Abstract

AL-OTOOM, MUAWYA MOHAMED. EXACT: Explicit Dynamic-Branch Prediction with Active Updates. (Under the direction of Dr. Eric Rotenberg.)

Branches that depend on load instructions are a leading cause of mispredictions by state-of-the-art branch predictors. An in-depth study of mispredictions reveals two problems:

(i) The context used by state-of-the-art branch predictors (global branch history) often fails to distinguish multiple instances of a branch that test different elements of a large data structure (e.g., arrays). The predictor is unable to specialize predictions for different elements, causing mispredictions if their corresponding branch outcomes differ.

(ii) A store to an element of the data structure may cause its corresponding branch outcome to flip when it is next encountered. A misprediction is suffered before the predictor is retrained.

This work proposes a new branch predictor, called EXACT, to address these two problems. First, the context for predicting an instance of a branch is based on the address of the element it tests. Using addresses ensures dedicated predictions for different elements. Second, stores to elements directly update their predictions in the predictor. This novel “active update” concept avoids mispredictions that are otherwise incurred by conventional passive training.

We encountered three major implementation challenges while developing EXACT:
1) Using addresses to index into the predictor is difficult to do in practice, because loads are unlikely to have generated their addresses by the time their dependent branches are fetched.

2) The most problematic aspect of active updates is converting a store’s address into a predictor index that must be updated. The complexity and storage cost of address-to-index conversion is potentially prohibitive.

3) Providing a dedicated prediction for each dynamic branch generally requires a large prediction table. The chief problem with a large prediction table is its long access time.

This work explores two implementations of EXACT, the first is a hardware-only solution (EXACT-H) and the second is a combined hardware/software solution (EXACT-S).

EXACT-H deals with the first implementation issue by basing the predictor index for the current branch not on its own load address (which is unavailable) but the load address of a prior retired branch. An unfortunate but necessary side-effect of this indirect indexing strategy is that a branch’s index into the predictor cannot be deduced from the load address on which it depends. This leads to a complex and storage-intensive active update unit: a large table is needed to convert store addresses to the predictor indices that must be updated. EXACT-H limits the amount of dedicated storage for its address-to-index conversion table by virtualizing it (caching it in the general-purpose memory hierarchy), exploiting the key observation that active updates are tolerant of 100s of cycle of latency.
EXACT-S deals with the first implementation issue by enabling the programmer or compiler to convey key information directly to the fetch unit that it can use to generate branches’ load addresses in a timely manner. This yields a number of benefits over EXACT-H. EXACT-S is more accurate because it uses branches’ load addresses directly rather than prior branches’ load addresses. Moreover, with regard to active updates, there is no need for a large table to convert store addresses to predictor indices because of the direct indexing strategy. As a result, active updates are simple and inexpensive in EXACT-S.

Regarding the last issue of the predictor’s size and latency, we show that the prediction tables of both EXACT-H and EXACT-S are scalably pipelinable because consecutive indices do not depend on the output of the predictor.
EXACT: Explicit Dynamic-Branch Prediction with Active Updates

by

Muawya Mohamed Al-Ottoom

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

Computer Engineering

Raleigh, North Carolina

2010

Approved by:

______________________               ______________________
Dr. Eric Rotenberg                              Dr. James M. Tuck
(Committee Chair)

______________________               ______________________
Dr. Gregory T. Byrd                              Dr. Ken V. Vu
(IBM)

______________________               ______________________
Dr. Xiaosong Ma                                Dr. James M. Tuck
(IBM)
Dedication

To my beloved parents
To my lovely wife Sajeda
To my dear daughter Julia
To all of my family members
Biography

Muawya Al-Otoom was born in Amman in 1981 as the second son for Mohamed and Raisa Al-Otoom. After he lived in Amman for seven years, he moved with his family to their native town of Souf where he attended his middle and high school. In 1999, and after he finished his high school, he started his undergraduate studies in Computer Engineering at Jordan University of Science and Technology (JUST). In 4 years, Muawya finished his Bachelor’s degree from JUST with honors, and worked as a teaching and research assistant at JUST for one year. In 2004, he got accepted in the Computer Engineering graduate program at North Carolina State University where he started his research in High-Performance Microarchitectures under the direction of Prof. Eric Rotenberg. In 2006, he finished his Masters degree in Computer Engineering. In his dissertation, he studied new possibilities for control-independence architectures. Soon after the completion of his Master’s degree, Muawya started his PhD program in NCSU also under the supervision of Prof. Eric Rotenberg. Muawya’s main focus of research was centered around solving the problem of branch prediction. During his PhD studies, Muawya got the chance to intern at Intel, IBM, and AMD where he worked with some talented people from the industry on the development on future commercial processors. In 2009, Muawya got married to Sajeda Tamimi, and they had their first baby Julia in 2010. Upon the completion of his PhD degree in Computer Engineering, he will be starting his first job at Intel Corporation in Hillsboro, Oregon. Muawya is a member of Phi Kappa Phi and IEEE.
Acknowledgments

First I would like to thank God for everything he gave me and everything I have today.

I would like to thank my parents General Dr. Mohamed Al-Otoom, and Raisa Al-Otoom. Through my journey they were a great support and motive for me to succeed. My wife Sajeda has been the most successful factor in my career, she supported all of my correct decisions, and even the wrong ones she made sure to stay the course with me to fix them. How can I forget the lovely Julia, since the day she arrived to our lives all she is bringing is good luck. Many thanks to my big brother Muaz Baik, sister in law Mariam, and their son Awn, without those Western Union checks the last year would have been difficult. My three evil sisters: Doa’a, Dima, and Amomah, thanks for your support. Thanks to my parents in law: Prof. Abdul Rahman Tamimi and Bushra Kilani. My brothers and sisters in law: Aya, Ibrahim, and Ahmed Tamimi. My uncles: Prof. Jalal and Prof. Awni Al-Otoom.

I would like to have special thanks to my Advisor Prof. Eric Rotenberg. His non-stopping help, attention, and guidance kept me going forward even during the rough times. During my six year grad school I learned a lot from him. My grad school journey was not an easy one, but sure after finishing now, I am proud of what we have achieved.
I would like to acknowledge my PhD committee members: Greg Byrd, James Tuck, Xiaosong Ma, and Ken Vu. Thanks for your comments during my preliminarily exam, they helped to improve the quality of my final dissertation. I would like to thank Sandy Bronson and Linda Fontes for their help in setting up my research assistantship during my grad school.

I would like to acknowledge some of the industry folks for their help during my internships. From Intel: Haitham Akkary, Srikanth Srinivasan, Konrad Lai, and Shadi Khasawneh. From IBM: Timothy Heil, Anil Krishna, Brian Rogers, and Ken Vu. From AMD: Adithya Yalavarti, Scot Hildebrandt, Tony Jarvis, Ravi Bhargava, Trivikram Krishnamurthy and Swamy Punyamurtula.


This research was supported in part by NSF grants CCF-0702632 and CCF-0916481, SRC grant 2007-HJ-1594, and an Intel gift. Any opinions, findings, and conclusions or
recommendations expressed herein are those of the author and do not necessarily represent the views of the National Science Foundation.
Table of Contents

List of Figures .................................................................................................................................................. xi
List of Tables ..................................................................................................................................................... xv

Chapter 1: Introduction ................................................................................................................................. 1

1.1 Background ............................................................................................................................................. 2

1.1.1 Superscalar Processors and ILP ................................................................. 2

1.1.2 Branch Instructions and Branch Prediction ...................................................... 3

1.1.3 Quantifying the Effect of Branch Mispredictions on Performance ................... 5

1.2 Understanding the Sources of Branch Mispredictions .................................................... 9

1.2.1 Definitions ........................................................................................................ 9

1.2.2 Code Example ................................................................................................ 11

1.3 EXACT: Explicit-Dynamic Branch Prediction with Active Updates ....................... 18

1.3.1 EXACT-H: A Hardware-only Solution ....................................................... 25

1.3.2 EXACT-S: A Combined Hardware/Software Solution .................................. 32

Chapter 2: Characterizing Mispredictions ................................................................................................. 39

2.1 Measuring the Ability of History-Based Predictors to Mirror a Program’s Data Objects ................................................................................................................................. 40

2.2 Measuring the Effect of Stores on Modifying Branch Outcomes ............................. 44

Chapter 3: The Microarchitecture of EXACT-H .................................................................................. 49

3.1 ID Generation Unit ......................................................................................................................... 50

3.2 Global Branch Queue .................................................................................................................... 51
3.3 Indexing the Explicit Predictor ................................................................. 51
3.4 Pipelining the Explicit Predictor ............................................................... 55
3.5 Explicit Loop Predictor ............................................................................ 57
3.6 Chooser .................................................................................................... 59
  3.6.1 Choosing the Default Predictor ....................................................... 59
  3.6.2 Choosing the Explicit Loop Predictor .............................................. 60
  3.6.3 Declining Global Branch History ...................................................... 61
  3.6.4 Summary of Chooser Contents ....................................................... 61
3.7 Active Update Unit .................................................................................. 62
  3.7.1 Store Address Conversion ................................................................. 62
    3.7.1.1 SACT ......................................................................................... 65
    3.7.1.2 MACT-A and MACT-B ............................................................... 65
  3.7.2 Store Value Conversion .................................................................... 66
    3.7.2.1 GRT .......................................................................................... 66
    3.7.2.2 RRT .......................................................................................... 67
Chapter 4: Evaluating EXACT-H .................................................................... 69
  4.1 Methodology ......................................................................................... 69
  4.2 Impact of Real Indexing ....................................................................... 71
  4.3 Impact of Active Update Latency ......................................................... 72
  4.4 Accuracy vs. Storage Budget ............................................................... 74
    4.4.1 Dedicated Storage for SACT ........................................................... 77
    4.4.2 Virtualized SACT ......................................................................... 77
List of Figures

Figure 1. Instruction window of a superscalar processor with out-of-order execution............ 3
Figure 2. Instruction window with branch instructions.......................................................... 4
Figure 3. Effect of branch mispredictions on performance. (a) Ideal data cache. (b) Real data cache......................................................................................................................... 7
Figure 4. (a) A static branch that depends on two static loads. (b) Two dynamic instances of the static branch, one that depends on addresses A1 and B1 and another that depends on addresses A2 and B2. .............................................................................................................. 10
Figure 5. (a) Good scenario: different dynamic branches access different prediction table entries, because different global branch history patterns (P1 versus P2) precede the dynamic branches. (b) Bad scenario: different dynamic branches access the same prediction table entry, because the same global branch history pattern (P1 = P2) precedes the dynamic branches......................................................................................................................... 11
Figure 6. Code example. ........................................................................................................ 13
Figure 7. Good scenario # 1. ............................................................................................... 14
Figure 8. Good scenario # 2. ............................................................................................... 14
Figure 9. Bad scenario # 1................................................................................................. 15
Figure 10. Indirect solution for bad scenario # 1................................................................. 16
Figure 11. Using longer history does not always work............................................................ 17
Figure 12. Bad scenario # 2............................................................................................... 18
Figure 13. First principle: The branch predictor should mirror a program’s data objects. (Best if viewed in color.) ................................................................. 19

Figure 14. Second principle: The branch predictor should *actively* mirror a program’s data objects. (Best if viewed in color.) ........................................... 21

Figure 15. Contributions to mispredictions by different IDs .................................................. 23

Figure 16. High-level view of EXACT-H............................................................................. 26

Figure 17. Two scenarios for using the proxy ID: (a) No control-flow dependence. (b) Control-flow affects which ID is to be predicted by the proxy ID. .................................. 27

Figure 18. High-level view of EXACT-S. ........................................................................... 32

Figure 19. Breakdown of (a) all branches, (b) just correctly predicted branches, as a percentage of all dynamic branches, and (c) just mispredicted branches, as a percentage of all dynamic branches. .................................................................................. 41

Figure 20. Combining the idealized explicit dynamic-branch predictor with (a) *gselect* and (b) *L-TAGE* .................................................................................................................. 46

Figure 21. Major components of EXACT-H. .................................................................... 49

Figure 22. Hash function for generating a dynamic branch’s ID.......................................... 51

Figure 23. The index for a branch is based on a prior branch’s ID some fixed distance away (20 in this example) ......................................................................................... 52

Figure 24. Example index function, described in C code. .................................................. 53

Figure 25. Effect of distance. (a) Prior branch is always retired. (b) Retired status based on processor simulation ................................................................. 54
Figure 26. Pipelining the explicit predictor. Diagram shows three consecutive accesses to a three-cycle-latency predictor. .............................................................. 56

Figure 27. Example for-loop in the sub_penal() function in twolf. The trip-count of the loop branch is dynamic and transitively depends on three heap loads in the caller function ucxx1(). The transitive dependence is shown with boldface type. ........................................ 57

Figure 28. A single entry in the explicit loop predictor. .............................................................. 58

Figure 29. Summary of chooser contents. .............................................................................. 61

Figure 30. Store address conversion, and its interaction with store value conversion and the predictor. ..................................................................................................................................... 63

Figure 31. SACT, MACT-A, and MACT-B. ................................................................................ 65

Figure 32. GRT and RRT entries. ...................................................................................... 67

Figure 33. Misprediction rates with large predictors. ............................................................... 71

Figure 34. Misprediction rate vs. active update latency............................................................ 73

Figure 35. Misprediction rate versus cost. (Note the log-scale x-axis and the y-axis does not start at 0%.) .................................................................................................................................................. 76

Figure 36. Impact of adding a 4KB EX-cache and 16KB overhead to different L-TAGE predictor sizes.................................................................................................................................................. 81

Figure 37. Major components of EXACT-S. ........................................................................... 83

Figure 38. The fetch unit registers read and written by the shadow code............................... 85

Figure 39. General format of a shadow-instruction. ................................................................. 88

Figure 40. Gzip source code.................................................................................................... 94

Figure 41. Assembly code and its associated shadow code for gzip. ..................................... 95
List of Tables

Table 1. Misprediction rates for the large gshare branch predictor used in this section. In the graphs of Figure 3, the 0% point represents performance with these accuracies...................... 6

Table 2. Microarchitecture parameters for EXACT-H. ........................................................... 69

Table 3. Fixed-configuration subcomponents for EXACT-H. Not included: default predictor, explicit predictor, and SACT. ................................................................. 70

Table 4. Summary of shadow-instructions used in this thesis. .............................................. 92

Table 5. Microarchitecture parameters of the modeled processor used in the evaluation of EXACT-S......................................................................................................... 105

Table 6. Fixed-configuration subcomponents for EXACT-S. Not included: default predictor and explicit predictor................................................................. 106

Table 7. Access latency for overriding predictor as a function of budget. ......................... 110
Chapter 1: Introduction

The important trend of placing multiple cores on a single chip has apparently shifted the research spotlight away from high-performance processor architectures and instruction-level parallelism (ILP), to chip-level architectures and thread-level parallelism (TLP). In reality, the diversity across and within workloads is too great to exclude either approach. Microprocessor companies continue to develop flagship high-performance cores (*e.g.*, AMD’s K10 [18] and Intel’s Nehalem [1]), even placing two, four, or more of these large cores on a single chip. Looking forward, a compelling strategy is to include a robust mix of core types in an Asymmetric Chip Multiprocessor (ACMP), *e.g.*, several flagship large cores and many simple cores, to support both low latency and high throughput [23][28][36][37][53]. Low latency is critical for serial workloads and serial regions of parallel workloads.

Continued microarchitecture performance scaling is hindered by many factors. One factor above all, the branch prediction bottleneck, constrains the ability to tackle other factors: the lookahead capability afforded by branch prediction exposes ILP for combating data dependences and memory latency or acts as a catalyst for other speculative techniques aimed at extracting more ILP [19].
1.1 Background

1.1.1 Superscalar Processors and ILP

In superscalar processors [49], ILP is harnessed by means of buffering a large pool of instructions called the “instruction window”. This pool of instructions is examined to find as many data-independent instructions as possible to be executed in parallel. Figure 1 shows the instruction window of a superscalar processor with out-of-order execution. Instructions with black shading represent instructions that have already been executed and retired from the instruction window. Instructions with blue shading represent future instructions that have not been fetched yet into the instruction window. The rest of the instructions are the in-flight instructions that are buffered in the instruction window to be examined for execution. Among these instructions, some are ready for execution (green shading) and some are not-ready (grey shading). Ready instructions are independent of each other and can execute in parallel (and not in program order) as long as there is enough issue bandwidth. After these instructions are issued and executed, they produce values that will allow previously not-ready instructions to issue and execute.
The ability to extract more ILP hinges on forming a larger instruction window. A larger window exposes more ready-to-execute instructions. Several microarchitectures [2][29][52] have been proposed to form large virtual instruction windows (thousands of in-flight instructions) with small physical resources (small physical register file, moderate number of checkpoints, etc.). However, the accuracy of the branch predictor was shown to be a limiting factor. Branches limit how far ahead into the dynamic instruction stream the processor can accurately fetch.

1.1.2 Branch Instructions and Branch Prediction

Each cycle, the fetch unit of an N-way superscalar processor attempts to fetch a block of N sequential instructions starting at the current program counter (PC). If there are no branches in the fetch block, the fetch unit simply increments the PC by the size of the fetch block. On the other hand, if there is a conditional branch in the current fetch block, the PC of the next fetch block depends on whether or not the conditional branch is taken. The conditional branch's outcome is not known until it executes. Stalling the fetch unit until the branch
executes would severely limit the size of the instruction window because branches are frequent, occurring about one in every five to ten instructions. Therefore, superscalar processors use branch prediction to avoid fetch disruptions. The overall branch prediction logic includes (1) detecting branches within the current fetch group, (2) predicting the taken-targets of branches in the current fetch group in the event that one is to be taken, and (3) predicting the directions (taken vs. not-taken) of conditional branches within the fetch group. The first two aspects are handled by the branch target buffer (BTB), which is usually very effective at detecting branches and producing their taken-targets since the information is static. The third aspect is the most challenging one because the taken/not-taken outcome is data dependent. It is the job of the conditional branch predictor to predict whether a conditional branch in the fetch group is taken or not-taken. This work focuses on just the conditional branch predictor.

