity for more accurate branch predictions. Looked at another way, with SBR and under the right circumstances of concurrent branches, the processor may pay only one misprediction penalty for multiple mispredictions by the branch predictor.

SBR exploits an existing structure in the fetch unit that we will refer to as the Branch Queue. The Branch Queue can be thought of as a reorder buffer for branches. Its purpose is to non-speculatively update the history table(s) of the branch predictor and checkpoint/restore the global branch history (and possibly other information) for each in-flight branch.

The Branch Queue keeps track of all in-flight branches in program order. Branches are inserted into the Branch Queue in program order at fetch and removed from the Branch Queue in program order at retirement. Each in-flight branch keeps its Branch Queue index with it as it flows down the pipeline, so that when it executes in the backend, it can update its Branch Queue entry with results from execution: its computed direction and whether or not it was mispredicted. Figure *1.1* shows an example snapshot of the Branch Queue during execution. The colored (red or green) branches have executed and the non-colored branches have not executed yet. Among the executed branches, the red ones are mispredicted and the green ones are correctly-predicted. The mispredicted branch at the head of the Branch Queue will initiate a pipeline squash, including rolling back the Branch Queue tail to the head and thereby discarding all branches after the head branch.

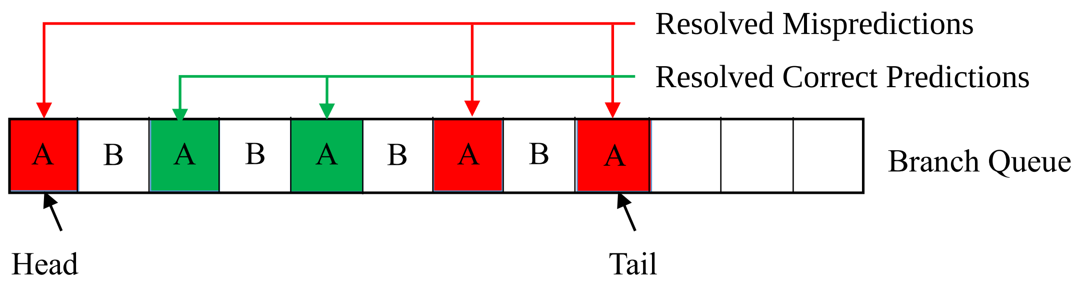

Figure 1.1 Branch Queue example

Figure *1.2* illustrates how SBR can reduce mispredictions. In Figure 1.2(a), which shows the Branch Queue before the squash, all dynamic iterations of A have executed. The green ones were correctly predicted and the red ones were mispredicted. The dynamic iteration of *A* at the head of the Branch Queue is triggering a squash. Without SBR (Figure 1.2(b)), two different misprediction scenarios are left after the squash. First, the two younger mispredicted branches are mispredicted again. Second, the youngest dynamic iteration of A that was correctly predicted before is now mispredicted, because of the change in global and/or local branch history (assuming a two-level branch predictor) caused by the head branch. SBR solves both of these cases by reusing all dynamic iterations of A (Figure 1.2(c)). Reuse is made possible by results from execution available in the Branch Queue.

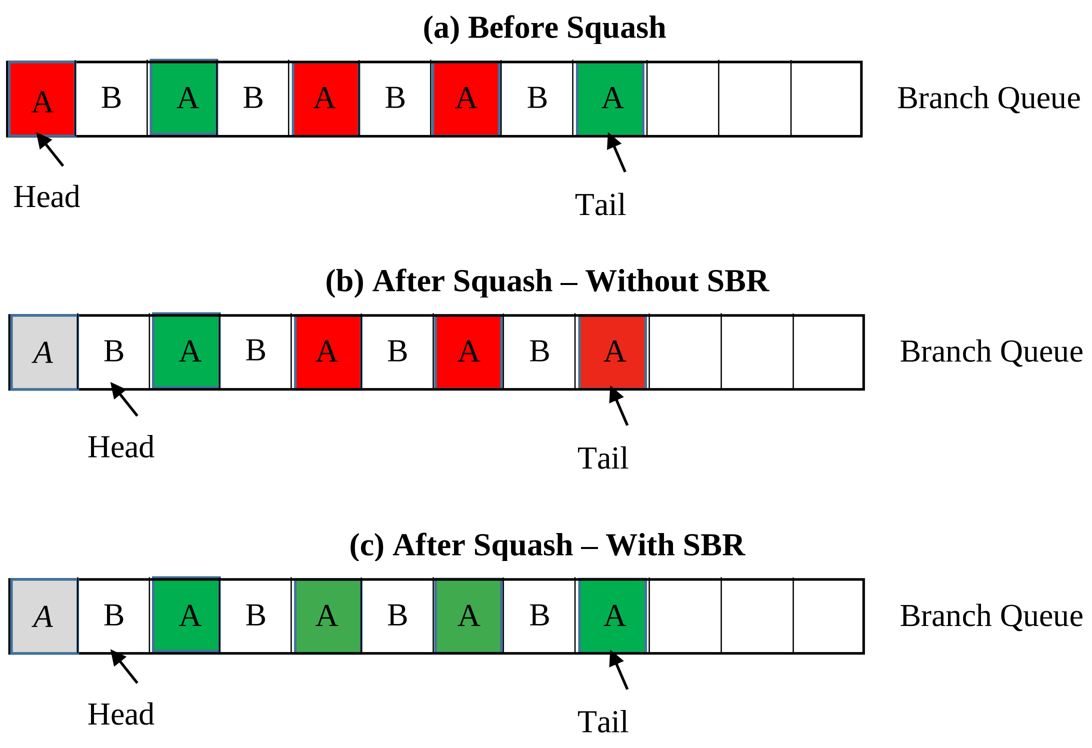

Figure 1.2 Misprediction scenarios without and with SBR.

SBR is most profitable for branches that have the following three qualities:

1. *Control independent of prior branches.* A branch is control independent when it is present in every iteration of the loop containing it. This allows SBR to easily line-up refetched branches with their previously squashed results. The branch is not considered control independent if its execution is dependent on another branch inside the loop, such as a nested branch.

2. *Data independent of prior branches.* In this case, the control-independent branch's outcome is unaffected by prior branches' directions. This means its squashed result is correct – in spite of prior mispredicted branches – and reuse of this result is clearly profitable.

3. *Many dynamic iterations of the branch fit within the processor window.* The branch exists in a tight loop. This increases the likelihood of many dynamic iterations being available for reuse inside the Branch Queue at the time of a squash.

The branch "m >= WSIZE" shown in Figure *1.3*, line 547, from the benchmark gzip, is a prime example of a control and data independent branch within a tight loop. First, the branch is executed in every iteration of the loop (control independent). Second, the branch's direction depends on head[n] and influences the value subsequently stored to head[n], where n is for the current loop iteration. Thus, future dynamic iterations of the branch are unaffected by its previous dynamic iterations (data independent). Third, the loop body is small, allowing for many dynamic iterations of the branch to reside in the Branch Queue.

```
545:    for (n = 0; n < HASH_SIZE; n++) {
546:        m = head[n];
547:        head[n] = (Pos)(m >= WSIZE ? m-WSIZE : NIL);
548:    }
```

Figure 1.3 gzip target branch source sample

Rather than implement complex hardware to explicitly detect control and data independent branches, SBR uses a confidence mechanism to learn which branches are reusable for nearly all of its dynamic iterations. The confidence mechanism implicitly selects branches that conform to the control independence and data independence characteristics without explicitly checking for these characteristics. In addition, SBR automatically and accurately aligns refetched branches with their previously squashed results using a technique called "dynamic iteration analysis". Finally, SBR interacts only with the existing Branch

Queue. Thus, integration of SBR into a superscalar processor is simple, in that it is localized within the fetch unit and can be included alongside any branch predictor.

In this thesis, we will discuss the methodology and targeted branches that SBR is designed for (Chapter 2). After this, we will outline the SBR implementation (Chapter 3), and the results from running SBR and our baseline on multiple benchmarks (Chapter 5). Finally, we will conclude with comparisons to related work (Chapter 6) and discuss possible future work in SBR (Chapter 7).

Chapter 2

Methodology and Target Branches

We aim to reduce mispredictions of known hard-to-predict reusable branches, through automatic detection and reuse of these branches. To test this, we analyze benchmarks that have branches which show the characteristics for good reuse, and determine if SBR automatically detects and improves prediction accuracy of these branches. In addition, we run other benchmarks to assure that SBR does not harm the performance of benchmarks where this particular class of hard-to-predict branches is not the focus. Our goal is to discover if SBR can improve known hard-to-predict reusable branches while, at a minimum, not harming other benchmarks, and finding other branches valid for reuse automatically. These other benchmarks come from the SPEC2000 and SPEC 2006 benchmark suites [2] [3]. The known hard-to-predict branches showing good characteristics for reuse come from astar, gzip, and soplex. The details of these branches are outlined here, and how they will assist in analyzing SBR's performance.

2.1 Methodology

Two benchmark suites are used, CPU2000 and CPU2006. For all benchmarks, the same configuration was used. The simulator modeled an Intel Sandy Bridge architecture with a gshare branch predictor using a 16KB entry table. The details of the configuration can be found in Evaluation Environment (Chapter 4).

The benchmarks were run without SBR enabled, to find the baseline misprediction rates. These rates can be seen in Figure *2.1*.