![Instruction Window with Branch Instructions](image)

Figure 2. Instruction window with branch instructions.
Figure 2 shows the same instruction window presented in Figure 1, but with some of the not-ready instructions highlighted as conditional branch instructions (red shading). Predicting the three conditional branches enabled the fetch unit to fetch beyond them, fill the large window, and thereby look deeper in the window to extract ILP (shown by allowing 4 instructions to issue in the same cycle). If the first branch was mispredicted, all instructions fetched after the branch will have to be squashed. The misprediction reduces the effective size of the instruction window, which in the example reduces the extracted ILP from 4 to 1.

1.1.3 Quantifying the Effect of Branch Mispredictions on Performance

To understand the effect of branch mispredictions on performance, the following experiment was performed. We used a 4-way superscalar processor – its peak fetch/issue rate is 4 instructions-per-cycle (IPC) – that supports as many as 1,024 in-flight instructions (this is the instruction window size) and has an unusually large (2MB) gshare branch predictor [33][38][55]. We used the 100 million instruction SimPoints [48] of 11 SPEC2K integer benchmarks. The branch misprediction rates for their SimPoints are shown in Table 1. For each benchmark, we ran 100 experiments. In the first experiment, we randomly removed 1% of the branch mispredictions and measured performance in instructions-per-cycle (IPC) for that run. Then we tried the same experiment again, but this time randomly removed 2% of the mispredictions, and measured the IPC. We repeated the same process until 100% of the mispredictions were removed, which is tantamount to perfect branch prediction.
Table 1. Misprediction rates for the large gshare branch predictor used in this section. In the graphs of Figure 3, the 0% point represents performance with these accuracies.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Misprediction Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>4.45%</td>
</tr>
<tr>
<td>crafty</td>
<td>4.26%</td>
</tr>
<tr>
<td>gap</td>
<td>1.65%</td>
</tr>
<tr>
<td>gcc</td>
<td>3.77%</td>
</tr>
<tr>
<td>gzip</td>
<td>9.06%</td>
</tr>
<tr>
<td>mcf</td>
<td>0.26%</td>
</tr>
<tr>
<td>parser</td>
<td>6.00%</td>
</tr>
<tr>
<td>perlbmk</td>
<td>1.88%</td>
</tr>
<tr>
<td>twolf</td>
<td>10.52%</td>
</tr>
<tr>
<td>vortex</td>
<td>0.55%</td>
</tr>
<tr>
<td>vpr</td>
<td>9.43%</td>
</tr>
</tbody>
</table>

Figure 3 shows the results of these experiments. The graphs show how IPC increases as the percentage of mispredictions removed varies from 0% to 100%. IPC at the 0% point corresponds to performance with the unaltered misprediction rates shown in Table 1. IPC at the 100% point corresponds to performance with perfect branch prediction (0% misprediction rate).
(a) ideal data cache

(b) real data cache

Figure 3. Effect of branch mispredictions on performance. (a) Ideal data cache. (b) Real data cache.
The benchmarks are divided into four groups based on their branch misprediction rates and cache miss rates:

- Bad branch prediction accuracy, good data cache behavior: bzip2, gzip, and twolf.
- Good branch prediction accuracy, good data cache behavior: crafty, gap, gcc, perl, and vortex.
- Good branch prediction accuracy, bad data cache behavior: mcf.
- Bad branch prediction accuracy, bad data cache behavior: parser and vpr.

In Figure 3, graphs in the left column show results using an ideal data cache (0% data cache miss rate) while graphs in the right column show results with a real data cache.

We are interested in benchmarks from the first and last groups. Benchmarks in the first group have good data cache behavior, which is why their performance does not improve significantly going from the real to the ideal data cache. Removing mispredictions clearly improves the performance of these benchmarks. Achieving perfect branch prediction could double (or almost triple) the performance of these benchmarks. Actually, it not only doubles their performance, but IPC nearly reaches its theoretical maximum of 4 IPC (4-way superscalar). Similarly, benchmarks in the fourth group experience the same behavior, but with either a bigger cache or cache-enhancing techniques such as data prefetching.

These experiments show the importance of fixing the remaining 5-10% of branches that are mispredicted. Section 1.2 explores the nature of those remaining mispredictions.
1.2 Understanding the Sources of Branch Mispredictions

1.2.1 Definitions

Today’s best known branch predictors push the envelope of what is possible using global branch or path history as context for making predictions. While this context is basically the same used by precursor predictors since the advent of two-level adaptive branch prediction [55], clever combinations and organizations have yielded nearly perfect branch prediction on some programs and program phases. Yet, results in this dissertation show that branch history alone cannot scale accuracy in other programs beyond 90-95%. In these programs, the leading cause of branch mispredictions are branches that depend directly or indirectly on load instructions.

An example of this type of branches is depicted in Figure 4(a). It shows a static branch at program counter (PC) Z that depends on two static loads at PCs X and Y. At run-time, the static branch translates into many different dynamic branches corresponding to different combinations of load addresses. Two dynamic instances of the branch are shown in Figure 4(b). In the first instance, the two load instructions load from addresses A1 and B1, respectively. In the second instance, the two load instructions load from addresses A2 and B2, respectively. Thus, it is the combination of load addresses that distinguishes one dynamic branch from another. More generally, a dynamic branch is uniquely identified by the combination of its PC and the addresses of loads on which it depends directly or indirectly. We call this combination the identity, or ID, of the dynamic branch. The IDs of the two dynamic branches in Figure 4(b) are \{Z, A1, B1\} and \{Z, A2, B2\}, respectively.
Many state-of-the-art branch predictors \[26\],[34],[35],[41],[42],[43] exploit global branch history as context for predicting dynamic branches. When a static branch’s PC is combined with global branch history, the static branch uses multiple prediction table entries instead of just one, an entry for each unique global branch history pattern preceding the branch. Ideally, this enables specializing predictions to different dynamic instances of the static branch. For example, for the static branch Z of Figure 4, a particular global branch history pattern, P1, may precede dynamic branch \{Z, A1, B1\} and a different pattern, P2, may precede dynamic branch \{Z, A2, B2\}. As shown in Figure 5(a), the two dynamic branches access different prediction table entries because they use different indices formed from \{Z, P1\} and \{Z, P2\}, respectively. This is advantageous if the two dynamic branches have different outcomes. They each have a dedicated entry in the prediction table for making different predictions. In a
sense, the goal of combining PC with global branch history is to forecast which dynamic branch, \textit{i.e.,} which ID, is currently being fetched and to provide a dedicated prediction for it.

![Figure 5](image)

\begin{center}
(a) Good scenario: different dynamic branches access different prediction table entries, because different global branch history patterns (P1 versus P2) precede the dynamic branches. (b) Bad scenario: different dynamic branches access the same prediction table entry, because the same global branch history pattern (P1 = P2) precedes the dynamic branches.
\end{center}

On the other hand, if the two dynamic branches are preceded by the same global branch history patterns (P1=P2), they share a prediction table entry. This scenario is depicted in Figure 5(b). One or both of the dynamic branches will be mispredicted if their outcomes differ.

\subsection{1.2.2 Code Example}

This section demonstrates the scenarios explained in Section 1.2.1 using a code example, similar to code in the gzip benchmark. The example is shown in Figure 6. The code has a \texttt{while} loop that iterates some number of times. Inside the \texttt{while} loop, there is a \texttt{for} loop. The \texttt{for} loop iterates over an array \texttt{a} that has 16 entries. In the original code in gzip, the size of the array is much larger. In this example, we are interested in the \texttt{if} branch that tests whether or not each element of \texttt{a} is greater than 10. The corresponding pseudo-assembly
code for the `if` statement is shown in the small box to the right. A load instruction loads the contents of `a[i]`, then it is tested with a branch instruction. The branch instruction has a PC of `X`. Values of the array elements are shown inside the array. Below each array element is the outcome of the dynamic branch that tests it. The misprediction rate of this branch is 42% for both the gshare [33][32][38][55] and L-TAGE [43] branch predictors, and this would contribute to 20% of the mispredictions in gzip.

The way a gshare branch predictor will try to predict this branch is also shown in Figure 6. If the predictor is trying to predict the outcome of the dynamic branch testing `a[4]`, it will hash the PC of the branch (`X`) with the preceding global branch history which is reflected in the global\(^1\) history register (GHR). To simplify the explanation of this example, we assumed that the GHR length is 3-bits.

\(^1\) If we apply the optimization of removing the history bits of extremely biased branches from GHR, the global and local history patterns will be similar.
Figure 6. Code example.

Figure 7 shows the first good scenario where the branch predictor succeeds in predicting the dynamic branch that tests $a[4]$. The GHR value of 001, used to predict the outcome of $a[4]$, is unique across the whole array. This will create a dedicated prediction entry for $a[4]$ in the pattern history table (PHT).
outcome (taken) which would result in no branch mispredictions. Fortunately, both array elements happen to produce the same branch outcome (taken) which would result in no branch mispredictions.

**Good Scenario # 1**
- \(a[4]\) has a unique GHR context
- GHR implicitly identify \(a[4]\)
- Dedicated storage for \(a[4]\)

![Good Scenario # 1](image)

Figure 7. Good scenario # 1.

The second good scenario is shown in Figure 8. \(a[6]\) is predicted using a GHR value of 101. Similarly, \(a[10]\) is predicted using the same GHR value of 101. Both array elements are preceded by the same GHR context which creates a shared prediction entry for both array elements in the PHT. Fortunately, both array elements happen to produce the same branch outcome (taken) which would result in no branch mispredictions.

**Good Scenario # 2**
- \(a[6]\) and \(a[10]\) have the same GHR context
- GHR does not distinguish \(a[6], a[10]\)
- Shared prediction entry for \(a[6], a[10]\)
- Fortunately they have same outcome

![Good Scenario # 2](image)

Figure 8. Good scenario # 2.
The first bad scenario for the same array is shown in Figure 9. Similar to the previous scenario, the predictor tries to predict the branch outcomes of \(a[6]\) and \(a[15]\) using the same GHR context of 101. Both array elements have the same GHR context which means they will share the same entry in the PHT. Unfortunately, these two elements have different branch outcomes, and will keep overwriting each other in the PHT causing branch mispredictions for either of them, if not for both.

<table>
<thead>
<tr>
<th>Load Value</th>
<th>(a[0])</th>
<th>(a[1])</th>
<th>(a[2])</th>
<th>(a[3])</th>
<th>(a[4])</th>
<th>(a[5])</th>
<th>(a[6])</th>
<th>(a[7])</th>
<th>(a[8])</th>
<th>(a[9])</th>
<th>(a[10])</th>
<th>(a[11])</th>
<th>(a[12])</th>
<th>(a[13])</th>
<th>(a[14])</th>
<th>(a[15])</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>11</td>
<td>15</td>
<td>2</td>
<td>33</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>52</td>
<td>9</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>55</td>
<td>8</td>
<td>18</td>
</tr>
</tbody>
</table>

| Branch Outcome | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |

**Bad Scenario # 1**
- \(a[6]\) and \(a[15]\) have the same GHR context
- GHR does not distinguish \(a[6]\), \(a[15]\)
- Shared prediction entry for \(a[6]\), \(a[15]\)
- Unfortunately they have different outcomes

![Figure 9. Bad scenario # 1.](image)

An obvious solution to this problem is to use a longer branch history. Figure 10 shows the same scenario using a GHR length of 4-bits instead of 3-bits. Now, \(a[6]\) will use a GHR context of 0101 while \(a[15]\) will use a GHR context of 1101. This will create dedicated prediction entries for each array element.
Indirect Solution

- Use longer GHR length
- GHR distinguish \(a[6]\), \(a[15]\)
- Dedicated prediction entry for \(a[6]\), \(a[15]\)

Figure 10. Indirect solution for bad scenario # 1.

Longer history lengths have been proven to improve the accuracy of the branch predictor [33], simply because it provides more unique context for predicting branches. However, simply increasing history length does not guarantee unique context. Figure 11 shows the same example of a 4-bit GHR but a different pair of elements is highlighted. Notice both \(a[10]\) and \(a[15]\) are predicted using the same GHR context of 1101. Unfortunately, the two elements have different branch outcomes, and this will cause them to negatively interfere in the PHT, causing mispredictions.

To solve this problem completely, the address of each array element could be used as a unique context to distinguish it from other array elements. Using the address of the array element is equivalent to using the ID of the branch. This will guarantee that no two array
elements will share the same prediction entry. In this way, the branch predictor mirrors the data structure being tested by the branch, since it allocates a dedicated prediction entry per data structure element.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4   11  15  2   33   7   1   3   52   9   3   8   5   55   8   18</td>
</tr>
</tbody>
</table>

| Branch Outcome | 1   0   0   1   0   1   1   1   0   1   1   1   0   1   0 |

**Solution does not always work**

- Use longer GHR length
- GHR distinguish a[6], a[15]
- Dedicated prediction entry for a[6], a[15]
- GHR does not distinguish a[10], a[15]
- Shared prediction entry for a[10], a[15]
- Unfortunately they have different outcomes

Another bad scenario that could occur is shown in Figure 12. In this case, a[4] is predicted using a 3-bit GHR pattern of 001, which is a unique context throughout the entire array. Once the branch predictor is trained, it will correctly predict a[4]. However, a store instruction alters the value of a[4] from 33 to 6 which causes the branch outcome of a[4] to flip from 0 to 1. Unfortunately, the branch predictor will have no idea that this happened: when a[4] is next predicted, the predictor will provide a stale prediction, and it will cost the processor a misprediction in order to retrain the predictor and predict it correctly the next time.

Figure 11. Using longer history does not always work.
The solution for this problem is to allow store instructions to actively update the branch predictor. This will keep the predictor entries coherent with the data structures it is mirroring.

![Diagram](image)

**Bad Scenario # 2**
- Predictor entry is stale after the store
- Predictor entry is retrained after suffering a misprediction

**Figure 12. Bad scenario # 2.**

### 1.3 EXACT: Explicit-Dynamic Branch Prediction with Active Updates

In Chapter 2, we measure the causes of mispredictions using the dynamic-branch framework defined in Section 1.2. We use very large versions of the gselect predictor [33][38][55] and L-TAGE predictor [43] [41](the latter predictor took first place in the most recent championship branch prediction [57]). The study confirms that the two bad scenarios discussed in Section 1.2.2 are the leading causes of mispredictions in large predictors:

1) **Insufficient specialization (bad scenario #1).** Often, global branch history, even very long history (640 bits) as used by some components in L-TAGE, does not distinguish
between two or more dynamic branches. If these dynamic branches have different outcomes, some will be mispredicted because only a single prediction is available to predict all of them.

This problem can be solved by using the branch’s ID to index into the branch predictor. This way, the branch predictor will provide a dedicated entry for each dynamic branch.

Essentially, a branch predictor should mirror a program’s data objects that are tested by branches. Figure 13 contrasts a conventional history-indexed predictor with the proposed ID-indexed predictor, in terms of their ability to mirror data objects. With the former, multiple data objects in memory might share the same branch predictor entry (the green and blue objects in the figure), which will cause mispredictions if their corresponding branch outcomes are different. The proposed ID-indexed predictor explicitly mirrors data objects, ensuring dedicated branch predictor entries for each data object.

Figure 13. First principle: The branch predictor should mirror a program’s data objects. (Best if viewed in color.)
2) Stores (bad scenario #2). A store to an address on which a dynamic branch depends, may cause its outcome to be different the next time it is encountered. In this case, the dynamic branch will be mispredicted because its prediction table entry is stale with respect to the updated data in memory. The entry is only retrained after the misprediction is incurred.

To address this problem, we propose that stores to the addresses on which a dynamic branch depends, directly update its prediction in the predictor. This novel “active update” concept avoids mispredictions that are otherwise incurred by conventional passive training. With passive updates, the branch predictor is retrained after mispredicting. With active updates, the store updates both memory and the branch predictor, avoiding the misprediction by actively mirroring memory. Figure 14 contrasts a conventional predictor with the proposed predictor, in terms of stores to data objects. The former has a stale prediction corresponding to the updated data object: due to the store, the data object’s color changes from red to cyan whereas the predictor entry is still colored red. The latter updates both the data object and the predictor entry, simultaneously: both the data object and predictor entry change color from red to cyan.
Figure 14. Second principle: The branch predictor should *actively* mirror a program’s data objects. (Best if viewed in color.)

We call the proposed predictor EXACT, for “EXplicit dynamic-branch prediction with ACTive updates”. “EX” conveys that dynamic branches are explicitly identified so that they can be provided dedicated predictions and “ACT” conveys that their predictions are actively updated by stores.

We encountered three major implementation challenges while developing the EXACT branch predictor:

1) *Using a dynamic branch’s ID to index into the predictor is difficult to do in practice, because its producer loads are unlikely to have generated their addresses by the time the dynamic branch is fetched.*
2) The most problematic aspect of active updates is converting a store's address into a predictor index (or indices) that must be updated. The complexity and storage cost of address-to-index conversion is potentially prohibitive.

3) Providing a dedicated prediction for each dynamic branch generally requires a large prediction table. The chief problem with a large prediction table is its long access time.

To demonstrate the scale of this last issue (large prediction table), we performed an experiment with an unbounded L-TAGE predictor. The graph in Figure 15 was generated by recording the number of mispredictions caused by each ID (i.e., each unique dynamic branch), sorting IDs by their contributions to total mispredictions, and plotting the cumulative contribution of sorted IDs. For example, 50% of total mispredictions are caused by 13K different IDs in twolf and 51K different IDs in gzip. Thus, many IDs contribute to mispredictions. Even more do not, yet they would contend for an explicit predictor's resources. As a result, in general, the predictor needs to be large to provide substantial coverage of mispredictions.
In this thesis, we propose two different implementations of EXACT: a hardware-only implementation (EXACT-H) [4] and a combined hardware/software implementation (EXACT-S). EXACT-H and EXACT-S are introduced in Sections 1.3.1 and 1.3.2, respectively.

EXACT-H deals with the first implementation issue by using a prior branch’s ID as the index for the current branch, rather than the ID of the current branch itself, which is unavailable. The accuracy of this context is further enhanced with optional global branch history. An unfortunate but necessary side-effect of this indirect indexing strategy is that a branch’s
index into the predictor cannot be deduced from the load address on which it depends. This leads to a complex and storage-intensive active update unit: a large table is needed to convert store addresses to the predictor indices that must be updated. EXACT-H limits the amount of dedicated storage for its address conversion table by virtualizing it (caching it in the general-purpose memory hierarchy), exploiting the key observation that active updates are tolerant of 100s of cycle of latency.

EXACT-S deals with the first implementation issue by enabling the programmer or compiler to convey key information directly to the fetch unit that it can use to generate branches’ IDs in a timely manner. This yields a number of benefits over EXACT-H. First, EXACT-S is more accurate because it uses branches’ IDs directly rather than prior branches’ IDs. Second, its prediction table is more efficient because global history does not need to be added to the context as in the case of EXACT-H, eliminating unwanted redundancy. Third, with regard to active updates, there is no need for a large table to convert store addresses to predictor indices because of the direct indexing strategy. As a result, active updates are simple and inexpensive in EXACT-S.

Regarding the last issue of the predictor’s size and latency, we show that the prediction tables of both EXACT-H and EXACT-S are scalably pipelinable because consecutive indices do not depend on the output of the predictor.
1.3.1 EXACT-H: A Hardware-only Solution

Figure 16 shows a high level view of EXACT-H. Instruction fetch is directed by a hybrid predictor, comprised of a default history-based predictor, an explicit predictor, and a chooser. As branches retire from the processor, their IDs are deduced from producer loads that retired before them (ID Gen) and the IDs are pushed into the Global Branch Queue (GBQ). As alluded to previously, the explicit predictor predicts the current branch using the ID of a retired branch a fixed distance away. This indirect indexing strategy is facilitated by the GBQ (see label 1). The explicit predictor is both passively updated (branches’ outcomes are recorded as they retire, see label 2) and actively updated. When a store retires from the processor, its address and value are converted by an active update unit into updates of the explicit predictor (see label 3).
Indexing Strategy

The ID of a prior retired branch a fixed distance away is a reasonable proxy for the current branch ID because of repetition in the global sequence of dynamic branches. For example, Figure 17(a) shows a sequence of five IDs that repeats. Assuming a distance of 4 is used, ID\(_1\) is a reliable proxy ID for ID\(_5\) regardless of the control-flow outcomes between ID\(_1\) and ID\(_3\).
Figure 17. Two scenarios for using the proxy ID: (a) No control-flow dependence. (b) Control-flow affects which ID is to be predicted by the proxy ID.
Figure 17(b) shows another example where control-flow affects how to use the proxy ID. Assuming a distance of 4 is used, ID$_1$ is alternately the proxy for both ID$_5$ and ID$_6$. Which one is to be predicted by a given instance of ID$_1$ depends on the direction of ID$_4$. Therefore, indexing with only ID$_1$ is unreliable: it would cause ID$_5$ and ID$_6$ to share the same entry in the predictor (leading to the very problem of lack of specialization that we are trying to solve). To solve this problem, we combine the proxy ID with some recent bits from the global branch history. In the example, ID$_1$ can be combined with the most recent bit of global branch history to provide two indices, one for ID$_5$ (if the branch outcome of ID$_4$ is not-taken) and the other for ID$_6$ (if the branch outcome of ID$_4$ is taken).

The chooser of EXACT-H has a mechanism for recognizing the two scenarios in Figure 17. This mechanism is used to learn, for each static branch, whether or not some bits of global history should be included in its index.

**Active Update Implementation**

Two mechanisms are required for active updates: 1) converting the store address into a predictor index to be updated, and 2) converting the store value into a branch outcome.

For a dynamic branch that depends on only a single load (*i.e.*, a single address), address-to-index conversion would be straightforward if the index were based on the branch’s ID: the
index would be the store address combined with the branch’s PC. To obtain the branch’s PC, EXACT-H learns which static stores affect which static branches by training a small store-PC-to-branch-PC conversion table (Figure 16).

Unfortunately, the dynamic branch’s index is based on a proxy ID. This means its index cannot be determined from the store address directly. Instead, a large address-to-index conversion table is required. The table records, for each address, the explicit predictor’s indices corresponding to dynamic branches that depend on that address.

Converting the store value to a branch outcome is achieved by the value-to-outcome conversion table (Figure 16), which is a reuse table that remembers for each static branch what was the branch’s outcome for different load values. The inputs to this table are the branch’s PC (produced by the store-PC-to-branch-PC conversion) and the store value.

The store-PC-to-branch-PC and value-to-outcome conversion tables are small in size since they record static branch information unlike the address-to-index conversion table that records information about dynamic branches.

We can significantly reduce the amount of dedicated storage required for the address-to-index conversion table by virtualizing it [8]. The idea is to implement a small level one (L1) version of the component in dedicated storage, backed by a full version in physical memory which can then be transparently cached in higher levels of the general-purpose memory.
hierarchy (e.g., L2 cache). The advantages include a substantial reduction in dedicated storage, flexible allocation of virtualized active update resources according to application characteristics, and persistence of microarchitectural state.

A potential drawback of virtualization is that it significantly increases the worst-case latency for performing a single active update. Fortunately, latency is not an issue for the active update unit. A key result is that most benchmarks are tolerant of 400+ cycles of latency to perform active updates, due to the long distances between stores and reencounters with branches that they update. This makes the active update unit an ideal candidate for virtualization. Virtualization of the address-to-index conversion table is depicted in Figure 16.

**Pipelining the Explicit Predictor**

The problem with a large explicit predictor is latency. This component is on the critical fetch path and must provide a prediction each clock cycle. A large branch predictor can be pipelined in a straightforward way if its next index does not depend on immediately preceding predictions [25][46]. The explicit predictor can be pipelined because consecutive indices – which are derived from retired branches’ IDs – are independent from the predictor (refer back to Figure 16). While global branch history is optionally included in the index, this is easily accommodated by using an abbreviated index to read out a row of candidate
predictions and post-selecting the finalist at the end of the prediction pipeline when the most recent history bits become available [25][46].

**Summary of EXACT-H Results**

For equal cost, a hybrid gshare+EXACT-H yields 60%, 30%, and 33% fewer mispredictions than gshare alone, for three misprediction-heavy benchmarks: bzip2, gzip, and twolf, respectively. Similarly L-TAGE+EXACT-H yields 60%, 33%, and 14% fewer mispredictions than L-TAGE alone, for bzip2, gzip, and twolf, respectively.

This thesis explores branch prediction with the view of increasing accuracy when artifacts of limited size, limited history, etc., have already been removed. This is the domain of large branch predictors [25][46], and the domain in which new directions in branch predictor research, such as EXACT, are needed to continue scaling accuracy as conventional history will not.

Nevertheless, some designs may favor a small predictor. For example, the leap from an unpipelined or moderately pipelined fetch unit design to a deeply pipelined one may be deemed too great, or the area budget for the fetch unit may preclude a large predictor. Accordingly, we also present a more compact form of the explicit predictor. It takes the form of a small cache, which caches only dynamic instances of a static branch that differ from its overall bias. Attaching this small cache to an existing predictor improves accuracy
comparably to scaling the existing predictor, but without extending the cycle time appreciably. Adding a 4KB explicit predictor cache and 16KB of other overhead (off the critical-path of instruction fetch) removes 33% of mispredictions from a 4KB \( L-TAGE \) and 23% of mispredictions from a 8KB \( L-TAGE \). These results are comparable to the accuracy of doubling the \( L-TAGE \) size, but without extending cycle time.

### 1.3.2 EXACT-S: A Combined Hardware/Software Solution

Figure 18 shows a high-level view of EXACT-S. EXACT-S has the same high-level structure as EXACT-H but is simpler.

![Figure 18. High-level view of EXACT-S.](image)
Indexing Strategy

The indirect indexing strategy of EXACT-H works reasonably well. However, even with the inclusion of bits from the GHR, there is a significant gap in prediction accuracy between using a distant branch ID and using the current branch ID to index into the explicit predictor. Moreover, indirect indexing is inefficient because it creates redundant entries in the explicit predictor, due to the use of global branch history and multiple proxy IDs leading to the same ID.

EXACT-S exploits software intervention to make the indexing strategy both more accurate and more efficient. In most of the applications that benefit from EXACT, many of the data structures tested by branches are arrays. The fetch unit in EXACT-S features a set of software-managed base registers and offset registers (shown in Figure 18) which enable the fetch unit to calculate the addresses of array elements as the dynamic branches that test them are fetched. This way, a dynamic branch’s ID is known when the branch is fetched, so that the explicit predictor can be indexed directly by the current branch ID.

Software (i.e., the programmer or compiler) allocates a single base register and offset register pair for a static branch that tests elements of an array. Software writes the base address of the array into the base register, increments/decrements the offset register, and signals when to index the explicit predictor for a dynamic instance of the static branch. The index is the sum
of the base and offset registers, hashed with the dynamic branch’s PC to form the ID, as shown in Figure 18.

One approach to enable software to manage the fetch unit is to modify the instruction-set architecture (ISA): (1) add new instructions to write the base and offset registers, and (2) add a bit to conditional branch opcodes to signal whether a static branch should use the explicit predictor or default predictor. There are two drawbacks to this approach. First, it requires changing the ISA which may not be an option. Second, this approach will increase the dynamic instruction count of programs modified to use EXACT-S, reducing the performance gain of EXACT-S.

Therefore, we propose managing the fetch unit with *shadow code*. Shadow code is appended to the data segment of the program binary. We use the term “shadow” because the programmer or compiler creates a correspondence between selected instructions in the original program and shadow-instructions in the shadow code. That is, each shadow-instruction *shadows* a particular instruction in the original program. Each shadow-instruction is tagged with the PC of the instruction that it shadows. When the instruction is fetched, the shadow-instruction that shadows it is triggered via the PC.

Typically, key instructions that are shadowed by shadow-instructions include: (1) the instruction that generates the base address of an array, (2) any instruction that is convenient for signaling when to increment the offset and by what positive or negative value, and (3) the
static branch that tests elements of the array. For example, consider a loop that iterates over an array. The instruction prior to the loop that generates the base address of the array is shadowed by a shadow-instruction in the shadow code called the seed shadow-instruction. The seed shadow-instruction writes the base address into a base register in the fetch unit. An arbitrary instruction prior to the loop is shadowed by a shadow-instruction to initialize an offset register. A convenient instruction within the loop is shadowed by a shadow-instruction that increments the offset register by a certain stride. Finally, a branch within the loop that tests elements of the array is shadowed by a shadow-instruction which signals that the branch should be predicted with the explicit predictor, and which base/offset register pair to use for indexing.

The seed shadow-instruction is special in that it involves communication between the general-purpose register file and the fetch unit’s base registers (to write a value into to the base register). The seed shadow-instruction records the physical register number allocated to the destination of the instruction that it shadows. This way, when the instruction executes, its result (the base address) can be obtained from the function unit’s result bus. Moreover, there is a ready bit associated with each base register in the fetch unit. The ready bit is reset when the seed shadow-instruction is initially triggered and set when the shadowed instruction executes. If a branch is fetched that needs to index the explicit predictor, but the base register is not yet ready, then the branch is predicted by the default predictor. To maximize use of the explicit predictor, the programmer or compiler tries to hoist the instruction that generates the base address as early as possible in the program.
Note that EXACT-S does not require a chooser (contrast Figure 18 with Figure 16). The shadow code, and the readiness of base registers, controls selection of the explicit predictor or default predictor.

The current implementation of EXACT-S only supports dynamic branches that depend on a single load (single address in the ID), for two reasons. First, the index function does not currently support combining multiple base/offset register pairs, although it is conceivable to do so. Second, and more importantly, supporting dynamic branches that depend on multiple loads would once again require an address-to-index conversion table in the active update unit since the store address is not sufficient to figure out the affected branch ID. That said, it might be practical to do so: while implementing EXACT-H, we have observed that the address-to-index conversion table for multiple-address dynamic branches does not need to be large.

As a final point, while we focused on arrays in the description above, it is possible to target EXACT-S for stable linked-lists as well, as we discuss in Chapter 5. Essentially, they can be treated as virtual arrays.

**Active Update Implementation**

Since the current branch ID is used to index into the explicit predictor, there is no need for an address-to-index conversion table in the active update unit. As shown in Figure 18, the active
update unit simply hashes the store address with the branch PC to determine the index of the dynamic branch that must be updated. Only the store-PC-to-branch-PC and value-to-outcome tables remain. As an alternative to using hardware to train these tables, they can be pre-loaded with information appended to the data segment by the programmer or compiler.

**Pipelining the Explicit Predictor**

Similar to EXACT-H, the explicit predictor of EXACT-S is scalably pipelinable. If the base register is ready, multiple pipelined accesses to the explicit predictor can be initiated.

Moreover, the indexing strategy of EXACT-S prevents the creation of redundant entries in the explicit predictor: there are no GHR bits in the index and there is no problem of multiple proxy IDs for the same ID. This means the pressure on the explicit predictor will be reduced as compared to EXACT-H, allowing for a smaller explicit predictor for the same effective capacity.

**Summary of EXACT-S Results**

For equal cost, a hybrid *L-TAGE+EXACT-S* predictor yields 44% and 49% fewer mispredictions than *L-TAGE* alone, for gzip and twolf, respectively. These improvements in accuracy yield performance improvements of 21% and 19%, respectively. Moreover, these
results were achieved with small and straightforward shadow codes, fewer than 64 shadowinstructions.
Chapter 2: Characterizing Mispredictions

In this chapter, we characterize mispredictions that escape two global history based branch predictors, *gselect* [33][38][55] and *L-TAGE* [41][43]. The *gselect* predictor has a pattern history table (PHT) of $2^{28}$ entries and the index is formed by concatenating 14 bits of the branch PC with 14 bits of the global branch history register. The *L-TAGE* predictor is composed of 13 predictor components (a simple bimodal component and 12 other partially tagged components) in addition to a loop predictor. Similar to *gselect*, the index for each component is formed by concatenating 14 bits of the PC with a component-specific amount of folded global history. A geometric series is used to determine global history lengths for each component ranging from 4 bits to 640 bits. We used the *L-TAGE* source code provided by the authors [44].

We characterize mispredictions in eleven of the integer SPEC2K benchmarks with reference inputs. The SimPoint toolset [48] was used to locate representative simulation points. The simulator skips to the SimPoint minus 10 million instructions, warms up for 10 million instructions, and then simulates for 100 million instructions.
2.1 Measuring the Ability of History-Based Predictors to Mirror a Program’s Data Objects

This section characterizes how well the large gselect and L-TAGE predictors mirror a program’s data objects that are tested by branch instructions. Mainly, we want to determine how many mispredictions are due to not providing dedicated predictions to data objects that need them. To do so, we include additional information in every prediction entry (all PHT entries of the gselect predictor and all entries in all components of the L-TAGE predictor) purely for diagnosing the root causes of mispredictions. Firstly, every prediction entry is tagged with the ID of the last dynamic branch to have updated the prediction entry (other diagnosis information will be introduced later). As defined in Section 1.2.1, the ID of a dynamic branch is the combination of (1) PC of the dynamic branch and (2) addresses of loads on which the dynamic branch depends, which we will refer to simply as “load addresses”. The current global branch history is effective context for predicting a given dynamic branch, if the indexed prediction entry is tagged with the dynamic branch’s ID.
Figure 19. Breakdown of (a) all branches, (b) just correctly predicted branches, as a percentage of all dynamic branches, and (c) just mispredicted branches, as a percentage of all dynamic branches.
The graphs in Figure 19 show breakdowns of (a) all branches, (b) just correctly predicted branches, as a percentage of all dynamic branches, and (c) just mispredicted branches, as a percentage of all dynamic branches. The percentages in (b) and (c) add up to the percentages in (a). Each bar is broken down into six components. The “no address” component means the dynamic branch either does not depend on any loads or its outcome is not determined solely by loads. From Figure 19(c), the fact that these dynamic branches contribute a relatively small fraction of mispredictions suggests that global history is effective context for specializing predictions for non-load-dependent or non-load-influenced dynamic branches. In Figure 19(a) *Gselect* and *L-TAGE* have equal “no address” components because the “no address” component is a property of the program.

The “entry miss” component corresponds to accessing a prediction entry for the first time (cold miss). The “pc mismatch” component corresponds to accessing an entry that was last updated by a different static branch (conflict miss/aliasing). Both are negligible. The “pc mismatch” component is nearly zero, because aliasing among different static branches has been reduced significantly in both predictors (14 PC bits, preserved through concatenation).