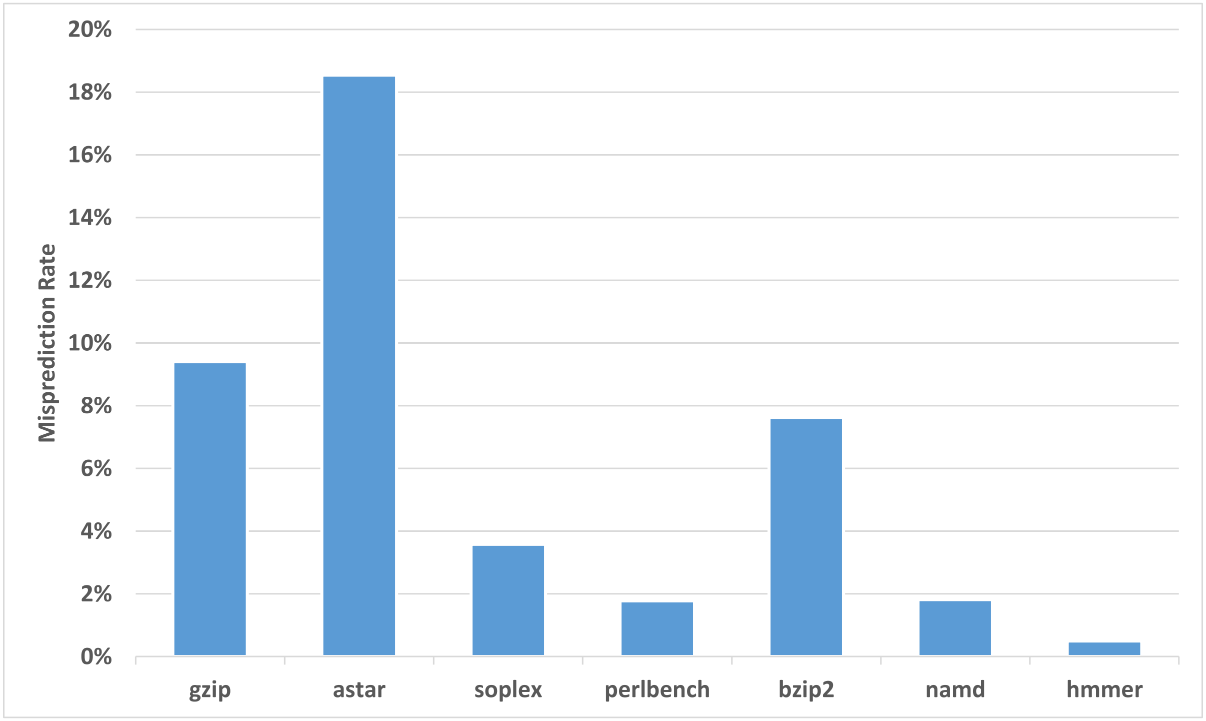

Figure 2.1 Base benchmark misprediction rates

2.2 Target Branches

Three benchmarks contain target, representative hard-to-predict branches. These three benchmarks are soplex, gzip, and astar. Each contain different situations that test different parts of SBR, from amount of entries, ideal case that should be captured, and abnormal exit cases for the loop.

2.2.1 soplex

The first one we will examine is soplex. Soplex contains one known hard-to-predict branch that resides inside an outer for loop. This branch is an ideal case for SBR, as it is independent and data dependent, while having no abnormal exit cases. Abnormal exit cases would be a situation that causes the thread to leave the loop early, such as break statements. In this regard, soplex creates the situation where SBR should work well, with considerable improvement. However, because of the amount of instructions inside the loop, we would expect to see fewer mispredictions removed by SBR from this as opposed to the gzip case, which focuses on a tight, small loop with many iterations.

As shown in Figure 2.2, the target branch relies on *theeps* and *x*. *Theeps* is set before the loop, and not modified inside the loop. The other variable *x* is loaded from fTest[i], similar to gzip, so each dynamic iteration of the branch is not affected by other dynamic iterations. These branch, located at program counter 12005e1c8, is a prime candidate for SBR.

soplex source code and assembly code (spxsteeppr.cpp)

```
291:   for (int i = thesolver->dim() - 1 - start; i >= 0; i -= incr)
292:   {
293:      x = fTest[i];
294:
295:      if (x < -theeps)
296:      {
297:         /**@todo this was an assert! is an assertion correct?*/
298:         // assert(coPenalty_ptr[i] >= theeps);
299:         #ifndef NDEBUG
300:         if( coPenalty_ptr[i] < theeps )
301:         {
```

```
/450.soplex/spxsteeppr.cc:295
   12005e1c4:      4621003c     c.lt.d      $f0,$f1
   12005e1c8:      4500000c     bc1f  12005e1fc
<_ZN6soplex10SPxSteepPR11selectLeaveEv+0xc4>
   12005e1cc:      6484fff8     daddiu      a0,a0,-8
```

Figure 2.2 soplex target branches

2.2.2 gzip

Gzip contains target branches nested within a tight loop. This stresses the size of the structs that SBR relies on, as many dynamic results will need to be reused from the Branch Queue. This also stresses the ability of SBR to correctly retrieve the result that corresponds to a specific dynamic iteration of the branch. If the SBR does not retrieve the branch that corresponds to the current dynamic iteration in the fetch unit, then the prediction is not guaranteed to be correct.

gzip source code and assembly code (deflate.c)

```
545:    for (n = 0; n < HASH_SIZE; n++) {
546:        m = head[n];
547:        head[n] = (Pos)(m >= WSIZE ? m-WSIZE : NIL);
548:    }
549:    for (n = 0; n < WSIZE; n++) {
550:        m = prev[n];
551:        prev[n] = (Pos)(m >= WSIZE ? m-WSIZE : NIL);
552:        /* If n is not on any hash chain, prev[n] is garbage but
553:         * its value will never be used.
554:         */
555:    }
```

```
/164.gzip/deflate.c:547
   12000465c:      24658000     addiu a1,v1,-32768
   120004660:      0066182b     sltu  v1,v1,a2
   120004664:      14600002     bnez  v1,120004670 <fill_window+0x190>
   120004668:      0000202d     move  a0,zero
   12000466c:      00a0202d     move  a0,a1
   120004670:      a4440000     sh    a0,0(v0)
   120004674:      64420002     daddiu      v0,v0,2
...
/164.gzip/deflate.c:551
   120004694:      24658000     addiu a1,v1,-32768
   120004698:      0066182b     sltu  v1,v1,a2
   12000469c:      14600002     bnez  v1,1200046a8 <fill_window+0x1c8>
   1200046a0:      0000202d     move  a0,zero
   1200046a4:      00a0202d     move  a0,a1
   1200046a8:      a4440000     sh    a0,0(v0)
   1200046ac:      64420002     daddiu      v0,v0,2
```

Figure 2.3 gzip target branches

The target branches for gzip, both with source and the corresponding assembly, can be seen in *Figure 2.3*. The branches are the tests within the conditional statements on lines 547 and 551. They exhibit similar qualities to soplex, that m relies on an array load, and that the work inside the conditional statement does not modify data that is used in another dynamic iteration of the branch. However, the raw amount of work that is done in the loop is considerably less than in soplex, stressing SBR to correctly match the dynamic iterations during replay, and store all of the results.

2.2.3 astar

Astar exhibits hard-to-predict branches with good reuse characteristics, but it presents another corner case that stresses SBR's ability to match branch results that are stored with the dynamic iteration of the branch currently being predicted in the fetch unit. Each of the targeted, hard-to-predict branches have dependent slices that contain return statements. These return statements create a chance that the branches being retrieved in the fetch unit do not rely on the same data that the branches stored in the SBR relied on. Because of this, the dynamic iterations of the branches can become out of sync. However, since SBR seeks to be completely automated in hardware, requiring no assistance from software, while remaining simple, it will need to be able to capture improvements from these branches as well. One

astar source code and assembly code (Way_.cpp)

```
57:  for (i=0; i<bound1l; i++)
58:  {
59:    index=bound1p[i];
60:
61:    index1=index-yoffset-1;
62:    if (waymap[index1].fillnum!=fillnum)
63:      if (maparp[index1]==0)
64:      {
...
```

```
/473.astar/Way_.cpp:62
   12000dfa8:      000b68b8     dsll  t1,a7,0x2
   12000dfac:      006d682d     daddu t1,v1,t1
   12000dfb0:      95b90000     lhu   t9,0(t1)
   12000dfb4:      13260010     beq   t9,a2,12000dff8
<_ZN6wayobj10makebound2EPiiS0_+0x90>
   12000dfb8:      0120702d     move  t2,a5
...
```

Figure 2.4 astar target branches

sample of the eight branches in astar is shown in Figure 2.4. All of the eight branches share similar structure.

The general benchmarks and the targeted branches will serve as a baseline to test the overall performance of SBR. The known targeted branches will allow close examination of how well SBR is able to capture branches that show good reuse characteristics, while the general benchmarks will measure SBR’s impact when the presence of targeted benchmarks is not known.

Chapter 3

Squashed Branch Reuse

Squashed Branch Reuse (SBR) is a novel technique that attempts to target hard-to-predict branches by reusing results that would be squashed in the shadow of a previous misprediction. To do this, SBR uses a Reuse Queue to store the results saved from a squash of the Branch Queue, using an algorithm to set iteration counters to be used during replay. This is one part of SBR. The second part of SBR is replaying the results from the Reuse Queue, which involves staying in sync with the predictions currently taking place in the fetch unit, and deciding if the result from the Reuse Queue should be used or not.

3.1 Filling the Reuse Queue

When the branch at the head of the Branch Queue is a misprediction, a squash is normally initiated which will destroy all of the contents of the Branch Queue. In SBR, however, a Reuse Queue is used to store the contents of the Branch Queue from being squashed. All branches are saved in this manner, whether or not they have been executed. This is to simplify staying in sync with the fetch unit during replay. The full iteration makes determining the iteration simpler. If only the executed results are saved, it results in gaps within the iteration of a loop. This is covered more extensively in section 3.2, where replay is covered..

At the moment of a squash, the Branch Queue's head points to a mispredicted branch that is being resolved. This is shown as the box in red in Figure *3.1*. In between the head and the tail of the Branch Queue are branches that are in different states. Some might be executed and await retirement, computed either as taken or not taken, while others might not yet be executed. As the squash process begins, three actions occur in sequential order.

1. The Reuse Head is decremented by one to prepare for the copy of the branch.
2. The branch is copied from the Branch Queue tail to the Reuse Head of the Reuse Queue.
3. The iteration counter of the branch at the Reuse Head is set.