The “address mismatch” component is of primary interest. In this case, the dynamic branch being predicted differs from the dynamic branch which last updated the prediction entry. They are different dynamic instances of the same static branch: their PCs match but their load addresses differ. The mismatch means global branch history fails to distinguish the two
dynamic branches, therefore, the predictor fails to specialize predictions for them. From Figure 19(a), there is an address mismatch in \texttt{gselect} for about 8% (vortex) to 61% (twolf) of all branch predictions and in \textit{L-TAGE} for about 18% (bzip) to 61% (twolf) of all branch predictions. Mcf has about 100% address mismatches, but it has a very low misprediction rate in the simulated region in any case (its SimPoint [48]).

In relation to the four scenarios described in Chapter 1 (Section 1.2.2), the “address mismatch” component in Figure 19(b) corresponds to \textit{good scenario \#2}, where two dynamic branches have the same global branch history context and the same branch outcome. The “address mismatch” component in Figure 19(c) corresponds to \textit{bad scenario \#1}, where two dynamic branches have the same global branch history context but different branch outcomes.

An address mismatch does not necessarily mean the branch will be mispredicted, since different dynamic branches may have the same outcome, only that it is more likely to be mispredicted than if the dynamic branch being predicted were the same as the dynamic branch which last updated the prediction entry. From Figure 19(c), seven of the eleven benchmarks have a misprediction rate higher than 3% in either \texttt{gselect} or \textit{L-TAGE}. For five of these – gcc, gzip, parser, twolf, and vpr – a large majority of mispredictions is attributed to “address mismatch”. Summing up, global branch history does not necessarily distinguish among different dynamic instances of the same static branch, and this is a leading or major contributor to mispredictions in some benchmarks.
2.2 Measuring the Effect of Stores on Modifying Branch Outcomes

The “value mismatch” component in Figure 19 corresponds to the case where the dynamic branch being predicted is the same as the dynamic branch which last updated the prediction entry (their IDs match), but the values at its load addresses were changed by stores since it last updated the prediction entry. To detect this case, each prediction entry is not only tagged with the ID of the dynamic branch that last updated the entry, but also the values contained at its load addresses at that time.

A value mismatch does not necessarily mean the branch will be mispredicted, since different values may lead to the same branch outcome, only that it is more likely to be mispredicted than if the values had not changed. The “match all” and “value mismatch” categories in Figure 19(b) correspond to good scenario #1: global branch history provided unique context for the dynamic branch, and either there was no store (“match all”) or there was a store but it did not change the branch’s outcome (“value mismatch”). The “value mismatch” category in Figure 19(c) corresponds to bad scenario #2: global branch history provided unique context for the dynamic branch, but its prediction entry was stale due to a store to the branch’s load address. From Figure 19(c), the “value mismatch” component is not a major cause of mispredictions, except in the case of bzip2.
On the other hand, mispredictions caused by insufficient specialization (“address mismatch” component) may hide mispredictions that would otherwise be caused by stores (“value mismatch” component). To explore this issue, we use an idealized predictor that explicitly identifies dynamic branches and specializes predictions for them. The predictor is ideal in two ways. First, its size is unbounded. Second, the ID of the dynamic branch is known *a priori* (at the time the branch is predicted). The index is simply the ID of the dynamic branch. The explicit predictor is combined with a global history based predictor (either *gselect* or *L-TAGE*), the latter being used for non-load-dependent or non-load-influenced dynamic branches.
The graph in Figure 20(a) shows the breakdown of mispredictions when the explicit predictor is combined with \textit{gselect}. The graph in Figure 20(b) shows the breakdown of mispredictions when the explicit predictor is combined with \textit{L-TAGE}. There are three bars for each benchmark. For comparison, the first bar ("gselect" or "L-TAGE") shows mispredictions when only the global history based predictor is used (no explicit predictor), which is the same bar from Figure 19(c). The second bar combines the explicit predictor (EX) with the global
history based predictor (“gselect + EX” or “L-TAGE + EX”). The “address mismatch” component of mispredictions is zero because dynamic branches are now provided with dedicated prediction entries. On the other hand, the fraction of mispredictions attributed to “value mismatch” is more substantial now. The results with gselect and L-TAGE show similar trends. Since L-TAGE is more accurate than gselect overall, we focus on the results in Figure 20(b) and only discuss the seven benchmarks that have higher than a 2% misprediction rate. For crafty, gcc, gzip, and twolf, the overall misprediction rate increases compared to just using L-TAGE, due to value mismatches supplanting address mismatches as the chief source of mispredictions. In fact, the “value mismatch” component often exceeds the “address mismatch” component that it replaced. There is also an increase in the “entry miss” component for gcc, gzip, and parser, indicating that a non-trivial fraction of mispredictions are ultimately due to seeing a given dynamic branch ID for the first time; in this study, when there is an “entry miss” in the explicit predictor, the default predictor is used to make the prediction and we do not diagnose the misprediction other than to indicate that it was produced by the default predictor. For vpr, entry misses supplant address mismatches almost one-for-one. In its SimPoint, vpr pair-wise compares elements from two large arrays, creating many dynamic branch IDs that are visited only once during the SimPoint. For bzip, the “value mismatch” component was the chief source of mispredictions originally (“L-TAGE”) and its contribution doubles (“L-TAGE + EX”). The third bar in Figure 20 (a) and (b) augments the explicit predictor with active updates by store instructions, using the active-update implementation described in Section 3.7 of Chapter 3 (“gselect + EXACT” and “L-TAGE + EXACT”). Explicit dynamic-branch prediction and active updates work in concert
to substantially reduce the misprediction rate with respect to \textit{gselect} and \textit{L-TAGE}. From Figure 20(b), bzip, crafty, gzip, and twolf, the four benchmarks which experienced the largest increases in “value mismatch” type mispredictions, have many of these mispredictions eliminated by active updates, for a substantial decrease in the overall misprediction rate with respect to \textit{L-TAGE}.

Overall, Figure 20 shows the potential for dramatic reductions in misprediction rates using the two principles of EXACT: explicit dynamic-branch prediction (for achieving desired specialization) and active updates. Notably, except for bzip, the two techniques are needed in combination: the first technique is needed to eliminate mispredictions caused by insufficient specialization, but in doing so, the predictor is also more vulnerable to stores that require active updates. Bzip only requires active updates.
Chapter 3: The Microarchitecture of EXACT-H

Figure 21 shows the major components of the EXACT-H predictor. A prediction is supplied by either the default predictor (e.g., L-TAGE) or the explicit predictor. The explicit predictor is simply a table of 1-bit predictions. The chooser classifies static branches as more suitable for the default predictor or the explicit predictor. The chooser also singles out static branches that exhibit loop behavior and directs an explicit loop predictor to provide a trip-count to the fetch unit instead of single prediction.

All components are passively trained as dynamic branches retire from the processor. In addition, both the explicit predictor and the explicit loop predictor may be actively updated by the active update unit. An active update occurs when a store retires from the processor. The ID generation unit observes all instructions as they retire from the processor, in order to propagate load addresses (the basis for IDs) to branches. When a branch retires, its ID is
pushed onto the global branch queue (GBQ). The GBQ is used for indexing the explicit predictor and explicit loop predictor.

Sections 3.1 through 3.4 explain ID generation, the GBQ, indexing the explicit predictor, and pipelining the explicit predictor. The explicit loop predictor, chooser, and active update unit are explained in Sections 3.5 through 3.7.

### 3.1 ID Generation Unit

A non-functional architectural register file (ARF) propagates addresses of loads to branches that depend on them directly or indirectly. In this work, each logical register in the ARF holds up to four load addresses. Loads write their addresses into their destination registers when they retire from the load queue. ALU instructions propagate addresses from their source registers to their destination registers when they retire from the reorder buffer. Branch instructions obtain their load addresses from their source registers when they retire from the reorder buffer.

To handle registers that are spilled to the stack, we augment the ARF with a small fully-associative cache, called a stack cache. We use a 32-entry stack cache in this work. Like the ARF, a stack cache entry contains up to four addresses but it is also tagged with a stack address to check for stack-store and stack-load hits. A stack-store copies the addresses contained in its source register from the ARF to the stack cache. If a stack-load hits in the stack cache, the addresses are copied from the stack cache to the stack-load’s destination
register in the ARF. In summary, load addresses are propagated to branches through both registers and the stack.

```c
hash = 0;
for (i = 0; i < num_addresses; i++)
    hash = hash ^ (address[i] << i);
hash = hash & 0xFFFFF;  // this mask corresponds to N=20
branch_ID = hash ^ (((PC >> 3) & 0xFF) << 12);
```

Figure 22. Hash function for generating a dynamic branch’s ID.

A dynamic branch forms its ID by hashing its PC and load addresses together, as follows. The first address is XORed with the second address shifted left by one bit, the third address shifted left by two bits, and so on, for as many load addresses as there are. The result is then ANDed with a mask to extract the low N bits, for an explicit predictor that has $2^N$ entries. The upper 8 bits of the result is XORed with the lower 8 bits of the PC. Figure 22 describes the hash function in C code, for N=20.

### 3.2 Global Branch Queue

The GBQ contains IDs of recently retired dynamic branches. When a dynamic branch retires, the ID generation unit pushes its ID onto the GBQ, displacing the oldest ID in the GBQ. GBQ length is discussed in the next section.

### 3.3 Indexing the Explicit Predictor

We cannot use the ID of a dynamic branch to index the explicit predictor for two reasons. First, typically, its producer loads have not generated their addresses by the time it is fetched. Second, even if addresses were available in time to predict the branch, assembling and
associating them with the branch currently being fetched is challenging, whereas the ID generation unit does this straightforwardly when the branch itself retires.

Instead, the index for a branch is based on the ID of a prior retired branch some fixed distance away. This distance determines the length of the GBQ. The approach is illustrated in Figure 23 for three scenarios and a distance of 20. The three scenarios differ in how many unretired branches are currently in the processor pipeline, affecting which branch in the GBQ is used to predict the new branch. The third scenario shows what happens when the distance between the new branch and the youngest retired branch in the GBQ is greater than the fixed distance, 20. The problem is that the new branch needs the ID of a branch which has not yet retired. It cannot form an index into the explicit predictor and must use the default predictor instead.

**Figure 23.** The index for a branch is based on a prior branch’s ID some fixed distance away (20 in this example).
This indexing strategy is tantamount to predicting the ID of the current branch from the ID of a distant prior branch. A given ID may lead to any of a number of IDs downstream from it, depending on intervening control-flow (among other things). This is corroborated by our studies which show that hashing the ID of the distant prior branch with global branch history is essential for more closely approximating using the ID of the current branch. We have observed a few exceptions to this general rule. In these exceptional cases, including global branch history may be detrimental because it creates redundant entries in the explicit predictor which has two negative effects: thrashing the predictor and needlessly increases training time. We concluded that global branch history should be used for some branches and not others. To this end, the chooser – in addition to selecting among the default predictor, explicit predictor, and explicit loop predictor – identifies branches whose indices into the explicit predictor should not include global branch history.

When global branch history is included in the index, it is hashed into the low bits of the ID of the distant prior branch. Figure 24 describes the index function in C code, for a distance of 20 and 6 branch history bits.

```c
// 1. ID_20 is the ID of the branch that is 20 branches away
// 2. BHR is the global branch history register
// 3. use_history is from the chooser
index = (use_history ? (ID_20 ^ (BHR & 0x3F)) : (ID_20));
```

**Figure 24. Example index function, described in C code.**

Figure 25 shows the effect of distance, for two cases: (a) the prior branch whose ID is used for the index is assumed to be retired regardless of distance (*i.e.*, never suffer scenario #3),
and (b) whether or not the prior branch is retired is determined through cycle-level processor simulation (may suffer scenario #3).

![Diagram](a)

![Diagram](b)

**Figure 25.** Effect of distance. (a) Prior branch is always retired. (b) Retired status based on processor simulation.

A distance of 0 means the branch being predicted uses its own ID. For case (a), this yields the lowest misprediction rate but is based on the flawed assumption that its ID is available. Case (b) shows the highest misprediction rate for distance 0 because the default predictor is used
almost exclusively. Case (a) shows increasing misprediction rate with distance, with gzip and
twolf showing large jumps between distance 0 and 1. This transition is effectively the gap
between ideal and real index prediction. Subsequently, misprediction rate increases gradually
with increasing distance. Case (b) shows decreasing misprediction rate with distance since
increasing distance increases the number of branches predicted by the explicit predictor.
Cases (a) and (b) converge in the low to mid 20s for gzip and twolf.

The rationale behind using the branch ID at a fixed distance as the index, is repetition in the
dynamic branch stream. Changes in intervening control-flow may cause different streams.
Using recent global branch history in the index accounts for this.

3.4 Pipelining the Explicit Predictor

The explicit predictor is straightforwardly pipelainable because consecutive indices are
independent of pending accesses. This is illustrated in Figure 26 for a three-cycle prediction
latency, for example.
Figure 26. Pipelining the explicit predictor. Diagram shows three consecutive accesses to a three-cycle-latency predictor.

The indices for the first, second, and third accesses are derived from a first ID (ID1), a second ID (ID2), and a third ID (ID3) in the GBQ. Thus, the second access may begin in the second cycle even though the first access is pending, and the third access may begin in the third cycle even though the first and second accesses are pending. Global branch history may be included in the index. The most recent B bits of global history (two bits in our example) are unavailable, however, because they have yet to be produced by preceding pending accesses. This is easily handled by using an abbreviated index that excludes the B unknown bits, reading out a row of $2^B$ candidate predictions, and post-selecting one prediction from among the candidate predictions using the late-arriving B bits at the end of the prediction pipeline [25]. In the example of Figure 26, notice that two bits of both the ID and BHR are omitted from the index, a row of four predictions is read out, and then the missing two index
bits are generated at the end of the prediction pipeline (by which time the preceding pending
accesses have produced their predictions), which control the MUX.

### 3.5 Explicit Loop Predictor

Some loops have trip-counts that depend on one or more loads preceding the loop. An
elementary example of such a loop from twolf is shown in Figure 27 and explained in the figure caption.
Applying the same static vs. dynamic framework defined in Chapter 1, Section 1.2.1, we say
that a static trip-count translates into multiple dynamic trip-counts at run-time. Like a
dynamic branch, a dynamic trip-count has an identity (ID) as a whole: the PC of the loop
branch combined with the addresses of loads that the dynamic trip-count depends on.
Different dynamic trip-counts are distinguished by their IDs. The role of the explicit loop
predictor is to provide specialized trip-count predictions to different dynamic trip-counts,
analogous to the role of the explicit predictor for dynamic branches.

```c
ucxx1(...) {
  ...
  int axcenter, aleft, aright, a1LoBin, a1HiBin;
  ...
  axcenter = acellptr->cxcenter;               // heap load
  aleft = atileptr->left;                      // heap load
  aright = atileptr->right;                    // heap load
  ...
  a1LoBin = SetBin( startxa1 = axcenter + aleft ); // SetBin() is a simple macro
  a1HiBin = SetBin( endxa1 = axcenter + aright );
  ...
  sub_penal( ... , a1LoBin, a1HiBin );
  ...
}
```

```c
sub_penal( ... , int LoBin , int HiBin ) {
  ...
  for ( bin = LoBin ; bin <= HiBin ; bin++ ) {  // example for-loop
    ...
  }
}
```

Figure 27. Example for-loop in the sub_penal() function in twolf. The trip-count of the loop branch is
dynamic and transitively depends on three heap loads in the caller function ucxx1(). The transitive
dependence is shown with boldface type.
The explicit loop predictor is accessed using the same index as the explicit predictor (Section 3.3). The explicit loop predictor is managed as a set-associative cache, however, so it can have fewer entries yet still use the full-length indices. The low bits of the index select a set. Entries within the set are tagged with the remaining high bits of the index. An entry is shown in Figure 28.

![Figure 28. A single entry in the explicit loop predictor.](image)

Dynamic trip-counts are identified as follows. When a backward branch is retired, successive instances of the branch are examined for signature behavior: (1) the branch is taken multiple times in a row followed by a not-taken outcome, thus determining the trip-count value, and (2) all of these dynamic instances of the branch have the same ID. They inherit the same ID because they all test a load-dependent trip-count that was preset outside the loop. It is interesting to note that, without the explicit loop predictor, the explicit predictor alone would be incapable of predicting the dynamic instance that exits the loop since all instances have the same ID (no specialization). Encoding the trip-count (i.e., bundling all iterations into a single prediction) not only provides the immediate benefit of predicting the exit like other loop predictors [14], but also some unprecedented benefits such as (1) explicitly specializing the trip-count for different dynamic data structures and (2) actively-updating the trip-count when these data structures are modified by stores.
3.6 Chooser

The chooser has three jobs: 1) identify branches that are best predicted by the default predictor (§3.6.1), 2) identify branches that exhibit dynamic trip-counts, hence, are best predicted by the explicit loop predictor if it hits or the default predictor otherwise (§3.6.2), and 3) identify branches that do not need global branch history included in their indices (§3.6.3).

The chooser is PC-indexed and is not tagged. Thus, classification is on a per-static-branch basis. This is not only efficient in terms of storage but it also enables rapid classification since all dynamic instances of a static branch train its entry.

3.6.1 Choosing the Default Predictor

Training for this decision occurs in two phases. The first phase is a brief warm-up period during which both the explicit predictor and default predictor are trained for all branches. A simple optimization is that non-load-dependent branches (ID=PC) never train the explicit predictor; Chapter 2, Section 2.1 showed that these are predicted just fine by the default predictor as one would expect. The fetch unit uses predictions solely from the default predictor during the first phase since the chooser is not yet trained. In the second phase, the fetch unit switches to using predictions from the explicit predictor (except for non-load-dependent branches). Meanwhile, training of the chooser proceeds by comparing the ability of the two predictors to predict branches. A chooser entry has a saturating counter and a sticky bit for this purpose. The counter is incremented when the default and explicit predictors are correct and incorrect, respectively; decremented when the default and explicit
predictors are incorrect and correct, respectively; and unchanged when both predictors are correct or incorrect. If the counter saturates, the sticky bit is set. A sticky bit of 1 signifies that the default predictor should be used for all dynamic instances of the static branch. Crucially, dynamic instances of this static branch no longer participate in training the explicit predictor or active update unit, dedicating these important resources to more suitable branches.

Certain load-dependent branches are best left for the default predictor for two reasons. First, the indexing strategy may not be accurate for these branches. Recall that this strategy is tantamount to inferring the current branch’s ID from a previous branch’s ID, a form of address prediction. This may be inaccurate for some branches and they should be weeded out. Second, there may be thrashing in the explicit predictor: explicitly mirroring memory is a worthy cause but requires sufficient predictor capacity, and off-loading some branches to the default predictor may be necessary.

3.6.2 Choosing the Explicit Loop Predictor

Section 3.5 explained how a dynamic trip-count is detected at retirement and how this causes it to be cached in the explicit loop predictor. At the same time, using the PC of the corresponding branch to index the chooser, a sticky bit is set to 1 to indicate that this branch should use the explicit loop predictor if it hits and the default predictor otherwise. If it misses in the explicit loop predictor, the explicit predictor is no better than the default predictor (and maybe worse) since all dynamic instances of the branch have the same ID and predictions cannot be specialized for them individually (discussed at length in Section 3.5).
3.6.3 Declining Global Branch History

The rationale for including global branch history in the index is that a given ID may lead to any of a number of different IDs downstream from it, depending on intervening control-flow. It is unnecessary, however, if a given ID always leads to the same ID downstream from it. We add a small set-associative cache to detect one scenario or the other (different IDs vs. same ID downstream). It is indexed by ID and the entry contains the previously observed downstream ID. The next time the ID is encountered, if its downstream ID differs from the previously observed downstream ID, then a counter in the chooser (that of the downstream branch’s PC) is incremented, otherwise, it is decremented. If the counter saturates at its maximum value, a sticky bit is set indicating to always use global history in the index for the downstream branch. If the counter saturates at its minimum value, a different sticky bit is set indicating to always omit global history from the index of the downstream branch.

3.6.4 Summary of Chooser Contents

Figure 29 summarizes the contents of the two structures in the overall chooser unit. Fields are also annotated with the relevant sections where they were explained.

<table>
<thead>
<tr>
<th>Chooser Entry</th>
<th>Sampled Downstream ID Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1 SB1 SB2 C2 SB3 SB4</td>
<td>valid bit tag prev. downstream ID</td>
</tr>
</tbody>
</table>

C: counter
SB: sticky bit

Figure 29. Summary of chooser contents.
3.7 Active Update Unit

Implementing active updates requires two mechanisms. The first mechanism determines which dynamic branches in the explicit predictor and loop predictor are affected by the store. This process is called *store address conversion* because the store address is converted into affected predictor indices. The second mechanism determines the impact that the store value will have on these dynamic branches in the future. This process is called *store value conversion*, i.e., converting the value into branch outcomes.

3.7.1 Store Address Conversion

Branches that depend on a single load and branches that depend on multiple loads use different structures for store address conversion, to optimize total cost. Store address conversion – and how it interacts with store value conversion and the predictor – is shown in Figure 30 (a) and (b) for single-address and multiple-address branches, respectively.
For single-address branches (Figure 30a), the Single-Address Conversion Table (SACT) outputs two pieces of information for each dynamic branch that depends on the address being stored to: (1) the branch’s index in the predictor and (2) the PC of the branch. Store value conversion is based on a novel form of a reuse table [50] which produces a new branch outcome or trip-count using only the branch’s PC and the store value. The only value needed is that of the store because the branch depends on only one load. The predictor can now be actively updated using the branch’s index and new outcome or trip-count.

---

Figure 30. Store address conversion, and its interaction with store value conversion and the predictor.
The picture is almost identical for multiple-address branches (Figure 30b), with a key difference. The store value alone is not enough to infer the new branch outcome or trip-count because the branch also depends on the current values at its other addresses (those not being stored to). Thus, the Multiple-Address Conversion Table (MACT), actually composed of two structures MACT-A and MACT-B, outputs two additional pieces of information: (1) the values at all addresses on which the branch depends, called the value-combo, and (2) which value in the value-combo is being updated by the store, called the position. We have determined that the low 16 bits of values suffice. Thus, the value-combo is the concatenation of up to four 16-bit values. The old value-combo is updated by the store and this new value-combo as a whole is the input to store value conversion.

The SACT and MACT are passively trained as dynamic branches retire, by the ID generation unit. Training value-combos requires that the ARF and stack cache not only propagate loads’ addresses, but the low 16 bits of their values as well. Also, when a retired store performs an active update, a side-effect is updating the value-combo in the MACT if it exists.
3.7.1.1 SACT
The SACT is a set-associative cache indexed by the store address. The payload of a SACT entry, shown in Figure 31, is a list of \{PC, index\} tuples, one for each dynamic branch that depends on the address.

3.7.1.2 MACT-A and MACT-B
MACT-A is a set-associative cache indexed by the store address. A dynamic branch occupies multiple entries in MACT-A, one for each of the addresses it depends on, so that stores to any of the addresses can trigger an active update. The payload of a MACT-A entry, shown in Figure 31, consists of (1) the hash of the branch’s addresses (addr-hash), \textit{i.e.}, the branch’s ID\(^2\) excluding its PC, and (2) the position of the address (which one of up to four addresses) when it was hashed into the ID. The multiple entries occupied by the branch, corresponding to its multiple addresses, will all have the same addr-hash but a unique position. The addr-

\(^2\) Although more than one dynamic branch may depend on the address, accommodating only one suffices.
hash is used to access MACT-B whose payload consists of the other three required items, (1) the branch’s PC, (2) the branch’s index, and (3) the value-combo, as shown in Figure 31. Thus, all of the branch’s MACT-A entries point to the same MACT-B entry for these three items. This reduces cost. Also, by keeping only a single copy of the value-combo in MACT-B instead of replicating it in MACT-A, MACT-A entries perceive each other’s updates to the value-combo, which helps if there are multiple stores to different addresses before the next instance of the branch is encountered. This is simply an issue of accuracy and not correctness, however. If one of the branch’s MACT-A entries is evicted and there is a store to the evicted entry followed by a store to a resident entry before the branch is reencountered, a partially stale value-combo (first store not included) will be used for the second store’s active update. Again, this is only an issue of accuracy. Moreover, flaws are fleeting because MACT-A and MACT-B entries are refreshed by passive updates.

### 3.7.2 Store Value Conversion

The store value (for single-address branches) or store-updated value-combo (for multiple-address branches) is converted to a branch outcome (non-loop branch) or trip-count (loop branch) using novel reuse tables. Both the General Reuse Table (GRT) and Range Reuse Table (RRT) are set-associative caches indexed by the branch’s PC.

#### 3.7.2.1 GRT

A GRT entry records the outcome or trip-count that was observed for a given value or value-combo. Each entry can enumerate up to 16 \{value/value-combo, outcome/trip-count\} pairs, as shown in Figure 32. The reuse test requires a match on the PC and value or value-combo.
The GRT is general in that it can be used by any type of branch: non-loop or loop, single-address or dual-address or multiple-address.

![Figure 32. GRT and RRT entries.](image)

### 3.7.2.2 RRT

The RRT is narrower in scope yet powerful. It can only produce taken/not-taken outcomes (non-loop branches only) and is only intended for single-address and dual-address branches.

An RRT entry trains two value ranges: one range that produces a taken outcome and one range that produces a not-taken outcome. Using ranges has two advantages over enumerating distinct values. First, only four values need to be recorded for each static branch (maximum and minimum for taken outcomes and maximum and minimum for not-taken outcomes) instead of recording every distinct value seen by the branch, as shown in Figure 32. Second, an active update can be performed even for a value that has not been seen by the branch, if it falls within either the taken or not-taken ranges.

For single-address branches, the RRT reuse test checks for a match on the PC and for the store value to fall within one of the ranges. For dual-address branches, the two 16-bit values in its store-updated value-combo are subtracted and the reuse test checks for this result to fall
within one of the ranges. The intuition behind this heuristic is that, often, this type of branch compares two load-dependent values.

If, while training an RRT entry, one range is updated such that it overlaps the other range, the static branch is evicted from the RRT and added to the GRT. Its existence in the GRT prevents its further use of the RRT. This action adapts to the reality that using ranges for the branch is probably unreliable.
Chapter 4: Evaluating EXACT-H

4.1 Methodology

All results in this thesis are based on custom predictor and processor simulators derived from the SimpleScalar toolset [9]. Eleven of the integer SPEC2K benchmarks were used with reference inputs. We compiled these benchmarks to the SimpleScalar PISA instruction set using the SimpleScalar gcc-based compiler with –O3 optimization. The eon benchmark did not compile. The SimPoint toolset [48] was used to locate representative simulation points. The simulator skips to the SimPoint minus 10 million instructions, warms up for 10 million instructions, and then simulates for 100 million instructions. Microarchitecture parameters of the modeled processor are shown Table 2.

<table>
<thead>
<tr>
<th>Table 2. Microarchitecture parameters for EXACT-H.</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 I&amp;D Caches</td>
</tr>
<tr>
<td>L2 Cache</td>
</tr>
<tr>
<td>Reorder Buffer</td>
</tr>
<tr>
<td>Issue Queue</td>
</tr>
<tr>
<td>Load-Store Queue</td>
</tr>
<tr>
<td>Rename Map Checkpoints</td>
</tr>
<tr>
<td>Fetch-to-exec. pipe depth</td>
</tr>
<tr>
<td>Fetch/Issue/Retire Width</td>
</tr>
</tbody>
</table>

All results are presented in the context of cycle-level processor simulation to model the effect of branch distance in forming the predictor’s index. Throughout, the GBQ length is 21, therefore, the index is formed using the ID of the 21st branch away. If this branch is not yet
retired, the default predictor is used. The index includes either 0 or 6 bits of global branch history: the chooser guides this decision on a per-static branch basis. A warm-up period of 10 million instructions is used, after which the chooser commences training and hybrid prediction is engaged.

We divide EXACT-H’s subcomponents into two classes. Subcomponents in the first class can be scaled up in size with continuing effect on reducing misprediction rate, therefore, these are studied in Section 4.4. They include: (1) the default predictor, (2) the explicit

Table 3. Fixed-configuration subcomponents for EXACT-H. Not included: default predictor, explicit predictor, and SACT.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Structure</th>
<th># of entries / organization</th>
<th>Contents per entry</th>
<th>Size (K-Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID Generation Unit</td>
<td>ARF</td>
<td>67 entries</td>
<td>4 valid bits + 4 20-bit addresses + 4 16-bit values</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td>stack cache</td>
<td>52 entries / fully-assoc. cache</td>
<td>4 valid bits + 5-bit LRU + 20-bit tag + 4 valid bits + 4 20-bit addresses + 4 16-bit values</td>
<td>0.68</td>
</tr>
<tr>
<td>GBQ</td>
<td>GBQ</td>
<td>21 entries</td>
<td>N-bit ID</td>
<td>0.04 – 0.06</td>
</tr>
<tr>
<td>Explicit Loop Predictor</td>
<td>Explicit Loop Predictor</td>
<td>256 entries / 8-way set-assoc.</td>
<td>4 valid bit + 3-bit LRU + (N-5)-bit tag + 8-bit trip-count</td>
<td>0.75 – 0.94</td>
</tr>
<tr>
<td>Chooser</td>
<td>Chooser Table</td>
<td>1024 entries / not tagged</td>
<td>5-bit C1 + 1-bit SB1 + 1-bit SB2 + 9-bit C2 + 1-bit SB3 + 1-bit SB4</td>
<td>2.25</td>
</tr>
<tr>
<td></td>
<td>downstream ID</td>
<td>512 entries / 8-way set-assoc.</td>
<td>4 valid bit + (N-6)-bit tag + N-bit prev. downstream ID</td>
<td>1.69 – 2.44</td>
</tr>
<tr>
<td>Active Update Unit</td>
<td>RRT</td>
<td>128 entries / 4-way set-assoc.</td>
<td>4 valid bit + 2-bit LRU + 9-bit tag + 4x16-bit for (min_NT, max_NT, min_T, max_T)</td>
<td>1.19</td>
</tr>
<tr>
<td></td>
<td>GRT</td>
<td>52 entries / 4-way set-assoc.</td>
<td>4 valid bit + 2-bit LRU + 11-bit tag + 16x64-bit values + 16x8-bit trip-count</td>
<td>4.55</td>
</tr>
<tr>
<td></td>
<td>MACT-A</td>
<td>512 entries / 16-way set-assoc.</td>
<td>4 valid bit + 4-bit LRU + 15-bit tag + 2-bit position + 20-bit addr-hash</td>
<td>2.63</td>
</tr>
<tr>
<td></td>
<td>MACT-B</td>
<td>256 entries / 16-way set-assoc.</td>
<td>4 valid bit + 4-bit LRU + 16-bit tag + 14-bit PC + N-bit index + 64-bit value-combo</td>
<td>3.59 – 3.78</td>
</tr>
</tbody>
</table>

3 Notes:

1. Regarding number of entries in ARF: PISA ISA has 32 integer registers, 32 floating-point registers, and 3 other registers (hi, lo, fcc).
2. N = # bits in the ID = # bits in the index. N is varied between 16 and 22. This introduces variation in the total cost of the fixed-configuration subcomponents, as shown.
3. All addresses are truncated to 20 bits after discarding the low 1, 2, or 3 bits for halfwords, words, and doublewords, respectively. Likewise, the addr-hash is 20 bits. When N>20, the upper few bits of the ID consists of only PC bits.
4. All PCs are truncated to 14 bits after discarding the low 3 bits (PISA instructions are 8 bytes).

Total cost of fixed subcomponents (does not include default predictor, explicit predictor, and SACT): 18.6 – 19.7
predictor, and (3) the SACT component of the active update unit. Subcomponents in the second class are no less critical but their sizes are not as closely coupled to a program’s large data structures, hence, scaling them up has negligible effect. The configurations of subcomponents in the second class are fixed and shown in Table 3. The configuration of each subcomponent was arrived at by unconstraining all other subcomponents and finding the point of diminishing returns. The total cost of these fixed subcomponents is 18.6 – 19.7 KB, depending on N (the number of bits in the ID and index).

4.2 Impact of Real Indexing

In this section, we measure the impact of real indexing in the context of unconstrained resources: the unconstrained gselect and L-TAGE predictors of Chapter 2 and unconstrained EXACT-H subcomponents. Figure 33 shows misprediction rates of each default predictor alone (gselect or L-TAGE), each default predictor with ideal-indexed EXACT-H, and each default predictor with real-indexed EXACT-H. Ideal indexing is where the ID of the branch is used. Real indexing is where the ID of the 21st prior branch is used (plus history if chosen) if it has been retired.

Figure 33. Misprediction rates with large predictors.
From Figure 33 and earlier characterizations from Chapter 2, we classify the eleven benchmarks into three categories. The first category consists of benchmarks that do not benefit from EXACT-H, either because they already have impressive accuracy (gap, mcf, perlbmk, vortex), particularly with L-TAGE, or there is no repetition of dynamic branches / IDs (vpr). For these benchmarks, EXACT-H does not improve or appreciably degrade the accuracy for both ideal and real indexing. The second category consists of benchmarks that have moderate misprediction rates, show some improvement with ideal indexing, but very slightly improve or degrade with real indexing (crafty, gcc, parser). The third category consists of benchmarks that show an impressive reduction in misprediction rate with ideal indexing and attain a considerable portion of this reduction with real indexing (bzip, gzip, twolf). On average, EXACT-H with ideal indexing removes 44% and 37% of mispredictions in gselect and L-TAGE, respectively, while EXACT-H with real indexing removes 26% and 20% of mispredictions in gselect and L-TAGE, respectively.

4.3 Impact of Active Update Latency

In this section, we measure the impact of active update latency in the context of the unconstrained gselect+EXACT-H(real-indexing) from the previous section. Figure 34 shows misprediction rate as a function of active update latency in increments of 50 cycles. The latency is from the cycle that the store retires to the cycle when the updated outcome or trip-count is reflected in the explicit predictor or loop predictor. The three benchmarks that benefit substantially from EXACT-H are included as well as crafty which mildly benefits. Gzip is insensitive to latency. Twolf is very tolerant of latency up to 400 or 500 cycles. The
Bzip2 is very sensitive to active update latency. This behavior stems from a highly mispredicted branch inside a loop that depends on a single global variable that is stored to every iteration. The store and the next instance of the branch are close enough that the store is not retired before the branch is fetched. Without special intervention, the store’s active updates are always a step behind fetching the branch. We implemented an ideal intervention to confirm that the active update is otherwise effective: the store’s active update is performed as soon as its value is available and it overrides the branch’s original prediction if it differs, resolving the misprediction early. The resulting accuracy is reflected in the 0-cycle data point for bzip2 (the 0-cycle data points for the other benchmarks are from the cycle that the store is retired, as usual). While we present results for bzip2 in subsequent sections with this ideal
intervention, the reader should keep in mind that bzip2 requires a mechanism for detecting and carrying out in-window active updates which we have not devised.

4.4 Accuracy vs. Storage Budget

This section demonstrates results for large predictors used in leading-edge processing cores, where the issue of predictor access latency is relieved by means of pipelining. EXACT-H is a good solution when scaling the conventional branch predictor does not provide any accuracy improvement.

The graphs in Figure 35 show misprediction rate as a function of cost using the methodology described in Section 4.1: the subcomponents in Table 3 have fixed configurations with a total cost around 19KB and only the default predictor, explicit predictor, and SACT configurations are varied. For each cost point, resources are allocated to these three subcomponents based on design space exploration that yields the lowest misprediction rate. With cost constrained, we substitute \textit{gselect} with \textit{gshare} [33]. The \textit{gshare} index for each cost point (\textit{e.g.}, length of global branch history register) is also based on exploration.