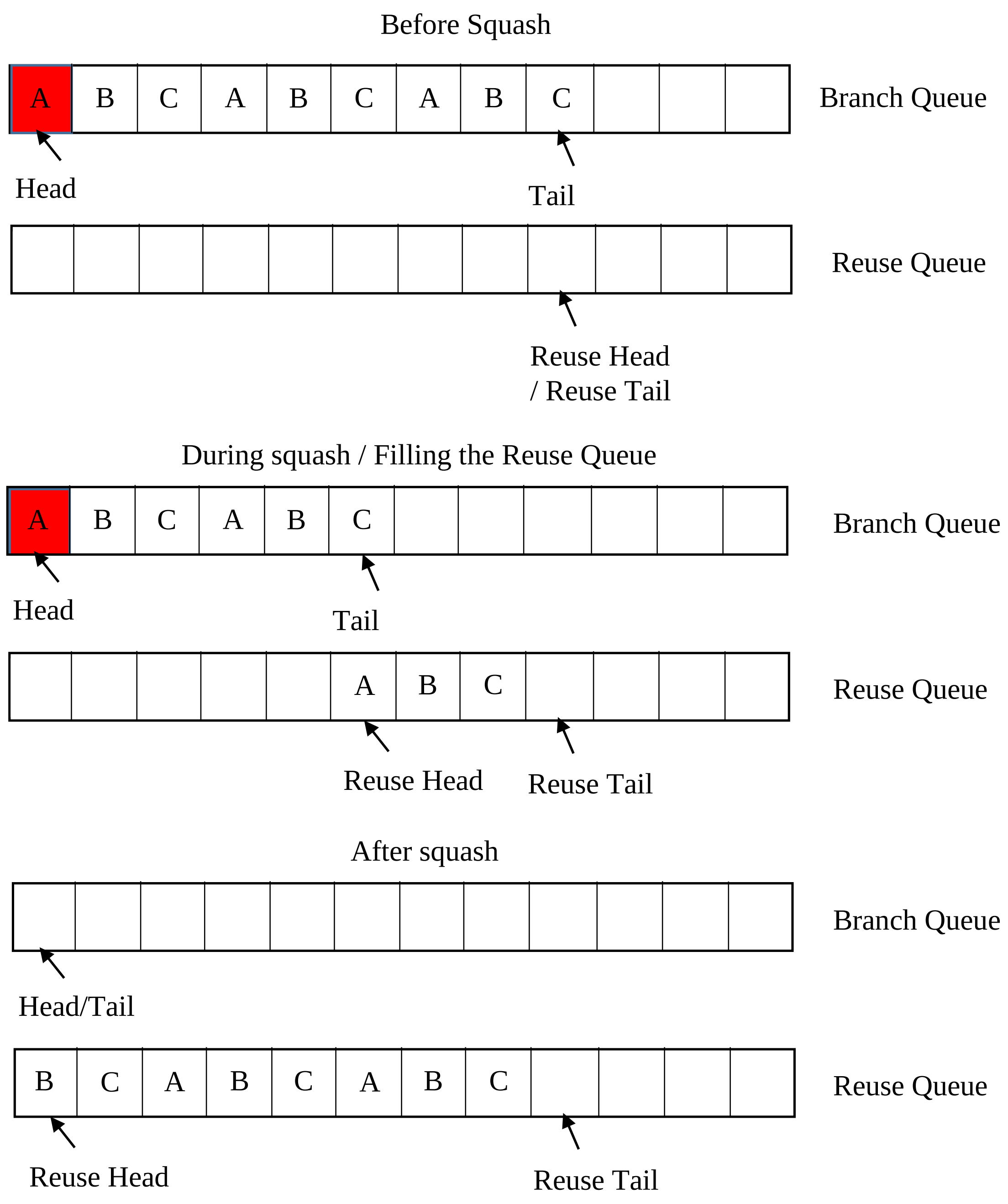

Figure 3.1 Squash overview

First, to prepare for the copy, the Reuse Head is decremented by one. The Reuse Queue acts as a circular buffer, and the entry at the tail is the oldest entry. If the head wraps around to the point that it becomes equal with the tail, then the tail is decremented with the head, essentially destroying the oldest value in the queue to make room for the new value being copied from the Branch Queue. In this way, the Reuse Queue avoids deadlock due to it being full, and it will always maintain that the value at the head is the newest value from the Branch Queue.

The branch at the tail of the Branch Queue is then copied to the Reuse Head in the Reuse Queue. This method of backfilling is to preserve any current contents that already exist in the Reuse Queue, as older results from a squash with more entries might still be present between the Reuse Head and the Reuse Tail.

Lastly, the iteration counter of the branch at the Reuse Head is set. The iteration counter will later be used during replay to determine the dynamic iteration of the branch in the fetch unit. This process is shown visually in Figure *3.2*. To decide the iteration counter for the branch, first, the previous iteration counter is selected, that is, the iteration counter at the entry right after the Reuse Head. Then, all entries in the Reuse Queue that have that same iteration counter are searched for the branch that was copied to the Reuse Head. If there is not a match, the branch that was copied is unique to this iteration, so its iteration counter is set to the same as the previous iteration counter. If there is a match, then this branch already exists in the current iteration. The branch at the Reuse Head then has its iteration counter set to the previous iteration counter decremented by one.

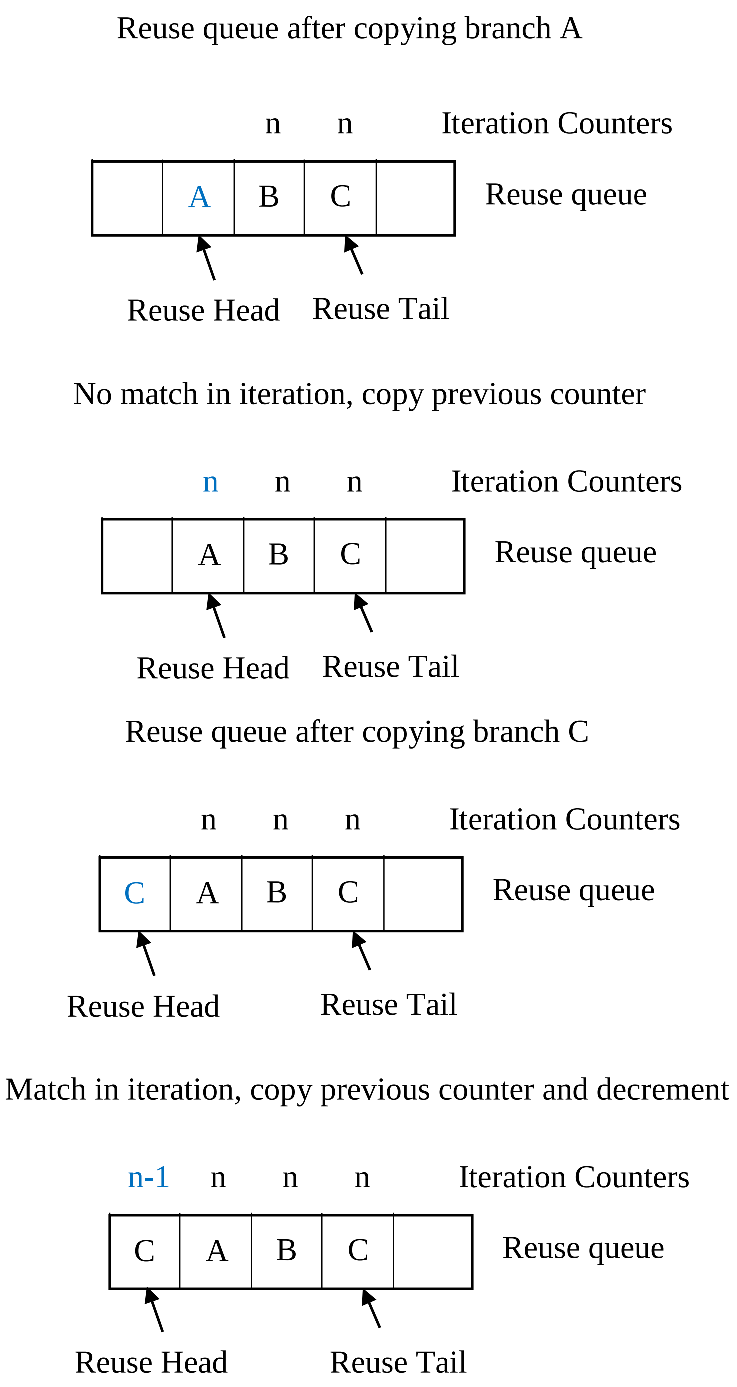

Figure 3.2 Setting iteration counters

This three step process is continued until the branch at the head of the Branch Queue is reached. Since that branch is the misprediction being resolved, it will not be copied over to the Reuse Queue, as another prediction will not be made on it. At this point, filling the Reuse Queue stops, and the rest of the squash for the processor continues as normal. The Reuse Queue will now hold the contents of the Branch Queue that would have been squashed, supplemented with iteration counters.

3.2 Replaying from the Reuse Queue

When the fetch unit fetches a branch, the PC of the branch is sent to the branch predictor for a prediction. At that time, the PC is also sent to the SBR unit, which uses the Reuse Queue to determine if a result from the Reuse Queue exists, and if it does, if the result should be used instead of a prediction from the branch predictor. If the SBR unit decides to use the result from the Reuse Queue, the result is returned to the fetch unit instead of a prediction from the branch predictor. However, the result from the SBR unit is treated just like a prediction. It has an entry in the Branch Queue, and goes through the same steps throughout the pipeline that a prediction does, including being verified for a misprediction and being squashed. All interactions with the SBR unit happen parallel to interactions with the branch predictor. A result from the SBR unit is checked at the same time a prediction from the branch predictor is checked, and fixes for mispredictions, including updating any confidence mechanisms, happen at the same time a branch predictor would be updated for a misprediction. In this way, the SBR unit stays integrated alongside the branch predictor and Branch Queue, living in the fetch unit. It requires no other changes to the rest of the pipeline.

Replaying a result from the Reuse Queue is then equivalent to making a prediction in the branch predictor. It follows certain steps to maintain the Reuse Head and return a result. These steps are as follows:

1. Upon receiving a PC from the fetch unit, perform dynamic iteration analysis
2. Check the Reuse Head for a match to the PC.
3. Determine if the result should be used or not.

First, we will define dynamic iteration analysis, as it contains the majority of the complexity in the SBR algorithms. After that, we will detail the implementation of returning the value, as well as determining whether to actually use the value or not.