We explore two implementations of the SACT:

1. The first uses dedicated storage. In this case, the SACT is included in the design space exploration (\textit{i.e.}, its size is varied) and its entire cost is included in the cost comparison. In Figure 35, grey points correspond to the SACT being implemented in dedicated storage (labeled \textit{gshare+EXACT-H} or \textit{L-TAGE+EXACT-H}).
2. The second uses a small, dedicated L1 SACT combined with a larger, virtualized L2 SACT, applying the concept proposed by Burcea et al.[8]. Both are fixed in size. The L1 SACT consumes 10KB of dedicated storage, which is added to the original fixed cost of 19KB for an adjusted fixed cost of 29KB. The L2 SACT is pinned in physical memory [8]. A whole SACT set is 80 bytes, which fits entirely within a single L2 cache block. Thus we align SACT sets at block boundaries. We fix the size of the L2 SACT to be 512 KB. This cost is not included in the cost comparison, but the virtualized L2 SACT contends with the application’s instructions and data for L2 cache resources, which is relevant for performance results presented in Section 4.5. In Figure 35, black points correspond to the virtualized SACT (labeled as \texttt{gshare+EXACT-H(VIRT. SACT)} or \texttt{L-TAGE+EXACT-H(VIRT. SACT)}).
Figure 35. Misprediction rate versus cost. (Note the log-scale x-axis and the y-axis does not start at 0%).
4.4.1 Dedicated Storage for SACT

The first cost point where $L$-$TAGE+EXACT$-$H$ overtakes $L$-$TAGE$ is 40KB for bzip2 (1.8% down from 3.8%), 353KB for gzip (6.7% down from 7.2%), and 155KB for twolf (7.1% down from 7.3%). Gzip’s sharp corner is due to its working set fitting in the SACT, which is the component that expands the most as more resources become available. Also notable about gzip is that $gshare+EXACT$-$H$ overtakes $L$-$TAGE$ at the 525KB cost point (6.3% down from 7.2%). At a close cost point of 533KB, $L$-$TAGE+EXACT$-$H$ reaches 5.4%, nearly 2 points lower than $L$-$TAGE$. Bzip2 and gzip are perfect examples in which scaling the conventional branch predictor ($gshare$ or $L$-$TAGE$) does not yield any accuracy improvement, whereas EXACT-H can easily yield benefits from scaling resources. For twolf, $L$-$TAGE$ is able to capitalize on more resources for a fairly broad cost range. So does $L$-$TAGE+EXACT$-$H$, improving $L$-$TAGE$’s misprediction rate by 0.2 points at 155KB to 0.5 points at 667KB, with the gap widening further at subsequent cost points.

4.4.2 Virtualized SACT

In Figure 35, the curves qualified with the label “(VIRT. SACT)” correspond to a virtualized L2 SACT. For bzip2, the virtualized implementation is slightly costlier than the dedicated one, since the L1 SACT is actually larger than needed (curve is slightly shifted to the right). Gzip benefits significantly from a large, virtualized L2 SACT: it sheds 200-300KB of dedicated storage for the same accuracy. For twolf, the virtualized L2 SACT only moderately improves its accuracy at a given cost point.
4.5 Explicit Predictor Cache (EX-cache)

Some designs may favor a small predictor, for example, the leap from an unipipelined or moderately pipelined fetch unit to a deeply pipelined one may be deemed too great, or the area budget for the fetch unit may preclude a large predictor. We present a more compact form of the explicit predictor to target the domain of smaller predictors.

First, the following features are simplified or removed: (1) the ARF only propagates a single address and value; (2) the stack cache is removed (no dependency-propagation through the stack); (3) the explicit loop predictor is removed; (4) all branches include history in their indices, so the downstream ID cache is removed and the chooser is simplified by removing extra bits relating to it; (5) the MACT-A and MACT-B tables are removed (since we only propagate one address in the ARF, the SACT alone will actively update all branches); (6) the GRT will no longer need trip-counts since the explicit loop predictor is removed. This cuts down the total cost of the fixed subcomponents (previously shown in Table 3) from about 20KB to 4KB. In the results that follow, we use an L1 SACT of 12KB with a virtualized L2 SACT in memory.

Second, we modified the explicit predictor component to control which predictions are provided by it. We use a tagged table to cache predictions, called the EX-cache. A dynamic branch’s index into the predictor is based on the same information as before (past retired branch’s ID and global history), except now only the lower part of this index is used to access a set in the EX-cache and the upper part of the index forms the tag. To reduce the
pressure on the EX-cache coming from the huge number of dynamic branches, the EX-cache caches only dynamic branches that have the uncommon branch outcome for a certain branch PC. To do so, we add a 3-bit saturation counter in the chooser table for each branch PC (similar to a bimodal predictor). The branch’s bias (indicated by the saturation counter) is checked when the EX-cache is passively trained at retirement. If the current branch outcome is the same as the branch’s bias in the chooser, the dynamic branch is not inserted in the EX-cache if it is not cached or evicted if it is cached. But if the current branch outcome differs from the branch’s bias, then the dynamic branch is inserted in the EX-cache (if it is not already cached). When making a prediction at fetch-time, the EX-cache is accessed in parallel with the chooser: if there is a hit in the EX-cache, then the prediction is opposite that of the branch’s bias in the chooser; if there is a miss in the EX-cache, then the prediction is that of the branch’s bias. Active updates are handled in a similar fashion: if the active-update outcome matches the branch’s bias, then the index being updated is removed from the EX-cache; if the active-update outcome differs from the branch’s bias, then the index being updated is inserted in the EX-cache. In the latter case, since the SACT caches \{PC, index\} tuples influenced by a certain address, it has the index that must be inserted into the EX-cache. This methodology requires the SACT to be trained with all branches that use the EX-cache regardless of their outcome. Again, this is efficiently accommodated by means of the virtualized L2 SACT. The EX-cache requires only tag arrays without any data arrays to cache predictions since the prediction is decided using hit/miss in the EX-cache and the bias bit in the chooser. The simplified EXACT-H is comprised of a 4KB EX-cache and 16KB of other fixed-cost overhead (including the L1 SACT). Note that only the 4KB EX-cache and
1KB chooser table reside on the critical path of instruction fetch, the other overhead is off the critical path which does not affect the cycle time of the processor.
Figure 36. Impact of adding a 4KB EX-cache and 16KB overhead to different L-TAGE predictor sizes.

Figure 36 shows both accuracy and performance improvement achieved by adding a 4KB EX-cache and 16KB overhead to different L-TAGE size configurations. For the two benchmarks bzip2 and twolf, adding the 4KB EX-cache and 16KB overhead improve on
accuracy and performance compared to doubling the $L\text{-}TAGE$ predictor size. Gzip benefits less from the simplified EXACT-H compared to results from the previous section. This is due to the large working set of gzip that needs a larger EX-cache to capture all dynamic branches (it benefits less from the bias optimization than other benchmarks). On average, adding the simplified EXACT-H provides a significant improvement in accuracy and performance that is comparable to doubling $L\text{-}TAGE$ but without sacrificing the cycle time.
Chapter 5: The Microarchitecture of EXACT-S

Figure 37 shows the major components of EXACT-S. The explicit predictor and default predictor are the same as in EXACT-H. Sections 5.1 through 5.3 describe shadow code and the Shadow-Code Table, the software-managed registers in the fetch unit, and shadow instructions that manage the registers, respectively. Active updates are explained in Section 5.4. Section 5.5 presents detailed examples of shadow code written for real benchmarks.

![Fig 37](image)

**Figure 37.** Major components of EXACT-S.

### 5.1 Shadow Code and the Shadow-Code Table

Shadow code is conveyed from software to hardware without changing the ISA, as follows.

- The shadow code is made a part of the data segment of the program binary. This allows for user-level loads to read bytes of the shadow code.
• The Shadow-Code Table is memory-mapped, making it possible to initialize its contents via user-level stores. Typically, a given machine comes with a machine manual that specifies its memory-mapped registers.

• Initialization code is added to the beginning of the program, that uses pairs of loads and stores to copy the bytes of shadow code from the data segment (loads) to the Shadow-Code Table (stores).

There is a one-to-one correspondence between selected instructions in the program and shadow-instructions in the shadow code. That is, each shadow-instruction shadows a particular instruction in the program. Each shadow-instruction is tagged with the PC of the instruction that it shadows. When the instruction is fetched, the shadow-instruction that shadows it is triggered via the PC. This process is explained next.

In parallel with fetching instructions from the instruction cache, the PCs of all instructions in the fetch bundle are searched in the Shadow-Code Table. If a PC hits, then the matching shadow-instruction should be read from the Shadow-Code Table and executed within the fetch unit. Shadow-instructions write, update, or read the software-managed registers in the fetch unit.

If a branch hits in the Shadow-Code Table, it will be predicted by the explicit predictor instead of the default predictor. This means that the Shadow-Code Table assumes the role of the chooser as shown in Figure 37.
For the tested benchmarks, we found that the Shadow-Code Table does not need to hold more than 64 shadow-instructions.

5.2 Software-Managed Registers in the Fetch Unit

Figure 38 shows the set of registers in the fetch unit that are read and written by the shadow code. The registers are divided into two sets: base registers and offset registers.

![Figure 38. The fetch unit registers read and written by the shadow code.]

**Base Registers (BR)**

A base register is used for conveying the base address of a data structure that is being traversed, such as an array or linked-list, to the fetch unit. There are two fields in a base register: base and shift amount. Base contains the base address. Shift amount is used by software for effectively modeling linked-lists as arrays in the explicit predictor, a process we refer to as “array-ifying” the linked-list. Array-ification is discussed below. Figure 38 also shows a ready bit associated with each base register. The ready bit is not visible to software,
i.e., it is purely microarchitectural. It will be explained when we discuss the shadow-instruction type that writes base registers.

To array-ify a linked-list, the address of its head node is considered the base address and subsequent nodes are considered to be at contiguous addresses with a stride of 1 (even though they are at arbitrary addresses). Thus, when sequencing a linked-list, dynamic branches that test its elements index a contiguous range of entries in the explicit predictor.

Basic array-ification, as described above, may cause different linked-lists to conflict in the explicit predictor because it is possible for their contiguous ranges to overlap. Whether or not two array-ified linked-lists conflict in the explicit predictor depends on their base addresses and sizes. True arrays cannot conflict in this way because they are laid out contiguously in memory; true arrays can only conflict in the explicit predictor due to its limited size.

The shift amount field is used by software to reduce conflicts between array-ified linked-lists. The shift amount field specifies an amount by which the base address should be shifted to the left when forming an index into the explicit predictor. Effectively, this provides a dedicated region for the linked-list within the explicit predictor as long as the number of elements is less than $2^{\text{shift amount}}$. 
Offset Registers (OR)

The fields of an offset register are designed with loops in mind, in particular, loops that iterate over and test the contents of data structures such as arrays and linked-lists. An offset register has three fields: offset, trip-count, and stride. The offset field generally corresponds to the loop induction variable. The trip-count is the number of times to iterate. The stride is the stride between elements that are accessed consecutively.

Thus, an offset register has two uses. First, the address of an element being tested can be calculated by adding offset*stride to a base address contained in a base register. Second, offset can be compared to trip-count to predict the loop branch: taken if they are not equal and not-taken if they are equal.

5.3 Shadow-Instructions

Figure 39 shows the format of a shadow-instruction. The PC field identifies the program instruction that is shadowed. The op-code field is 3-bits wide, supporting 8 different types of shadow-instructions. The base reg field is a 2-bit field that specifies which of the four base registers is used by the shadow-instruction (if applicable). Similarly, the offset reg field is a 2-bit field allowing the shadow-instruction to access one of four offset registers. The last field is immediate/flag. This field allows the shadow-instruction to use an immediate value. Some shadow-instructions use this field as a flag that controls incrementing an offset.
The shadow-instructions used in this thesis are enumerated below.

**seed shadow-instruction:**

The *seed* shadow-instruction shadows an instruction in the program that generates base addresses of arrays or linked-lists. Aside from *PC* and *op-code*, the fields relevant to the *seed* shadow-instruction are: *base reg* (specifies a base register) and *immediate/flag* (specifies a shift amount).

The *seed* shadow-instruction is the only one that cannot be executed immediately in the fetch unit because it must wait for its corresponding program instruction to execute. The *seed* shadow-instruction is executed as follows. First, the ready bit of the specified base register is cleared since the base address is not yet available. (Branches that are directed to use the explicit predictor – see *regular-branch* shadow-instruction – must use the default predictor if they reference a not-ready base register.) At the same time, its corresponding program instruction is annotated so that it knows to communicate with the fetch unit when it reaches two different pipeline stages: the rename stage and the writeback stage. When the program instruction is renamed, it sends the physical register tag of its destination register to the fetch unit. The fetch unit writes the physical register tag into the base register, in lieu of an actual value. When the program instruction has executed and reaches the writeback stage, it
communicates its physical register tag and value to the fetch unit. The fetch unit writes the value into the base register and sets the ready bit, only if the physical register tag matches the one currently in the base register (otherwise the value is simply discarded). The reason for comparing physical register tags is to handle the scenario of a second seed shadow-instruction reusing the base register before the first seed shadow-instruction has had a chance to write it. At this point, branches affiliated with the first seed shadow-instruction have already been predicted without the benefit of using the explicit predictor, moreover, the first seed shadow-instruction should not interfere with the branches affiliated with the second seed shadow-instruction that is now in progress. This implementation is basically an application of Tomasulo’s renaming algorithm [6].

**init-offset shadow-instruction:**

This shadow-instruction initializes all three fields of the specified offset register. It initializes the offset field to zero and the trip-count and stride fields to specified values. Aside from PC and op-code, the fields relevant to the init-offset shadow-instruction are: offset reg (specifies the offset register), base reg (specifies a stride value) and immediate/flag (specifies a trip-count value).

**loop-branch shadow-instruction:**

This shadow-instruction signals that the corresponding program instruction is a loop branch, and it will generate a prediction by comparing the offset and trip-count fields of the specified offset register. If they match, it means that the loop branch will fall-through and the
prediction should be not-taken, otherwise, the prediction should be taken. Executing this shadow-instruction will also automatically increment the *offset* field of the specified offset register. Aside from *PC* and *op-code*, the only other field relevant to the *loop-branch* shadow-instruction is *offset reg* (specifies the offset register).

**regular-branch shadow-instruction:**

This shadow-instruction signals that the corresponding program instruction is a branch, that should attempt to use the explicit predictor. The index into the explicit predictor is calculated by adding the specified base register (shifted for array-ified linked-lists) to the specified offset register and combining with the PC, as follows:

\[
\text{hash}\{\text{PC}, \((\text{base} \ll \text{shift amount}) + (\text{offset} \times \text{stride})\)\}
\]

The multiplication of the *offset* and *stride* can be performed by a shift operation if the stride is constrained to be a power of two.

Aside from *PC* and *op-code*, the fields relevant to the *regular-branch* shadow-instruction are: *base reg* (specifies the base register), *offset reg* (specifies the offset register), and *immediate/flag* (if 1, increment *offset*).

**linked-list-store shadow-instruction:**

After array-ification, the index of a linked-list element is not based on its address. Instead, it is based on the address of the head element plus a virtual offset. This complicates active updates for linked-list elements: the index cannot be inferred from the store address. The
software solution to this problem is to exploit opportunities to perform the store in the context of a linked-list traversal. If done in the context of a traversal, the store can infer its array-ified address the same way a branch does. This is achieved using the *linked-list-store* shadow-instruction. This shadow-instruction signals that the corresponding program instruction is a store that should calculate its active-update index using a specified base register and offset register, rather than use its own address for the active-update index. The partial index (does not include branch PC yet) is calculated in the fetch stage but will not be used until the store retires. Thus, the partial index is sent with the store down the pipeline until the dispatch stage at which time the partial index can be placed in the store’s reorder buffer entry for use at retirement.

Note, it is not a requirement that linked-list stores be performed in the context of a linked-list traversal. If it cannot be done in a linked-list traversal, there are other options depending on the situation:

- If the programmer or compiler knows that the store is rare or that it rarely flips branch outcomes, then it may be deemed unnecessary to trigger active updates by the store. As will be discussed in Section 5.4, active updates by a particular store PC can be disabled simply by excluding it from the store-PC-to-branch-PC table in the active update unit.

- If the programmer or compiler believes that the store frequently flips branch outcomes, then the affected static branches can be relegated to the default predictor, simply by not shadowing them with *regular-branch* shadow-instructions.
Aside from \textit{PC} and \textit{op-code}, the fields relevant to the \textit{linked-list-store} shadow-instruction are: \textit{base reg} (specifies the base register) and \textit{offset reg} (specifies the offset register).

**Summary of shadow-instructions:**

Table 4 shows a summary of the shadow-instructions used in this thesis. The table shows, for each shadow-instruction, the \textit{op-code}, \textit{base-reg}, \textit{offset-reg}, and \textit{immediate/flag} fields, as well as the operations performed by the shadow-instruction.