3.2.1 Dynamic Iteration Analysis

Dynamic iteration analysis serves as the first step for accessing the Reuse Queue. The purpose of dynamic iteration analysis is to keep the Reuse Queue in sync with the dynamic iteration of the branches being predicted in the fetch unit. To do this, dynamic iteration analysis chooses to move the Reuse Head based on a number of different situations that can occur during reuse from the Reuse Queue. In general, there are two possible situations upon querying for a match at the Reuse Head.

1. A hit of the requested reuse PC at the Reuse Head. Reuse Head is in sync with the fetch unit. No changes need to be made.

2. A miss of the requested reuse PC at the Reuse Head. Reuse Head is possibly out of sync, and further analysis is required to determine if the case is *insertion* or *removal*.

On a hit, dynamic iteration analysis is complete, and SBR continues on to returning the value and determining how to use it. However, a miss can mean that the Reuse Head is *possibly* out of sync.

The first step upon a miss at the Reuse Head is to scan all entries that match the iteration counter at the Reuse Head for a match on the requested PC. This is to try and implicitly determine if the branch being requested is an independent branch, one of the primary attributes of the hard-to-predict branches SBR targets. If there is no match for the requested PC in the current iteration, then SBR determines that the requested branch is most likely not control independent. This situation is called an *insertion*, where the branch being requested by the fetch unit is a dependent branch inserted in to the current dynamic iteration. It will leave the Reuse Head where it is, and let SBR continue on to return a miss.

If there is a hit in the current iteration, however, we are in a situation called the *removal* case. SBR assumes that the PC being requested is a control independent branch. Because of this, the branch currently at the Reuse Head is considered a control dependent branch, as it was not requested by the fetch unit before a different, matching PC was requested. Therefore, the branch at the Reuse Head has been removed from the dynamic iteration. The next step is to determine if the currently requested PC is part of the current dynamic iteration or the next dynamic iteration. Where to set the Reuse Head then relies on which of these two we are in. The two situations can be determined based on if the hit in the current dynamic iteration is before the Reuse Head or after.

If the hit is before the Reuse Head, the currently requested PC is a PC that has already happened earlier in the current dynamic iteration, as shown in Figure *3.3*. The Reuse Head then needs to be moved forward until it encounters an iteration counter higher than the counter at the Reuse Head, signifying the next dynamic iteration. Once the Reuse Head has been moved forward to the next dynamic iteration, the new iteration is scanned for the requested PC again. If there is a match, the Reuse Head is set to that entry, and SBR continues on towards returning the value and determining whether to use it or not.

If the match in the current iteration is after the Reuse Head, then the fetch unit is still within the current dynamic iteration, as shown in Figure *3.4*. All the branches between the Reuse Head and the match are considered dependent branches that were removed, and the SBR needs to leap forward to the requested PC to be able to stay in sync. The Reuse Head is then moved forward within the current iteration to the match, and SBR continues on to returning the information and determining whether or not to use them.

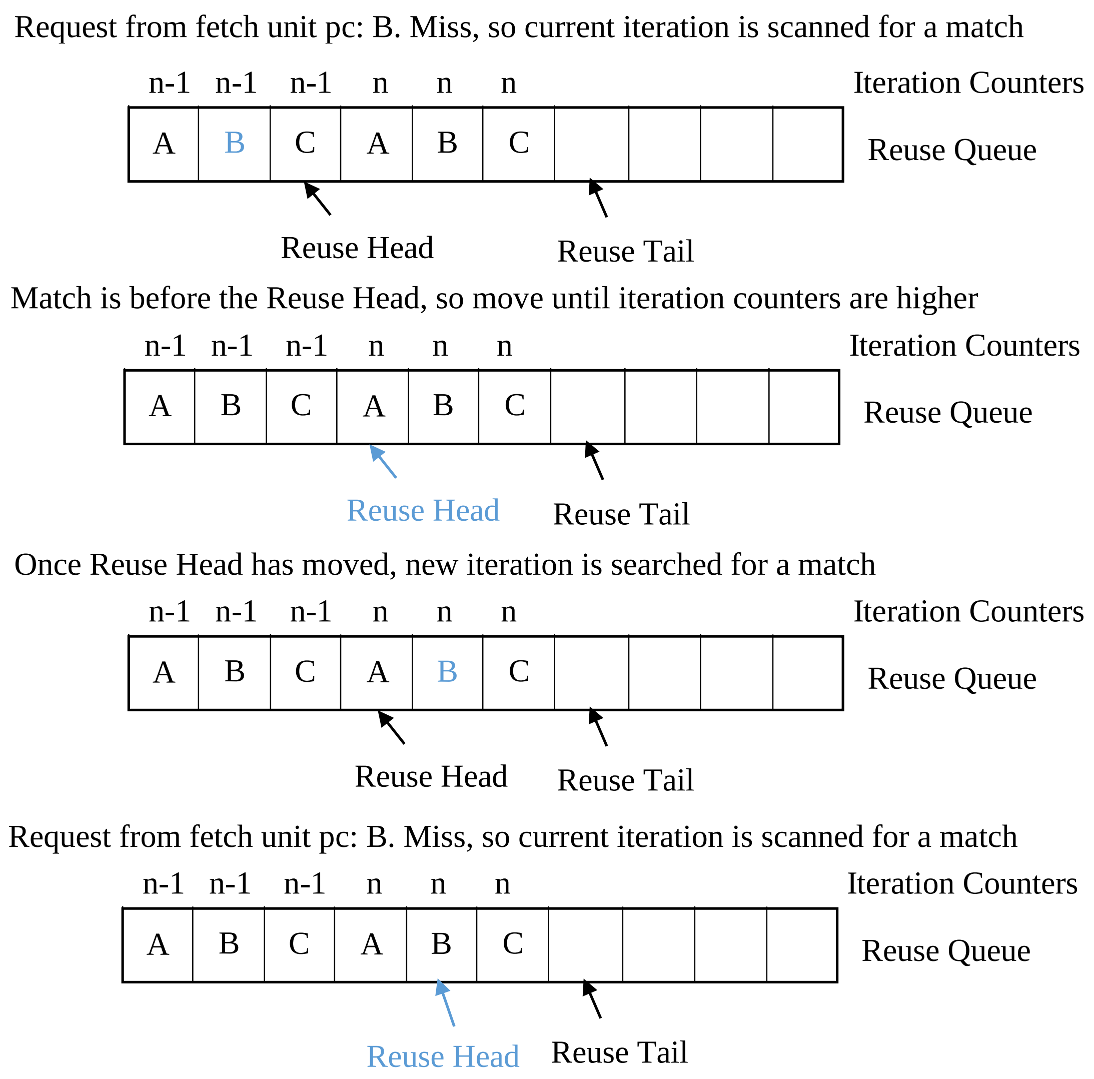

Figure 3.3 Removal for match before Reuse Head

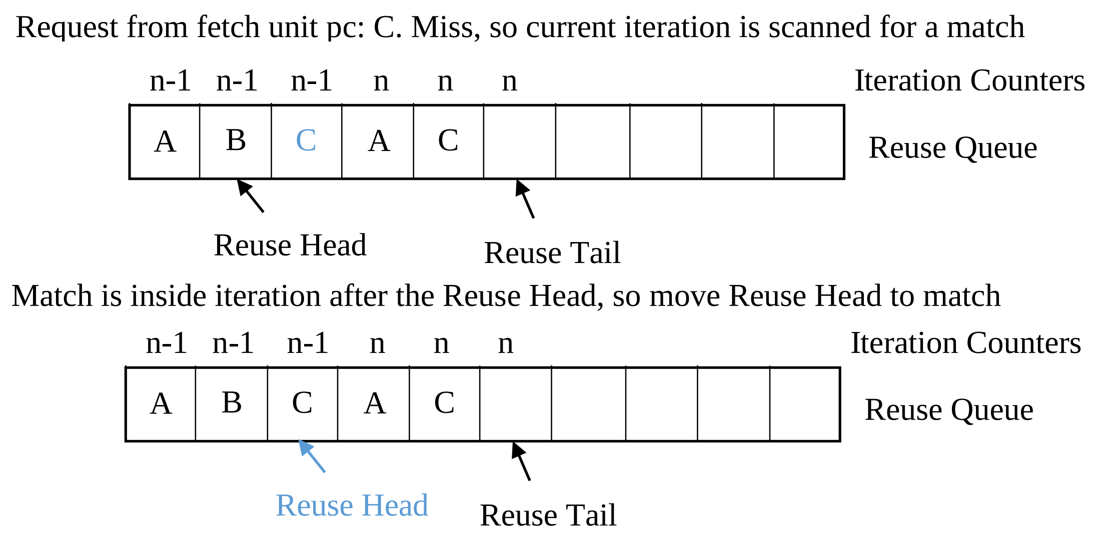
Request from fetch unit pc: C. Miss, so current iteration is scanned for a match
n-1 n-1 n-1 n n n
Iteration Counters
A B C A C
Reuse Queue
Reuse Head
Reuse Tail
Match is inside iteration after the Reuse Head, so move Reuse Head to match
n-1 n-1 n-1 n n n
Iteration Counters
A B C A C
Reuse Queue
Reuse Head
Reuse Tail

Figure 3.4 Removal for match after Reuse Head

3.2.2 Returning Results and Determining Confidence

After dynamic iteration analysis has assured that the Reuse Head is where the most likely possible match is, then SBR checks for a hit or miss of the requested PC from the fetch unit at the Reuse Head. If there is a miss after dynamic iteration analysis, then it is assumed all the work has been done to set the Reuse Head where it should be. The Reuse Head is not moved, and a miss is returned, instead of any information. The branch predictor's prediction is then used instead of any result from SBR.

In the case of a hit, the information at the Reuse Head is retrieved, and the Reuse Head is incremented by one, as shown in Figure *3.5*.