<table>
<thead>
<tr>
<th>shadow-instruction</th>
<th>op-code</th>
<th>base-reg</th>
<th>offset-reg</th>
<th>immediate/flag</th>
<th>operations</th>
</tr>
</thead>
</table>
| seed               | 0x0     | BR_0-BR_3 | N/A        | shift amount   | 1. BR_{base-reg}[base] = GPR_X  
|                    |         |          |            |                | 2. BR_{base-reg}[shift-amount] = shift amount |
| init-offset        | 0x1     | stride   | OR_0-OR_3 | trip-count     | 1. OR_{offset-reg}[offset] = 0  
|                    |         | value    |            |                | 2. OR_{offset-reg}[trip-count] = trip-count value  
|                    |         |          |            |                | 3. OR_{offset-reg}[stride] = stride value |
| loop-branch        | 0x4     | N/A      | OR_0-OR_3 | N/A            | 1. outcome = (OR_{offset-reg}[offset] = = OR_{offset-reg}[trip-count] ? NT : T)  
|                    |         |          |            |                | 2. OR_{offset-reg}[offset]++ |
| regular-branch     | 0x6     | BR_0-BR_3 | OR_0-OR_3 | increment offset | 1. pred-index = hash(branch-PC, 
|                    |         |          |            |                | ((BR_{base-reg}[base]<<BR_{base-reg}[shift amount]) + (OR_{offset-reg}[offset] + OR_{offset-reg}[stride])))  
|                    |         |          |            |                | 2. if (flag = = 1) OR_{offset-reg}[offset]++ |
| linked-list-store  | 0x7     | BR_0-BR_3 | OR_0-OR_3 | N/A            | active-update-address = ((BR_{base-reg}[base]<<BR_{base-reg}[shift amount]) + (OR_{offset-reg}[offset] + OR_{offset-reg}[stride])) |

**5.4 Active Update Unit**

The active update unit is similar to the active update unit in EXACT-H, with one major simplification. Since we use the ID of the branch to index into the explicit predictor, the
SACT is not required. Moreover, since we focus on single-address dynamic branches, the MACT-A and MACT-B are not required. Although MACT-A/MACT-B could be supported, they were found to be marginal to performance in our experiments with the simplified form of EXACT-H (see Section 4.5).

The only thing left is the value conversion mechanism, which has two pieces: (1) convert store PC to branch PC and (2) convert branch PC and store value to a branch outcome. In EXACT-H, the SACT/MACT-A/MACT-B also provided store-PC-to-branch-PC conversion. Now that those tables are gone, we need a dedicated store-PC-to-branch-PC conversion table. For converting the branch PC and store value into a branch outcome, we use the Ranges Reuse Table (RRT). (We did not see a need for the GRT in our experiments.)

While it is possible to dynamically train the store-PC-to-branch-PC table and the RRT, it is much more efficient to populate them via software (reduces hardware overhead, design complexity, and training time and power), in the same way that the Shadow-Code Table is populated: their contents are part of the data segment and they are populated using pairs of loads and memory-mapped stores in the initialization phase of the program.

For the tested benchmarks, we needed an RRT of only 8 entries (to support 8 branch PCs) and a store-PC-to-branch-PC conversion table of only 16 entries (to support 16 store PCs).
5.5 Code Examples

This section shows two code examples for the two benchmarks we experimented with for EXACT-S. For each benchmark, we show the source code that contains the branch instructions and shadow code associated with it. The gzip benchmark is a straightforward example. Generating its shadow code did not require altering the source code, since all the traversed data structures are long arrays. The twolf benchmark is a more challenging example. For best results, it requires altering the source code for different purposes, as explained in Section 5.5.2.

5.5.1 gzip

The source code of the two loops of interest in gzip is shown in Figure 40. There are two static branches that contribute almost 40% of gzip’s total mispredictions. The code that corresponds to these branches is highlighted in red font. The two branches are identical: a loop iterates on an array of 32,768 elements and tests whether or not each element is greater than a certain constant.

```c
// WSIZE = HASH_SIZE = 0x8000
for (n = 0; n < HASH_SIZE; n++)
{
    m = head[n];
    head[n] = (Pos) (m >= WSIZE ? m-WSIZE : NIL); // 1st branch of interest
}
for (n = 0; n < WSIZE; n++)
{
    m = prev[n];
    prev[n] = (Pos) (m >= WSIZE ? m-WSIZE : NIL); // 2nd branch of interest
}
```

Figure 40. Gzip source code.
Figure 41: Assembly code and its associated shadow code for gzip.
Figure 41 shows the assembly code (to the left) for the source code in Figure 40 and its shadow code (to the right). Assembly instructions that are not shadowed are marked with a light-grey font. The two branches of interest are marked with a red font. They are shadowed in the shadow code. All other shadowed instructions are marked with a bold font.

The shadow code for the two loops is similar, so we will explain only the shadow code for the first loop. The instruction at address 0x4013d8 is the instruction that will generate the base address of the first array, so it is shadowed by a seed shadow-instruction that writes BR0 (base = r7, the destination of the shadowed instruction, and shift amount = 0). We will shadow the instruction at address 0x4013e0 to initialize the offset (to 0), trip-count (to 32,768), and stride (to 2) fields of OR0. The instruction at address 0x401480 is the loop branch which will be shadowed by a loop-branch shadow-instruction (tests and increments OR0). And finally, the instruction at address 0x401440 is the branch of our interest, which will be shadowed by a regular-branch shadow-instruction.

5.5.2 twolf

Figure 42 shows the source code of the main kernel in twolf. The lines of code that correspond to the branches of our interest are marked with red font. These branches contribute to more than 70% of twolf’s total mispredictions. This code executes as follows:

There is an outer linked-list being traversed; each node in the outer linked-list contains a variable net; this variable is used to index an array to obtain two things, the head pointer netptr of an inner linked-list and the base pointer of an array rowsptr; both the inner linked-list and the array are traversed, and have branches with high misprediction rates.
Within the inner linked-list traversal, there is a branch that tests a flag variable
\( \text{netptr->flag} \) and two branches inside ABS macro instances that calculate absolute
values, in addition to the linked-list traversal branch. All four branches are highly
mispredicted. The macro that implements ABS is shown in Figure 43.

```c
#define ABS(value)   ( (value)>=0 ? (value) : -(value) )
```

Figure 42. Twolf source code.

Figure 43. Source code for the ABS macro.
The array `rowsptr` contains unsigned integers that are either 0 or 1. The array is traversed to calculate the number of 1’s and the number of 0’s in between the 1’s (i.e., excluding leading and trailing 0’s). The array traversal is a little bit complicated. The array is traversed by three for loops: (i) the first for loop traverses it from index zero until it reaches the first 1 and marks that index as `min`, (2) the second for loop traverses it backwards, from the end index of 17 (`numRows` has a fixed value of 16) until it reaches the last 1 (first 1 in reverse direction) and marks that index as `max`, and (iii) the third for loop traverses the indices in between the indices `min` and `max`.

The major challenges we faced while generating the shadow code for twolf were:

1) The branches belonging to the ABS macros depend on more than one address. (Aside from that, the `newx` input to the first ABS macro is control-dependent on the if `(netptr->flag == 1)` branch outcome. This makes it difficult to calculate its IDs. This could be fixed by moving the first ABS macro into both the if and else clauses, creating two different branches whose IDs can be calculated.)

2) The instruction that generates the base address of the array (`rowsptr = tmp_rows[net]`) is very close to the array branches, which means that the seed shadow-instruction will most likely not be able to copy the base address into its base register in timely fashion, especially in a deep OOO superscalar pipeline.

3) The array is accessed in a non-uniform fashion.
In light of the above problems, we modified the source code of this kernel as follows:

1) Since the current EXACT-S implementation does not support multiple-address dynamic branches, we could simply relegate the ABS branches to the default predictor. Instead, we if-converted the ABS macro so that it does not contain any branches at all. (Note that if-conversion does not require ISA support such as predication, conditional moves, etc.) The new ABS macro is implemented as an inline function as shown in Figure 44.

```c
inline int ABS(int value)
{
    int x;
    x = 0<value;
    x = x - 1;
    value = value ^ x;
    value = value - x;
    return value;
}
```

Figure 44. Modified ABS macro.

The modified ABS works as follows. After the first two statements, \( x \) is either \( 0x00000000 \) (0) or \( 0xFFFFFFFF \) (-1) depending on whether the ABS input (\( \text{value} \)) is positive or zero/negative, respectively. Then \( x \) is XORed with \( \text{value} \) and subtracted from that result. XORing with a zero and then subtracting a zero (the case when \( x=0x00000000 \)) will not change \( \text{value} \). XORing with -1 (the case when \( x=0xFFFFFFFF \)) will bitwise complement \( \text{value} \) and then adding 1 (subtracting -1) essentially gives the two’s complement of \( \text{value} \) which is equivalent to multiplying by -1 to change the sign.
The if (netptr->flag == 1) is if-converted in software in a similar way to the ABS macro. Strictly speaking, we could have targeted this branch with EXACT-S but chose to if-convert it instead. The modified code for it is shown in Figure 45. Similar operations are used to select either netptr->newx or oldx for newx. Depending on the value of netptr->flag, one of the two values will be masked with 0x00000000 and the other will be masked with 0xFFFFFFFF and propagated. The same principle is applied to conditionally reset netptr->flag (although in hindsight we could have simply unconditionally reset it).

\[
\text{newx} = (\text{netptr->newx} \& \neg(\text{netptr->flag} - 1))) + (\text{oldx} \& ((\text{netptr->flag}) - 1));
\text{netptr->flag} = \text{netptr->flag} \& (\text{netptr->flag} - 1);
\]

Figure 45. Modified code to replace the if (netptr->flag) statement.

2) The statement generating the base address of the array (rowsptr = tmp_rows[net] ;) is hoisted up in the code and placed right after the (net = termptr->net ;). This will give enough lead-time to obtain the base address of the array since the inner linked-list loop will precede the array traversal.

3) The way the array is traversed is changed to be more uniform. The main purpose of the three loops is to calculate the values of m (number of 1’s) and n (number of 0’s in between the 1’s). Figure 46 shows the new code to traverse the array and calculate the two values.
Figure 46. Modified code for traversing the array.

The array is traversed between indices 0 and numRows+1. We added an unsigned integer variable flag to record the occurrence of the first one. If flag is not set yet, the value of f and x will stay at zero. Once the first the one is encountered, flag will be set (and remain set) and both f and x will increment for each encountered zero. However, on every occurrence of 1, x will reset to zero. This way, m will record the number of 1’s, f will record the number of 0’s after encountering the first 1, and x will record the number of 0’s after the last-encountered 1 (i.e., the number of trailing 0’s). After exiting the loop, x is subtracted from f, which gives the number of 0’s between the 1’s.

In all fairness, we even found a way to if-convert the last remaining forward branch: if (rowsptr[row]). We did not, however, as we wanted to demonstrate the ability for EXACT-S to perfectly predict the remaining forward branch. In general, if-conversion is
impractical for forward branches that guard complex control-flow such as loops and function
calls. This thesis does raise a broader direction for invigorating microarchitecture research
and single-thread performance, which is to facilitate crafting source code for ILP similar to
how parallel programs are crafted for TLP.

The overall modified source code is shown in Figure 47. The statements that produce the
highly mispredicted branches are marked in red font. These of course will be addressed with
EXACT-S. The assembly code and the shadow code for it are shown in Figure 48.
for( termptr = antrmptr; termptr; termptr = termptr->nextterm )
{
    net = termptr->net;
    dimptr = netarray[net];
    rowsptr = tmp_rows[net];
    if( dimptr->dflag == 0 )
    {
        continue;
    }
    dimptr->dflag = 0;
    new_mean = dimptr->new_total / dimptr->numpins;
    old_mean = dimptr->old_total / dimptr->numpins;
    for( netptr = dimptr->netptr; netptr; netptr = netptr->nterm )
    {
        oldx = netptr->xpos;
        newx = (netptr->newx & (~(netptr->flag - 1))) + (oldx & ((netptr->flag) - 1));
        netptr->flag = netptr->flag & (netptr->flag - 1);
        *costptr += ABS( newx - new_mean ) - ABS( oldx - old_mean );
    }
    flag = 0;
    m = 0;
    f = 0;
    x = 0; /* # of leading 0's*/
    for(row=0; row<=numRows+1; row++)
    {
        flag = flag | rowsptr[row];
        if(rowsptr[row])
        {
            m++;
            x = 0;
        }
        else
        {
            f += flag;
            x += flag;
        }
    }
    f = f - x;
}

Figure 47. Overall modified source code for twolf.
Figure 48. Assembly code and the shadow code of the modified source code of twolf.

<table>
<thead>
<tr>
<th>PC</th>
<th>op-code</th>
<th>base-reg</th>
<th>offset-reg</th>
<th>immediate/flag</th>
<th>comment(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>412300</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>0</td>
<td>// seed: SR[base] = r12, (r12 is dest. register of 412300)</td>
</tr>
<tr>
<td>412318</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>17</td>
<td>// init-offset: [r0][offset] = 0,</td>
</tr>
<tr>
<td>412320</td>
<td>4</td>
<td>n/a</td>
<td>0</td>
<td>n/a</td>
<td>// regular-branch: pred_index = hash(PC, SR[base])</td>
</tr>
<tr>
<td>412360</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>// OR[offset] == OR[trip-count] ? NT : T,</td>
</tr>
<tr>
<td>412400</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>// OR[trip-count] = 0,</td>
</tr>
<tr>
<td>412448</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>// regular-branch: pred_index = hash(PC, (BR[base]&lt;&lt;BR[shift-amount]) + OR[offset]*OR[stride])</td>
</tr>
<tr>
<td>412490</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>// OR[offset]++</td>
</tr>
<tr>
<td>4124A8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>// OR[stride] = 1,</td>
</tr>
<tr>
<td>412530</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>// regular-branch: pred_index = hash(PC, SR[base] + OR[shift-amount])</td>
</tr>
<tr>
<td>412580</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>// OR[offset]++</td>
</tr>
<tr>
<td>412590</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>3</td>
<td>// seed: SR[base] = r8, (r8 is dest. register of 412590)</td>
</tr>
<tr>
<td>4125A0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>// OR[shift-amount] = 3,</td>
</tr>
<tr>
<td>4125F0</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>1</td>
<td>// OR[offset]++</td>
</tr>
<tr>
<td>412600</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>3</td>
<td>// seed: SR[base] = r11, (r11 is dest. register of 412600)</td>
</tr>
<tr>
<td>412650</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>1</td>
<td>// OR[offset]++</td>
</tr>
<tr>
<td>412700</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>// regular-branch: pred_index = hash(PC, SR[base] + OR[shift-amount])</td>
</tr>
<tr>
<td>412710</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>// OR[offset]++</td>
</tr>
</tbody>
</table>
Chapter 6: Evaluating EXACT-S

6.1 Methodology

For the experiments in this chapter, we present results for the two main benchmarks that benefit from EXACT: gzip and twolf. The microarchitecture parameters of the modeled processor are shown Table 5.

<table>
<thead>
<tr>
<th>Microarchitecture Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 I&amp;D Caches</td>
<td>64KB, 4-way, 64B line, hit=1 cycle, miss=10 cycles, 32 MHSRs</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Unified, 2MB, 8-way, 128B lines, hit=10 cycles, miss=200 cycles, 64 MHSRs</td>
</tr>
<tr>
<td>Reorder Buffer</td>
<td>256</td>
</tr>
<tr>
<td>Issue Queue</td>
<td>64</td>
</tr>
<tr>
<td>Load-Store Queue</td>
<td>64</td>
</tr>
<tr>
<td>Rename Map Checkpoints</td>
<td>16</td>
</tr>
<tr>
<td>Fetch-to-exec. Pipe depth</td>
<td>20 stages</td>
</tr>
<tr>
<td>Fetch/Issue/Retire Width</td>
<td>4 instr./cycle</td>
</tr>
</tbody>
</table>

For the gzip benchmark, we used the unmodified source code. For the twolf benchmark, we used the modified source code presented in Chapter 5. We noticed that modifying the source code for twolf increased the instruction count required to do an equivalent amount of work, from 100 million instructions to 116 million instructions. Accordingly, our performance comparisons in Section 6.3 are based on the total number of cycles to complete the same amount of work, not instructions per cycle (IPC).
For the experiments presented in Sections 6.2 and 6.3, we used fixed configurations for the shadow-code table, software managed registers, RRT, and store-PC-to-branch-PC conversion table. We only varied the sizes of the explicit predictor and default predictor. Since the overall cost of the four auxiliary components is less than a kilobyte (see Table 6), we did not include their cost in the overall budget of EXACT-S in the results of Sections 6.2 and 6.3.

Table 6. Fixed-configuration subcomponents for EXACT-S. Not included: default predictor and explicit predictor.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Structure</th>
<th># of entries / organization</th>
<th>Contents per entry</th>
<th>Size (K-Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Unit</td>
<td>Shadow-Code Table</td>
<td>64 entries / fully-assoc.</td>
<td>16-bit PC tag + 15-bit shadow-instruction payload</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td>Software-Managed Registers</td>
<td>4 entries / not tagged</td>
<td>1 ready bit + 20-bit base + 5-bit shift-amount + 16-bit offset + 16-bit trip-count + 3-bit stride</td>
<td>0.03</td>
</tr>
<tr>
<td>Active Update Unit</td>
<td>RRT</td>
<td>8 entries / 4-way set-assoc.</td>
<td>1 valid bit + 2-bit LRU + 13-bit tag + 4x16-bit for (min_NT, max_NT, min_T, max_T)</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td>Store-PC-to-Branch-PC</td>
<td>16 entries / 4-way set-assoc.</td>
<td>1 valid bit + 2-bit LRU + 12-bit tag + 8x4-bit branch-PCs</td>
<td>0.24</td>
</tr>
</tbody>
</table>

Total cost of fixed subcomponents (does not include default predictor and explicit predictor): 0.59

6.2 Accuracy vs. Storage Budget

Figure 49 shows the accuracies of different predictor configurations at different storage budgets, for (a) gzip and (b) twolf. The labels for the graphs in this section and the next one are interpreted as follows:

i) Which version of the source code was used: “original-code” means we used the original source code (unmodified), “modified-code” means we used the modified source code. The latter case only applies to twolf.
ii) Which predictor was used: “L-TAGE” means only L-TAGE was used, and the configuration for L-TAGE is the same as that in Section 4.4 for each cost point. “1/2 L-TAGE + 1/2 EXACT-S” means we divided the total budget equally between the default (L-TAGE) and explicit predictors of EXACT-S.

iii) Which indexing model was used for EXACT-S: For EXACT-S, two indexing models were evaluated: ideal-index assumes that a base register is always ready for predicting branches that reference it, whereas real-index checks whether or not the ready-bit is set in the base register.