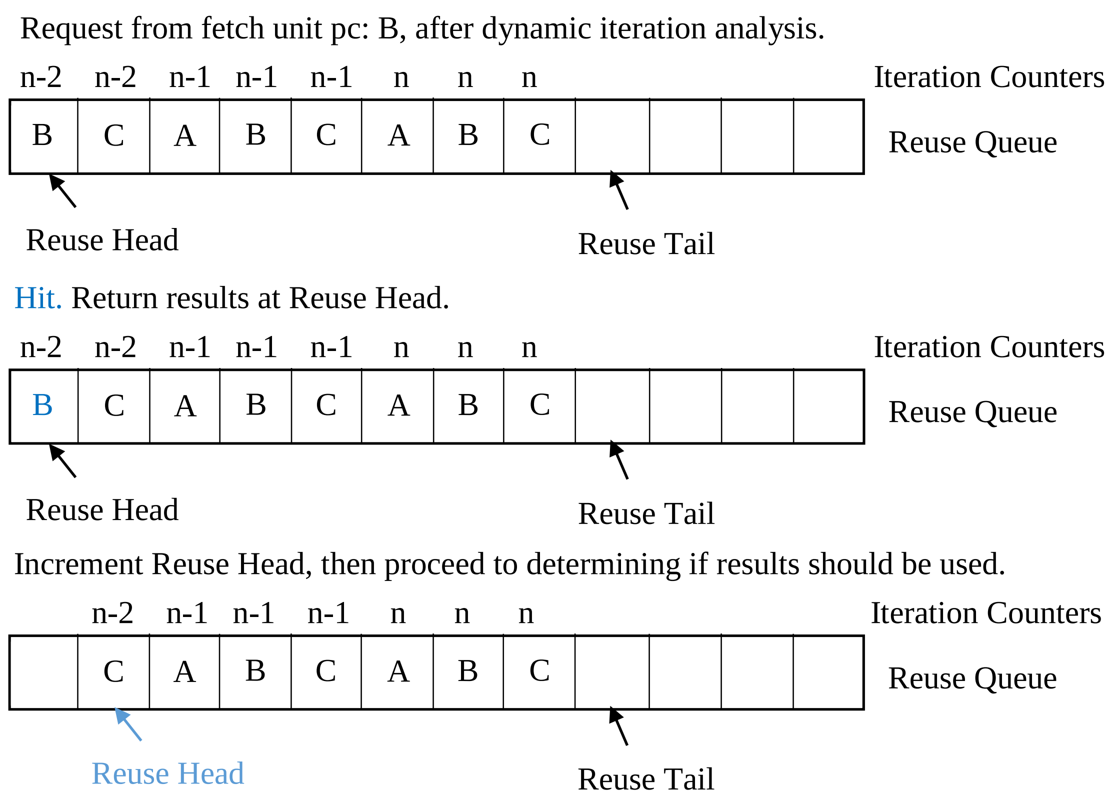
Request from fetch unit pc: B, after dynamic iteration analysis.
n-2 n-2 n-1 n-1 n-1 n n n
Iteration Counters
B C A B C A B C
Reuse Queue
Reuse Head
Reuse Tail
Hit. Return results at Reuse Head.
n-2 n-2 n-1 n-1 n-1 n n n
Iteration Counters
B C A B C A B C
Reuse Queue
Reuse Head
Reuse Tail
Increment Reuse Head, then proceed to determining if results should be used.
n-2 n-1 n-1 n-1 n n n
Iteration Counters
C A B C A B C
Reuse Queue
Reuse Head
Reuse Tail

Figure 3.5 Reuse hit case

At this point, while SBR has a hit, and the actions from dynamic iteration analysis could imply that the branch is most likely control independent, SBR still is not sure whether or not the branch is data independent. To determine whether or not the branch is of this type, SBR tests for the result changing between predictions. To assure this behavior, SBR uses a simple confidence mechanism with a resetting counter to determine if a branch should be used or not, as examined by Dr. Jacobsen in *Assigning Confidence to Conditional Branch Predictions* [4].

The confidence mechanism has two main parts.

1. Training the confidence mechanism
2. Querying the confidence mechanism to see if a particular counter is saturated

If the result returned from the Reuse Queue was executed, then it is used to train the confidence mechanism. The entry in the Branch Queue is marked with the result from the Reuse Queue. The entry in the Branch Queue is marked only if the branch from the Reuse Queue was previously executed, that is, it has a result for potential reuse. Whether or not the confidence mechanism says that that the result from the Reuse Queue is confident does not affect marking the Branch Queue with the result. When the branch is resolved, then the result in the Branch Queue is checked. If the result from the Reuse Queue matched the executed result, then the confidence counter for that branch is incremented. If the result from the Reuse Queue was not correct, then the confidence counter for that branch is reset down to zero. This assures that only branches that are highly accurate from the SBR are used.

If the result from the Reuse Queue is executed and SBR has marked the Branch Queue to train the confidence mechanism, then SBR queries the confidence mechanism to see if the counter for that branch is saturated. If the confidence mechanism returns a saturated counter for that branch, then the result from the Reuse Queue overrides the branch predictor, writing the result in Branch Queue. Finally, the reused result is returned to the fetch unit instead of the predictor's prediction.

Chapter 4

Evaluation Environment

To simulate the architecture for SBR, a cycle level MIPS-64 based simulator is used to model the processor behavior and execute the benchmarks chosen in chapter 2 [5].

The correctness of the architecture is confirmed by matching the results of every retired instruction back to a functional simulator that runs alongside the cycle level simulator. Therefore, if the cycle level simulator retires the same results as the functional simulator, the results from the cycle level simulator are taken as correct.

The sizing of the relevant structures for the architecture is chosen to be similar to an Intel Sandy Bridge processor [6]. For the relevant SBR structures, the two variables for sizing come in the size of the Reuse Queue and the amount of bits used in the confidence mechanism. No bits results in a result from SBR being used with no filtering, while more reset bits trims out less consistent branches at the cost of training time. For our testing purposes, we chose five bits. The sizing of the Reuse Queue should normally be at least the size of the Branch Queue, as a misprediction where the Branch Queue is full could fill the Reuse Queue completely.

This particular configuration serves to test SBR under a smaller window processor setting, instead of pursuing larger windows beyond what processors similar to Sandy Bridge would implement. An increase in window size would cause more opportunities for results on the SBR and speedup, therefore, this limitation is used.

All of the settings for the sizing of the different architecture structures are contained in Table *4.1*.

Table 4.1 Simulator core configuration

Branch Prediction	Branch Predictor: gshare predictor -Number of entries in gshare table: 64k (16KB) BTB - Sets: 1024 - Ways: 4 RAS entries: 16
Memory Hierarchy	Block size: 64B Victim Cache: 1 set 16 ways L1: 128KB, 4 way set associative, 1 cycle access latency L2: 512KB, 8 ways, 20 cycle access latency L3: 4096KB, 16 ways, 40 cycle access latency Memory: 200 cycle access latency
Fetch/Issue/Retire Width	4 Instructions/Cycle, interleaved
ROB/IQ/LDQ/STQ	Entries: 168/48/64/36, respectively
Fetch to Dispatch Latency	6 cycles
Physical RF	192
Checkpoints	64
SBR	Reuse Queue entries: 168 Confidence Mechanism: Reset counter, 5 bits per entry.

Chapter 5

Results and Analysis

Six benchmarks from SPEC2006 [3] and one benchmark from SPEC2000 [2] were run using the simulator configuration in Table *4.1*. Three of these benchmarks, gzip, astar, and soplex, contain targeted branches that SBR should perform well in, while the other five benchmarks from SPEC2006 were chosen to assure that SBR would have minimal impact on benchmarks that are not known to have branches that would result in a high benefit from SBR.

For the three targeted benchmarks, the first region where the targeted branches were identified. From that point, the simulator was run for one billion retired instructions. To measure against the baseline, the simulator was run with the same configuration with SBR disabled and SBR enabled.

For the five benchmarks outside of the targeted benchmarks, one billion retired instructions were executed, however, the first billion instructions were skipped to avoid data warmup and other preparatory work. These benchmarks were also run first with SBR disabled, then with SBR enabled.

Statistics for all runs were collected using the simulator itself, verifying all instructions retired and results against a functional simulator running the same benchmark.

For measuring the effectiveness of SBR, three statistics were measured. First, the misprediction rate for ever test was measured, using *mispredictions / total predictions*. Second, the overall IPC of the benchmark is measured with and without SBR. This allows us to calculate speedup, using *IPC with SBR / IPC without SBR*. Lastly, we give the prediction breakdown, measuring the predictions and mispredictions of both the branch predictor and the SBR when SBR is active. For targeted benchmarks, we include two other statistics. First, we measure the mispredictions of targeted branches with and without SBR. Second, the amount of mispredictions eliminated by SBR apart from the targeted branches is measured, to determine how well SBR is able to automatically reuse branches that are known to show good reuse characteristics.

5.1 Targeted Benchmark Performance

The targeted benchmarks, astar, gzip, and soplex all performed well overall. Each of these benchmarks had branches that had good reuse characteristics. These targeted branches present where the majority of the performance increase should come from in SBR. Because of this, we measured the mispredictions of the targeted branches with and without SBR. Not only will this tell us how SBR is performing in respect to these branches, but also how well SBR is finding improvements automatically in the rest of the benchmark. Because SBR is meant to be automatic without any programmer or compiler input, this metric is very important to test if SBR is performing well as a fully automatic, hardware solution.

First, the overall reduction of targeted branches is shown in Figure *5.1*.