The results of gzip in Figure 49(a) are very similar to the EXACT-H results in Figure 35. However, with EXACT-S, reducing the misprediction rate from 7.5% to 4.7% does not require a large SACT or the 20KB of other fixed structures. At a cost of only 32 KB, EXACT-S achieves most of its accuracy improvement, and at 64 KB and higher budgets,
EXACT-S reaches its peak accuracy. At the 256 KB budget, EXACT-S accuracy drops significantly: this is because of severe aliasing between the two arrays that were explained in Section 5.5.1. The hashing function we used to generate the predictor index produces aliased indices at that budget. Nonetheless, EXACT-S plus L-TAGE still outperforms L-TAGE alone at the 256 KB point. Results for EXACT-S using ideal-index and real-index are virtually identical. This is because the latency to generate the arrays’ base addresses is almost irrelevant, since the arrays have 32K elements.

From the results in Figure 49(b), the misprediction rate of twolf is halved by just modifying its source code (without any EXACT-S intervention): it improves from 8.7% to 4.2% at the 32 KB budget and from 6.7% to 3.7% at the 512 KB budget. At a total predictor budget of 32 KB, EXACT-S reduces the misprediction rate of L-TAGE from 4.2% to 2.6% and 2.4% for real-index and ideal-index, respectively. Overall, for twolf, the results for EXACT-S are superior to the results for EXACT-H, in several respects. Compared to L-TAGE alone on the modified source code, EXACT-S about halves the misprediction rate across all storage budgets, even at 32 KB. That is, the improvement is more substantial than with EXACT-H and it is achieved even at smaller budgets. Moreover, the gap between real-indexing and ideal-indexing is narrower with EXACT-S than with EXACT-H.

6.3 Performance Improvement

Figure 50 and Figure 51 show the performance improvement of EXACT-S over L-TAGE, for gzip and twolf, respectively. For each figure, we show performance improvement based on two different latency models of the predictor. The graph to the left (“ideal access latency”)
assumes that the specified predictor is accessed in a single cycle. The graph to the right (“real access latency”) assumes the specified predictor is an overriding predictor [27]: for each storage budget, the prediction system is composed of a small, 8 KB gshare predictor accessed in a single cycle and backed by the specified overriding predictor. The access latencies of different budgets of overriding predictors, for both “L-TAGE” and “1/2 L-TAGE + 1/2 EXACT-S”, are listed in Table 7.
Table 7. Access latency for overriding predictor as a function of budget.

<table>
<thead>
<tr>
<th>Budget</th>
<th>Access Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 KB</td>
<td>3 cycles</td>
</tr>
<tr>
<td>64 KB</td>
<td>4 cycles</td>
</tr>
<tr>
<td>128 KB</td>
<td>5 cycles</td>
</tr>
<tr>
<td>256 KB</td>
<td>6 cycles</td>
</tr>
<tr>
<td>512 KB</td>
<td>7 cycles</td>
</tr>
</tbody>
</table>

For example, if we have a 128 KB L-TAGE, a preliminary prediction is obtained from the 8 KB gshare in one cycle, and 5 cycles later this prediction is either confirmed or overridden by the 128 KB L-TAGE. For the EXACT-S case, a preliminary prediction is obtained from the 8 KB gshare in one cycle, and 5 cycles later, the prediction is confirmed or overridden by either the 64 KB L-TAGE or the 64 KB explicit predictor depending on which predictor is selected.

![Graph](image)

(a) ideal access latency  
(b) real access latency (overriding predictor)

Figure 50. Performance improvement for different configurations of EXACT-S, for the gzip benchmark. (a) all predictors are accessed in a single cycle, (b) all predictors are overriding predictors with a size-dependent access latency.
Gzip’s performance results follow from its accuracy results: EXACT-S improves performance by 13% at 32 KB and 17-18% at higher budgets. At 256 KB, a dip in the performance improvement occurs due to the aliasing problem explained in the previous section. “Real-index” and “ideal-index” achieve the same performance. Performance improvements are a bit higher in the overriding predictor case as compared to the 1-cycle latency assumption.

![Figure 51. Performance improvement for different configurations of EXACT-S, for the twolf benchmark.](image)

(a) ideal access latency   (b) real access latency (overriding predictor)

Similarly, twolf shows a significant performance improvement using EXACT-S. Unlike gzip, twolf shows a small gap between real-index and ideal-index performance. A positive result is that the performance improvement of the overriding EXACT-S is nearly constant with increasing budget.
Chapter 7: Related Work

7.1 Load-Based Branch Prediction Techniques

7.1.1 ARVI Predictor

The ARVI predictor (Available Register Value Information) [12] (Chen, Dropsho, and Albonesi) uses live-in register values of a dynamic branch’s backward-slice to predict the branch, if these values are available in the register file (committed). Backward-slices terminate at loads. Their results showed that 80% of dynamic branches depend on pending loads whose values are unavailable in the pipeline for making predictions. This highlights the need for explicitly predicting load values or addresses. Our real indexing strategy predicts the ID of the dynamic branch being fetched which is tantamount to predicting the addresses of loads in its backward-slice.

7.1.2 ABC Predictor

The ABC predictor (address-branch correlation) [20] (Gao, Ma, Dimitrov, and Zhou) specifically targets hard-to-predict branches that depend on loads that miss in the L2 cache. They exploit two observations: (1) the value contents of the data structures tested by these branches tend to be stable, therefore, a branch outcome correlates well with simply the address of the data structure, and (2) while the actual value is unavailable by virtue of being retrieved from the memory system, the address is available since the load on which the branch depends has already issued to the memory system. Accordingly, they use the address of the missed load to repredict the direction of the load’s dependent branch. The fetch unit is
redirected if the reprediction does not match the original prediction. Repredicting long-latency branches is comparatively simpler than the topic tackled in this thesis because the address is available for the branch being repredicted, or, for linked-data-structure type loads, the address of a previous iteration of the same load is available. In contrast, our predictor must hide the core pipeline latency for all branches, requiring the ID (essentially addresses) for every branch to be predicted which is the impetus for our novel indexing strategy. Moreover, this thesis is the first to propose active updates and show that active updates are important when targeting all branches: this result differs from Gao et al.’s observation that underlying data is stable, which is a reasonable assumption when narrowly targeting some mispredicted branches.

7.2 Techniques for Reducing Destructive Aliasing

With the advent of two-level adaptive branch prediction [33][38][55], there has been a plethora of research on branch predictors that combine branch PCs, local/global branch history, and path information in ingenious ways to achieve ever higher accuracy. Many of the branch prediction techniques that followed, focused on removing destructive aliasing due to a limited budget for the branch predictor.

7.2.1 The Bi-Mode Predictor

The Bi-Mode predictor [30] (Lee, Chen, and Mudge) divides the PHT into two halves. Branch PCs conflicting with each other get steered to one half of the table based on their outcomes. A PC-indexed chooser decides which half to use at prediction time.
7.2.2 The YAGS Predictor

The YAGS predictor [15] (Eden and Mudge) is similar to the Bi-Mode predictor, but it employs tags. Although tags are expensive, the direction bits are removed and the prediction is inferred from the PC-indexed chooser table. This enables keeping the dominant bias of the branch in the chooser table, and some rare cases in the direction caches. This cost-saving principle is exploited by the EX-cache implementation of EXACT-H, although indexing is fundamentally different (IDs instead of global branch history).

7.2.3 The gskew Predictor

The gskew branch predictor [35] (Michaud, Seznec, and Uhlig) and the 2bc-gskew predictor [45] (Seznec, Felix, Krishnan, and Sazeides) try to reduce destructive aliasing by using skewed redundancy. In this technique, the PHT is divided into an odd number of tables and each table is indexed with a different hashing function. The final prediction is extracted by reading predictions from all the tables and performing a majority vote. The rationale for this approach is, if one of the tables provides a wrong prediction due to a destructive update from another branch PC, most likely the other tables will provide the correct prediction. This technique was implemented in the Alpha EV8 processor [45].

7.2.4 The Filtering Technique

Filtering is another technique to reduce destructive aliasing [10] (Chang, Evers, and Patt). The authors suggested that branch PCs that are highly biased not update the PHT. For these branch PCs, their predictions are obtained from a direction bit kept in the BTB.
7.2.5 The Agree Predictor

The Agree predictor [51] (Sprangle, Chappell, Alsup, and Patt) used a similar idea. A direction bit is kept with each branch in the BTB. Now the PHT will be used in a different way: the 2-bit counter in the PHT indicates whether to agree or disagree with the prediction given by the direction bit in the BTB. This way, if two branches with different directions reference the same PHT entry, they will not be mispredicted because both will be pointing toward agreeing with their respective directions in the BTB and they both increment the agree counter although they have different outcomes.

7.3 Making Use of Long Branch History

7.3.1 The Perceptron Predictor

The Perceptron predictor [26] (Jiménez and Lin) used a single layer Perceptron to learn the target Boolean function (taken/not-taken) of n inputs. These n inputs correspond to n bits in the global history. A vector of weights is used to measure the correlation for each bit in the global history. A weight’s magnitude indicates the strength of correlation and its sign indicates either positive or negative correlation. The approach requires performing a dot-product of the bits in the global history with the weight vector and summing up the result to determine the final prediction.

The Scaled Neural Predictor 0[5] (Amant, Jiménez, and Burger) uses an analog circuit to perform the dot-product portion of the prediction loop, for power savings and faster computation of the prediction.
7.3.2 The O-GEHL Predictor

The O-GEHL predictor [41] (Seznec) employs multiple prediction tables. Each table is indexed with a combination of the branch PC and a different amount of global history. An entry in each of these tables is a signed saturation counter quantity. A positive quantity means taken prediction while negative means not-taken. The history length used for each table is determined through a geometric series equation. The final prediction is taken by adding the prediction counters from all the tables. If the final summation is positive then the final prediction is taken, otherwise it is not-taken. Summation of all prediction counters was proposed in the Fusion-based predictor [31] (Loh and Henry).

7.3.3 The L-TAGE Predictor

The TAGE [47] (Seznec and Michaud) and L-TAGE [43] (Seznec) predictors also use a geometric series to determine the history length for each prediction table. Each table entry contains a tag in addition to the prediction counter. The table with the longest global history that hits, determines the final branch prediction. The L-TAGE predictor is similar to the TAGE predictor, but it also contains a loop predictor.

7.3.4 Using Affectors and Affectees Information for Branch Prediction

Two previous works [54] (Thomas, Franklin, Wilkerson, and Stark) [40] (Sazeides, Moustakas, Constantinides, and Kleanthous) used the program’s dataflow graph to identify branches that are correlated, with the goal of pin-pointing non-consecutive global history bits that are correlated. Paring down global history to its useful bits is good for reducing training time and making a small predictor perform close to a large one. Their work is still confined
to the accuracy bounds of branch-history-based prediction, which we show is limited by lack of specialization and store updates.

7.4 Assigning Confidence for Branch Prediction

Branch confidence estimation involves consulting a secondary predictor to gauge the confidence level of the prediction provided by the branch predictor.

7.4.1 The JRS Confidence Estimator

The JRS confidence estimator [24] (Jacobsen, Rotenberg, and Smith) is a table of saturating counters indexed with the branch PC and global branch history (similar to gshare). A given counter is incremented on a correct prediction and reset on a misprediction. A dynamic branch’s prediction is considered confident if its confidence counter is saturated, which means it has been correct for the last n previous predictions.

7.4.2 The Perceptron-Based Confidence Estimator

In [3] (Akkary, Srinivasan, Koltur, and Refaai), the authors use a Perceptron-based scheme for estimating confidence of branch predictions, similar to regular branch prediction with perceptrons [26].

7.4.3 The Prophet/Critic Predictor

The Prophet/Critic branch predictor [16] (Falcón, Stark, Ramirez, Lai, and Valero) is similar to the previous confidence estimation techniques. It is composed of two predictors. The Prophet predictor is a conventional predictor like gshare, and it produces predictions for the current branch and future predictions. The Critic predictor uses previous branch history, the current branch prediction, and future predictions to critique predictions provided by the Prophet predictor.
7.5 Value-Based Techniques for Branch Prediction

7.5.1 Hardware-Only Value Prediction Techniques for Branch Prediction
In [21] (Gonzalez and Gonzalez), the authors explicitly value-predict the source operands of a branch to calculate its direction early in the pipeline. The authors of [22] (Heil, Smith, and Smith) proposed using the last committed difference between a branch’s source operands coupled with the number of outstanding instances of the branch.

7.5.2 Compiler-Assisted Techniques for Value-Based Branch Prediction
In [7] (August, Connors, Gyllenhaal, and Hwu), the authors harnessed the compiler to synthesize predicates that may reflect arbitrary state upon which a branch is correlated, including prior values. The compiler hoists the predicate condition instruction in the code in order to provide an early predicate value. However, this predicate value is used as a branch prediction. The original predicate condition instruction is still needed to validate the prediction. The authors of [32] (Mahlke and Natarajan) used profile information of prior register values to correlate them with the branch outcomes of certain branch PCs.

7.6 Pre-Execution-Based Techniques for Branch Prediction
In [17] (Farcy, Temam, Espasa, and Juan ), [39] (Roth and Sohi), and [56] (Zilles and Sohi), the authors proposed extracting, hoisting, and pre-executing the backward slices of hard-to-predict branches so that their outcomes are known when fetched.
7.7 Software Techniques for Explicitly Managing Microarchitectural Resources

7.7.1 Simultaneous Subordinate Micro Threading (SSMT)
In [11] (Chappell, Stark, Kim, Reinhardt, and Patt), the authors suggested using a micro-thread to manage a large PAg branch predictor [55] stored in main memory. After each branch is fetched, a micro-thread is spawned to update the branch predictor and prepare a prediction for the next branch instance, which is stored in a prediction cache.

7.7.2 Dynamic Instruction Stream Editing (DISE)
In DISE [13] (Corliss, Lewis, and Roth), the authors suggested using a pattern table to match on a sequence of instructions and replace them with either a native sequence of instructions or micro-instructions which are fetched from a replacement table. They applied DISE to memory fault isolation and dynamic code compression.
Chapter 8: Summary and Future Work

8.1 Summary

Branches that depend on load instructions are a leading cause of mispredictions by state-of-the-art branch predictors. The study of mispredictions provided in Chapter 2 shows that the global branch history used by state-of-the-art branch predictors often fails to distinguish multiple instances of a branch that test different elements of a large data structure. The study also reveals that store instructions to the elements of a data structure may cause their corresponding branch outcomes to flip when they are next encountered.

We proposed EXACT to solve these two problems. First, the context for predicting an instance of a branch is based on the address of the element it tests. Using addresses ensures dedicated predictions for different elements. Second, stores to elements directly update their predictions in the predictor. This novel “active update” concept avoids mispredictions that are otherwise incurred by conventional passive training.

This dissertation explored two implementations of EXACT. The first is a hardware-only solution (EXACT-H) and the second is a combined hardware/software solution (EXACT-S).
The EXACT-H predictor index is based on the load address of a prior retired branch since the load address of the current branch is not available when it is fetched. A side-effect of this indirect indexing strategy is that a branch’s index into the predictor cannot be deduced from the load address on which it depends. This leads to a storage-intensive active update unit. The dedicated storage requirement is reduced by virtualizing it, exploiting the key observation that active updates are tolerant of 100s of cycles of latency.

In EXACT-S, the programmer or compiler conveys key information directly to the fetch unit that it can use to generate branches’ load addresses in a timely manner. This both increases accuracy and greatly simplifies the active update unit, compared to EXACT-H, because of the direct indexing strategy.

Our results show that EXACT-S is able to capture the same or higher prediction accuracy achieved by EXACT-H using less dedicated storage. In both gzip and twolf, EXACT-S is able to remove close to 50% of their mispredictions compared to an equal size L-TAGE.

8.2 Future Work

1) Explore compilers for generating the shadow code. This could be performed easily, by identifying difficult-to-predict branches that test arrays or stable linked-list data structures. The compiler will find the instruction that generates the base address, the branch instruction, and the loop branch that iterates over the data structure. This will reduce the programmer’s
time spent on inspecting the code to map regular instructions to shadow instructions, and enables experimenting with a wider range of benchmarks.

2) Apply the if-conversion technique we used in conjunction with EXACT-S for removing branches a step further, and apply it across a wide range of benchmarks. This would allow us to determine if we can reach close-to-perfect branch prediction and peak IPC.

3) The complexity of the source code of certain applications might be prohibitive for extracting shadow code or performing if-conversion. At the same time, these benchmarks might benefit from maximizing resources allocated to the default predictor. In this case, it is desirable to allocate all resources to the default predictor and none to the explicit predictor. Yet, for applications that benefit significantly from EXACT-S, it is desirable to balance the resources between the default and explicit predictors. The fact that different applications will favor different allocations is motivation for a reconfigurable branch predictor. For example, if an application cannot make good use of the explicit predictor, its table could be reconfigured to be part of L-TAGE by logically extending one of its tables.
Chapter 9: Bibliography


[57] The 2nd JILP Championship Branch Prediction Competition (CBP-2). http://camino.rutgers.edu/cbp2/