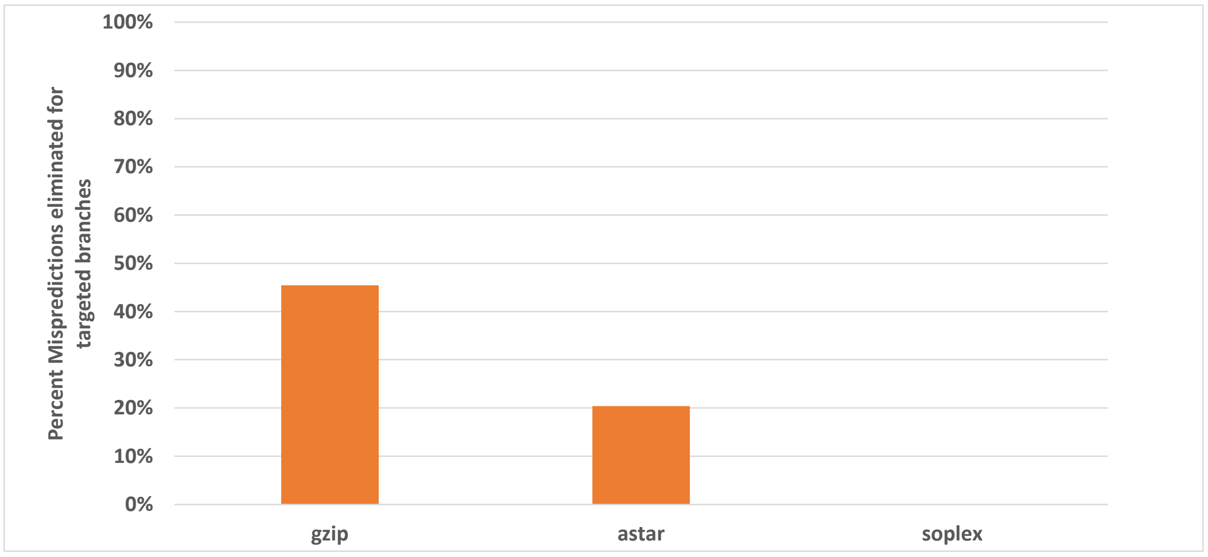

Figure 5.1 Target misprediction reduction from SBR

More gzip and astar perform well, as expected from the results shown in section 5.1. However, soplex shows a distinct lack of eliminated target mispredictions. The result, however, has to do with what it shares with astar and gzip. First, as shown in Chapters 1 and 2, gzip has a very tight loop. This results in a large amount of results being reused even in smaller window systems. Astar, however, does not have a tight loop. Its memory pattern is more erratic, causing longer loads, which can create large amounts of resolved branches before a load finishes. Soplex shares a larger loop body similar to astar, but the simpler memory pattern of gzip. This means that far fewer branches will be executed before a misprediction is resolved, resulting in a squash, the other instructions providing a buffer in between the mispredicted branch and other potential branches executing for reuse. In addition, the simpler memory pattern means that there will be fewer load misses, resulting in smaller amounts of branches executed before a squash recovery. Both of these together are possible scenarios in soplex that would affect SBR performance, and could account for the lack of reuse on the soplex targeted branches.

The second aspect of the targeted benchmark performance is how much of the mispredictions SBR eliminated came from branches other than the target branches themselves. This statistic will show how well SBR is able to find and capitalize on other branches that might not have been immediately noticeable if this process was done manually. These results can be found in Figure *5.2*.

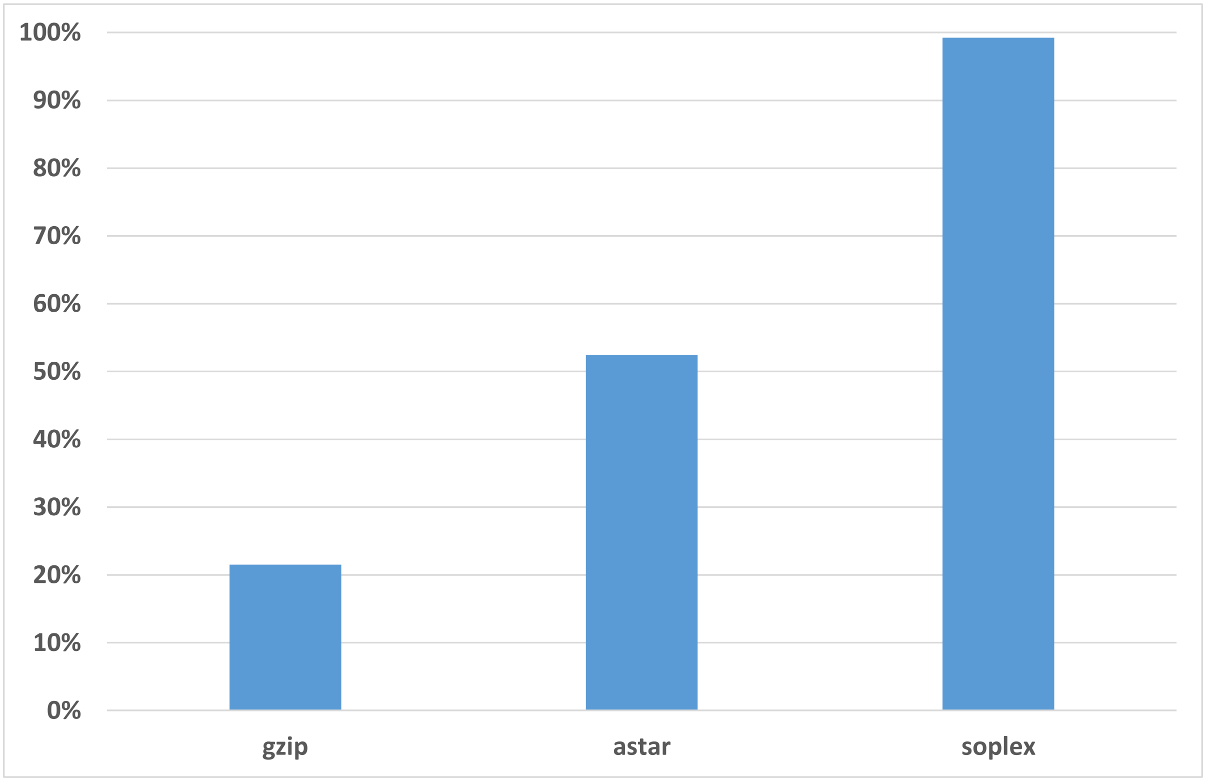

Figure 5.2 Misprediction reuse that came from branches other than targeted

The high percentage from soplex is from the behavior of soplex's target branch. Because most of the eliminated mispredictions did not come from that branch, improvements that SBR was able to create came completely from other branches in the benchmark. While the improvements in astar and gzip are much larger, the fact that soplex was still able to achieve an improvement despite the poor performance of its targeted branches shows the ability of SBR to automatically find other branches for reuse.

5.2 Misprediction Analysis

In the misprediction analysis, we compare the misprediction rates of the benchmarks with SBR and without SBR, shown in Figure *5.3*.

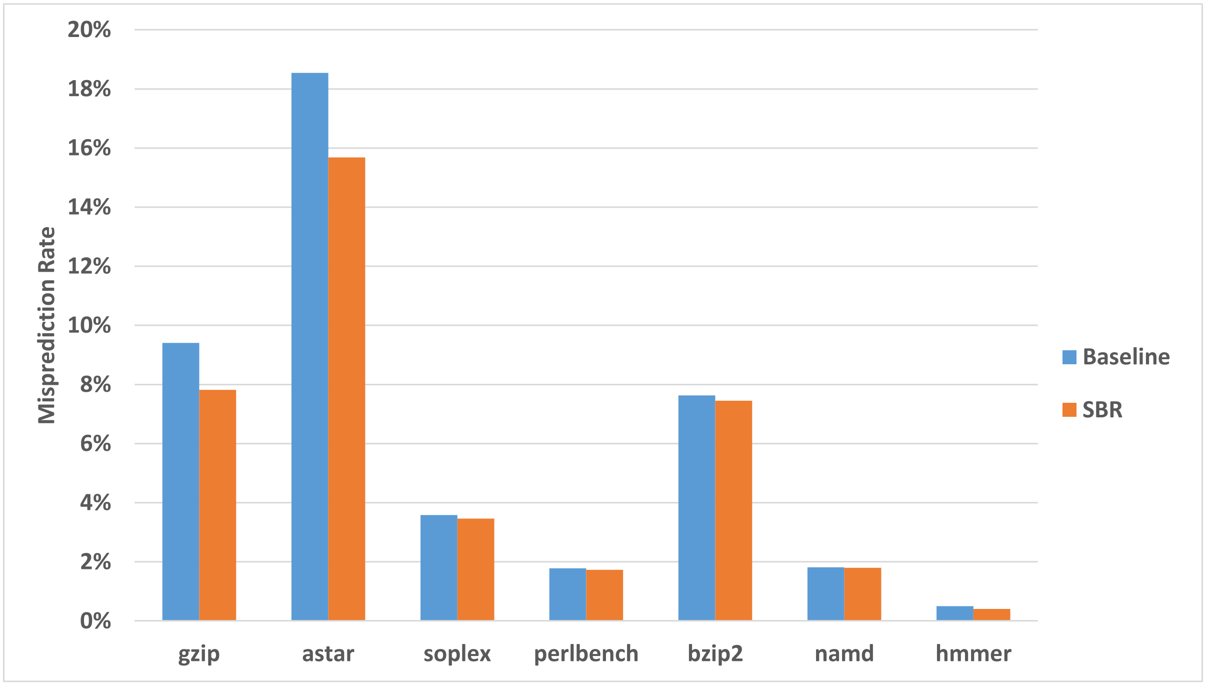

Figure 5.3 Misprediction rates with and without SBR

As the graph shows, SBR improves accuracy on every benchmark tested. On targeted benchmarks, such as gzip and astar, the improvement is quite large, near 17% for gzip and over 15% for astar, as shown in Figure *5.4*.

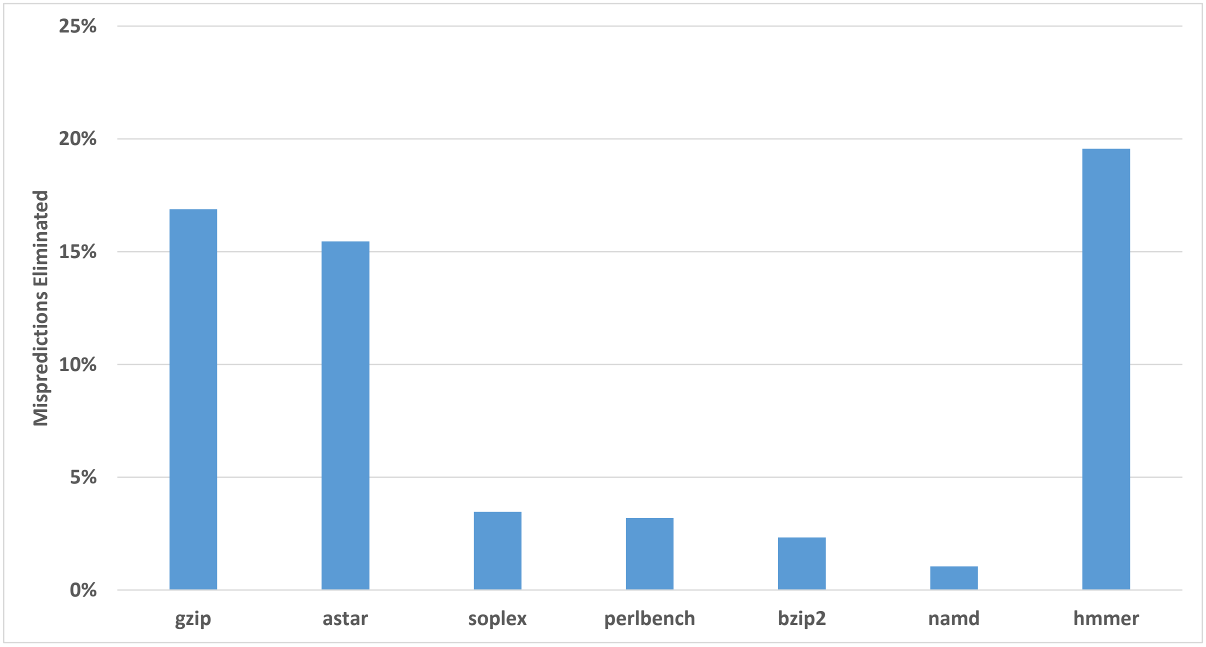

Figure 5.4 Percent of Base Mispredictions Eliminated

The benefits of SBR are shown clearly, with the targeted benchmarks gzip and astar achieving a larger percent of mispredictions eliminated than most other benchmarks.

Hmmer achieved good improvement from SBR, exhibiting the ability for SBR to add improvement for systems in an automated fashion. Branches that can be used are found and exploited without programmer input, as all of the binaries were run without any modifications specific to SBR. The other non-targeted benchmarks, perlbench, bzip2, and namd, all achieved small improvements, but not a large amount. SBR always improved the accuracy, even if the improvement was small.

The final statistic comes in analyzing the performance of the SBR versus the branch predictor. For the accuracy of the system to go up, the SBR is assumed to be more accurate than the branch predictor, by almost always choosing branches that are good for reuse.

Figure *5.5* pulls together all of the predictions for the benchmarks being run with SBR enabled. The entire bar represents the predictions made in the system, with each part of the bar broken down to show the branch predictor mispredictions and predictions, and the SBR mispredictions and predictions. In all situations, the yellow bar representing the mispredictions from the SBR is a very small percentage of the overall system, which supports that the confidence mechanism in the SBR is performing well at making sure results that are being used show behaviors positive towards reuse.

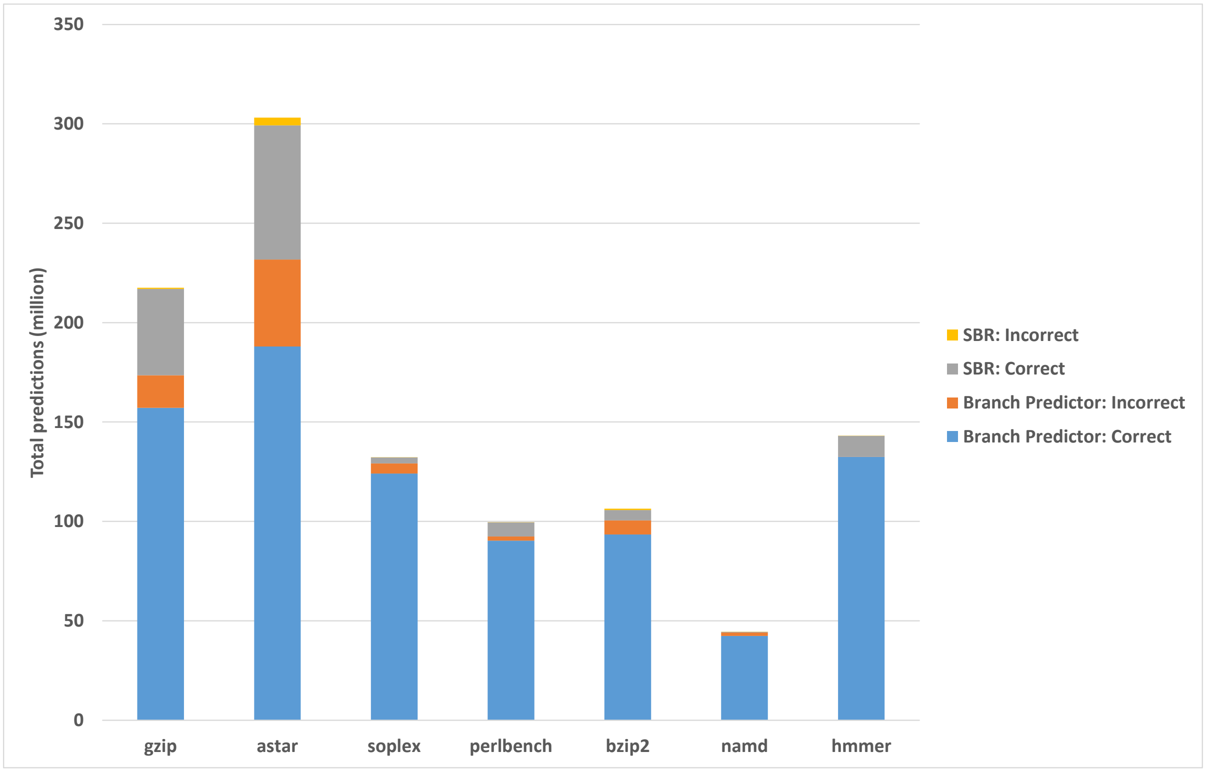

Figure 5.5 Total Prediction Breakdown

5.3 Overall Performance

In addition to measuring the accuracy of the branch predictor, we also measured the instructions per cycle (IPC) of each benchmark with and without SBR, and overall speedup. This serves to measure the quality of our accuracy gains. The results from these tests are shown in Figure *5.6* and Figure *5.7*.

The targeted benchmarks performed similarly to how they did relative to their misprediction rate. Astar and gzip both perform very well, with soplex still improving, just by a smaller amount. The other benchmarks vary more. Perlbench and bzip2 both see very slight dips in performance, with perlbench being the worse of the two at a speedup of 0.998. This loss of performance most likely comes from the mispredictions that are happening with the SBR. What mispredictions that do happen from the SBR would come from branches that exhibit common enough behavior to get through the confidence predictor. Such branches are similar to loop branches, ones that could result in a large amount of squashed work if the previous executed value changes because of a prior misprediction. This is most likely from loop branches that are dependent on data in the loop. A previously executed value stored in the SBR could be reliant on data that was changed based on the squash. This could result in a costly misprediction from the SBR. However, even in this case, the loss in performance is incredibly minor compared to the improvements in the other benchmarks, particularly gzip and astar. Bzip2 and hmmer, both being benchmarks that were not targeted, also perform well with SBR.

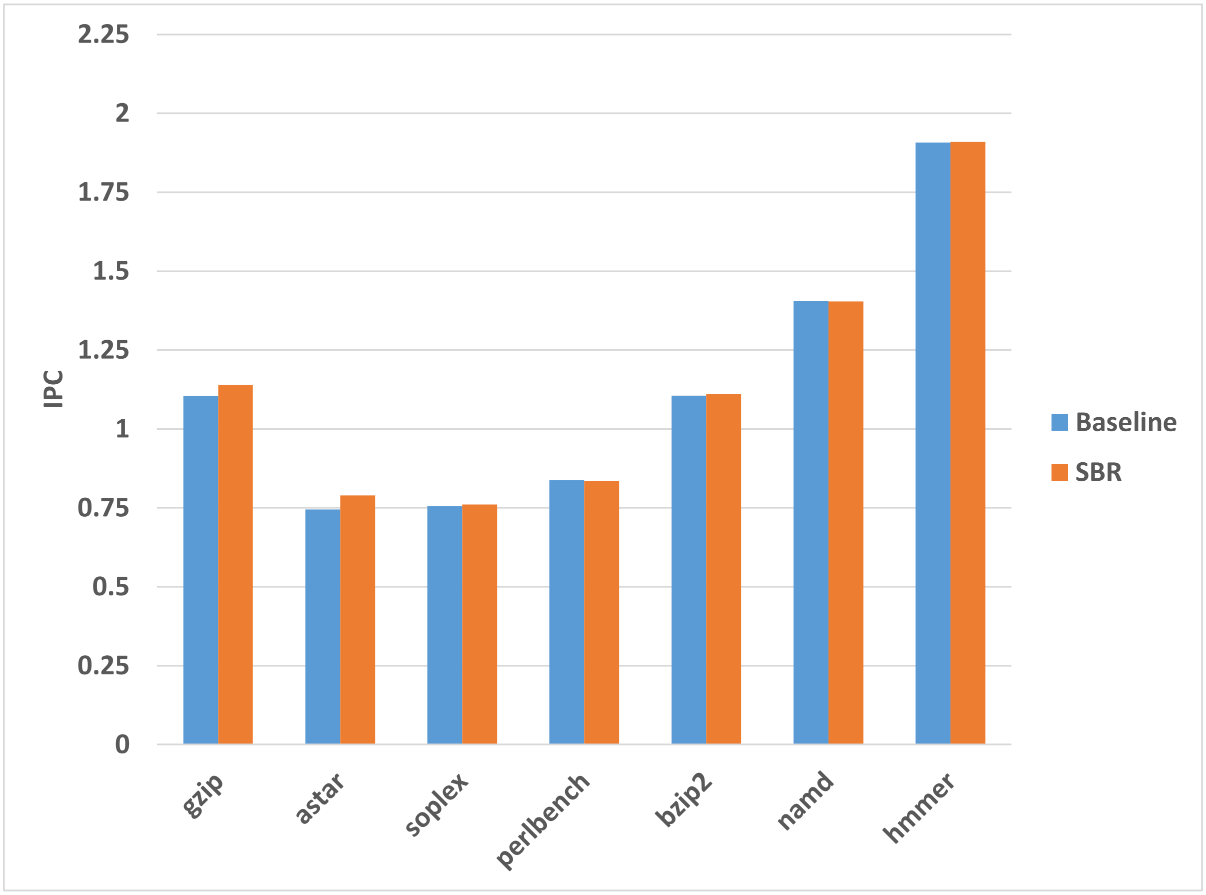

Figure 5.6 IPC with and without SBR

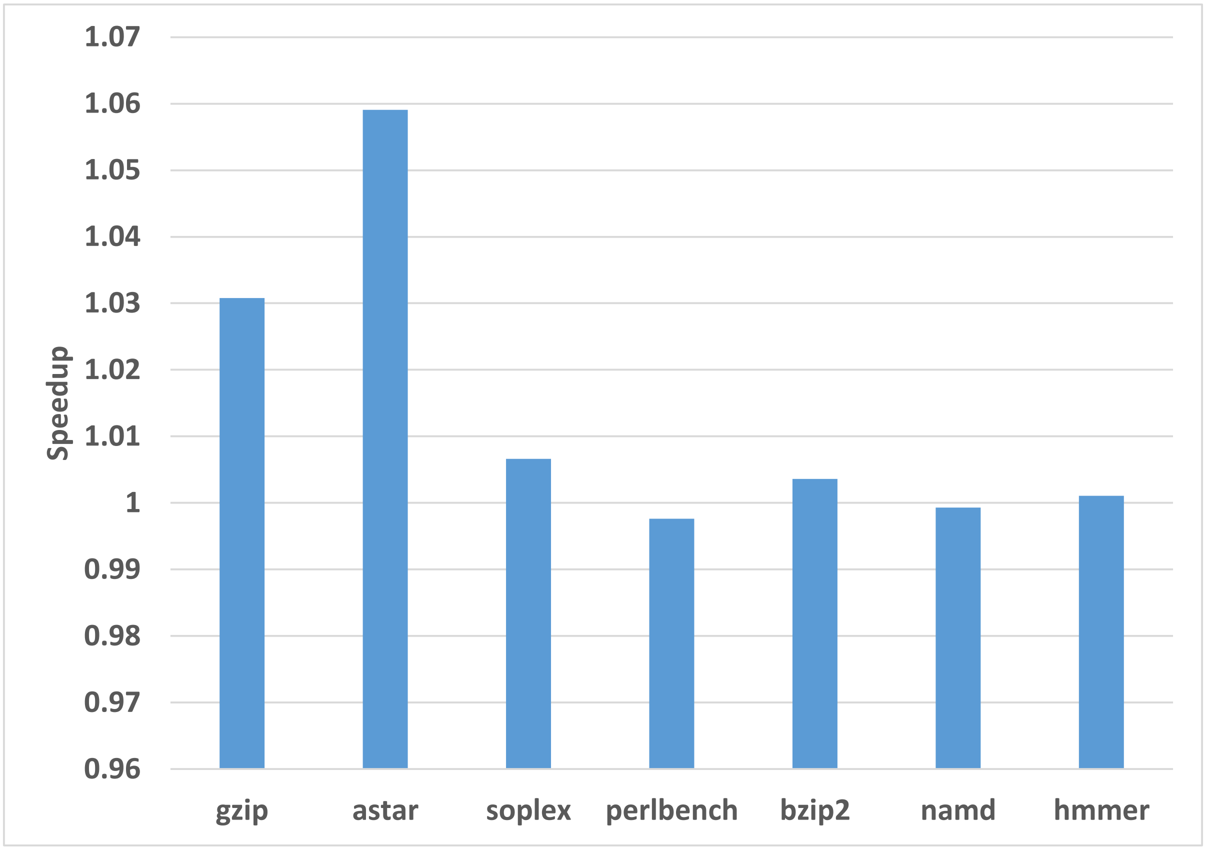

Figure 5.7 Speedup with SBR

Chapter 6

Related Work

In this chapter, two primary related mechanisms to SBR are discussed, as well as SBR in light of these mechanisms. The two mechanisms examined are control flow decoupling (CFD), and speculative update of the branch predictor.

6.1 Control Flow Decoupling

CFD is a novel technique that seeks to separate predictions from their dependent slices, instead choosing to do the work of the control slice separately and push those results on to a queue that is used later on to determine whether a dependent slice should be executed or not [1]. In this way, CFD seeks to remove predictions from these branches altogether.

CFD worked through three parts.

1. In software, the branch can be separated in to the control slice and the dependent slice. The control slice is brought up above the dependent slice and computed in a separate loop.
2. The ISA is modified to support a new set of registers that communicate branch results, called the architectural branch queue.
3. The architectural branch queue resides in the fetch unit for timely, non-speculative branching.

SBR distinguishes itself from CFD primarily in its simplicity and automated nature. Where CFD had to make changes to the pipeline and ISA to be able to support the architectural branch queue, SBR lives alongside the existing branch predictor, and only interacting with the same messages that come to the branch predictor normally. Because of this, the ISA does not need to be changed, extra access in the pipeline does not need to be added, and SBR becomes ambiguous to whatever ISA, programming language, or compiler is used, at the cost of having to rely on recovering squashed values.

6.2 Speculative Branch Update

Speculatively updating the branch predictor comes closer to the original idea of SBR, in an even simpler technique. Instead of updating the branch predictor upon retirement, instead, the branch predictor is updated after execute. The goal is to allow the branches that would be squashed to train the branch predictor prior to the squash, therefore improving the branch predictor's performance for that branch. This technique is very simple, as it does what is normally done later in the pipeline earlier, but doesn't add any new hardware queues or other algorithmic changes.

At first glance, speculative branch update seems to accomplish what SBR attempted to do. Provide better prediction for hard-to-predict branches, while keeping the solution simple and fully automated. However, this strategy proves ineffective for the hard-to-predict branches targeted by SBR. SBR seeks to capture these branches to be reused during the squash caused by a different misprediction. Because of the differing history, a speculative update in the branch predictor could lead to an update of an entry that would not be used when the branch is predicted again. In this way, speculatively updating the predictor can lead to poisoning of other entries as well. SBR is novel in that it targets these hard-to-predict branches without blindly poisoning other entries.

6.3 Squash Reuse

Squash Reuse and its implementation in Register Integration is a technique introduced by Dr. Amir Roth and Dry. Gurindar S. Sohi [7]. In their technique, they attempt to save instructions that have been squashed, then integrate those results back in to the instruction stream. Similar to SBR, they analyze instructions to see if they are control and data independent before deciding to reuse those results. The structures to do this are integrated in rename, in a stage called Rename Integrate that follows fetch.

SBR differentiates itself from Squash Reuse through Register Integration by focusing not on all instructions, but only branch prediction. SBR does not require any of the other pipeline modifications that Squash Reuse requires, meaning that SBR can be integrated in more superscalar microarchitectures. Furthermore, SBR simplifies control and data independence by only analyzing branches, resulting in simpler mechanisms overall, while still achieving benefits from reuse.

Chapter 7

Conclusion and Future Work

In this study, we have examined the benefits of intelligent reuse, focusing on characteristics that qualify branches to benefit from reuse, namely control independence, data independence, and a large amount of these branches being able to fit inside the processor window. We detailed the benefits of SBR, and then examined SBR's behavior on a set of benchmarks containing known branches of interest. Furthermore, we tested SBR's impact on benchmarks where the characteristics of the branches were not known. SBR lead to improvements in the branch predictor accuracy in all cases. Furthermore, SBR was able to achieve a speedup in the benchmarks of interest, while keeping detrimental impact small on benchmarks where SBR did not perform as well, showing the value of intelligent branch reuse.

SBR has other avenues that need to be studied and could lead to further improvement. First, the confidence mechanism serves as the primary way of filtering out which branches to reuse, and which branches should not. The reset counter works well, but leads to a delay before results are used from branches of interest, and can still lead to some branches getting through that could harm the overall performance of the benchmark. Different confidence mechanisms could lead to branches of interest being identified earlier, and other branches that are not desired to be filtered out more effectively. While varying the reset bits is one small change, this only exacerbates the problem that another confidence mechanism could potentially solve.

SBR is also heavily reliant on the amount of predictions that actually get executed before the squash occurs. Because of this, it naturally lends itself to larger window structures and longer queues, as these could result in more branches being executed before they are squashed. Studying SBR under different processor architectures that could lead to larger amounts of branches being stored could lead to higher improvements.

Lastly, the different possible branch predictors and misprediction recovery mechanisms both can affect the amount of mispredictions that are available for reuse. Both of these can be varied, to see how well SBR performs in these different scenarios. A different misprediction recovery method could lead to fewer mispredictions overall in the Branch Queue at the time of a squash recovery. A different branch predictor may be more accurate overall, leading to fewer mispredictions available for reuse, or actually be worse, leading to more mispredictions available for reuse. Both can be varied to test SBR more thoroughly.

REFERENCES

[1] R. Sheikh, J. Tuck and E. Rotenberg, "Control-Flow Decoupling," *IEEE/ACM International Symposium on Microarchitecture,* vol. 45, pp. 329-340, 2012.

[2] "SPEC CPU200 V1.3," Standard Performance Evaluation Corporation, 07 June 2007. [Online]. Available: www.spec.org/cpu2000/. [Accessed 24 March 2015].

[3] "SPEC CPU 2006," Standard Performance Evaluation Corporation, 10 June 2014. [Online]. Available: http://www.spec.org/cpu2006/. [Accessed 24 March 2015].

[4] E. Jacobson, E. Rotenberg and J. E. Smith, "Assigning Confidence to Conditional Branch Predictions," *Annual International Symposium on Microarchitecture.,* vol. 29, 1996.

[5] "MIPS64 Architecture," Imagination, [Online]. Available: http://www.imgtec.com/mips/architectures/mips64.asp. [Accessed 25 March 2015].

[6] B. Valentine, *Introducing Sandy Bridge,* San Francisco: Intel Developer Forum, 2010.

[7] A. Roth and S. G. Sohi, "Register Integration: A Simple and Efficient Implementation of Squash Reuse," *Annual IEEE/ACM International Symposium on Microarchitecture,* vol. 33, 2000.

[8] A. Seznec, "A 64 KBytes ISL-TAGE branch predictor," *3rd Championship Branch Prediction,* 2011.