EXPLORING VIRTUALIZATION TECHNIQUES FOR BRANCH OUTCOME PREDICTION

by

Maryam Sadooghi-Alvandi

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering

UNIVERSITY OF TORONTO

© Copyright by Maryam Sadooghi-Alvandi 2011
Abstract

Exploring Virtualization Techniques for Branch Outcome Prediction

Maryam Sadooghi-Alvandi
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2011

Modern processors use branch prediction to predict branch outcomes, in order to fetch ahead in the instruction stream, increasing concurrency and performance. Larger predictor tables can improve prediction accuracy, but come at the cost of larger area and longer access delay.

This work introduces a new branch predictor design that increases the perceived predictor capacity without increasing its delay, by using a large virtual second-level table allocated in the second-level caches. Virtualization is applied to a state-of-the-art multitable branch predictor. We evaluate the design using instruction count as proxy for timing on a set of commercial workloads. For a predictor whose size is determined by access delay constraints rather than area, accuracy can be improved by 8.7%. Alternatively, the design can be used to achieve the same accuracy as a non-virtualized design while using 25% less dedicated storage.
Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Andreas Moshovos, for his guidance, inspiration, and support. His dedication and expertise has helped me immensely during this work.

I am grateful to my thesis defense committee members, Professors Natalie Enright-Jerger, Greg Steffan, and Dimitrios Hatzinakos for their time and invaluable feedback.

My sincerest appreciation goes to Henry, Kaveh, Ioana, and Jason for all the assistance and suggestions that they provided me in completing my thesis. I would like to thank all the members of our group and the many wonderful people I have met in graduate school, especially Elias, Myrto, Goran, Ian, Patrick, Vitaly, Islam, Nahi, Andrew, Diego, Xander, Danyao, Pam, and Meric.

Finally, I would like to thank my parents, Mohammad and Mahrokh, and my brother, Milad, for their love, support, and encouragement throughout my life. I wouldn’t be here without you.
Table of Contents

Abstract ................................................. ii

Acknowledgements ...................................... iii

Table of Contents ....................................... iv

List of Tables .......................................... vii

List of Figures ......................................... viii

1 Introduction ......................................... 1

1.1 Motivation ........................................ 1

1.2 Thesis Contributions ............................. 4

2 Background ........................................... 6

2.1 Background on Branch Prediction ............... 6

2.1.1 Branch Prediction Mechanisms ............... 8

2.2 Background on Predictor Virtualization ...... 11

3 Related Work ........................................ 13

3.1 Related Work on PV ............................... 13

3.2 Related Work on Hierarchical Branch Predictors .................. 14

3.3 Related Work on Improving TAGE Prediction Accuracy .......... 16

4 Methodology ......................................... 17

4.1 Simulation Infrastructure ......................... 17
Table of Contents

4.1.1 Timing Model .................................................. 17
4.1.2 Benchmarks ..................................................... 18
4.1.3 Branch Prediction Accuracy Metric ....................... 19
4.1.4 Simulated Microprocessor ................................... 19
4.2 TAGE Parameters .................................................. 19
  4.2.1 Number of Tagged Tables .................................. 20
  4.2.2 Entry Sizes .................................................... 20
  4.2.3 History Lengths .............................................. 20
  4.2.4 Table Sizes ................................................... 21

5 Virtualization Overview ........................................... 25

6 Paging Scheme ..................................................... 28
  6.1 Design .......................................................... 28
  6.2 Architecture ..................................................... 29
  6.3 Design Space Exploration ..................................... 32
    6.3.1 TAGE Baseline and Potential for Virtualization ........... 33
    6.3.2 Choosing Ltable .......................................... 34
    6.3.3 Maximum Potential Improvement with the Paging Scheme ... 36
    6.3.4 Page Size .................................................. 37
    6.3.5 Instruction Block Size .................................... 46
    6.3.6 L2 Latency ................................................ 51
    6.3.7 Sensitivity to Initial Table Sizes ......................... 59
  6.4 Virtualized Design Accuracy and Improvement Over Baseline for Delay-Constrained Configurations ............... 61

6.5 Summary .......................................................... 63

7 Packaging Scheme .................................................. 64
  7.1 Design .......................................................... 64
  7.2 Architecture ..................................................... 66
# Table of Contents

7.3 Experimental Results

7.3.1 Choosing Ptable

7.3.2 Potential Improvement with the Packaging Scheme

7.3.3 Package Swap Frequency

7.3.4 Instruction Block Size

7.3.5 Improvement for Packaging Scheme Before Considering L2 Delay

7.3.6 Prefetching

7.4 Summary

8 Conclusion

8.1 Summary

8.1.1 Paging Scheme

8.1.2 Packaging Scheme

8.1.3 Comparison of the Two Schemes

8.2 Limitations and Future Work

Appendices

A Branch Prediction Schemes

A.1 Bimodal

A.2 Two-Level Adaptive Predictors

A.3 Gshare

A.4 2BC-Gshare

A.5 The Skewed Predictor and 2BC-Gskew

A.6 YAGS

A.7 Perceptron

A.8 O-GEHL

References
List of Tables

4.1 Workload Descriptions .......................................................... 18
4.2 Tag Width for Predictor Table Entries ................................. 20
4.3 Geometric Functions Used for History Lengths ...................... 20
4.4 Delay-Constrained Table Size Configurations Established Using CACTI ... 24

6.1 Default Ltable Configuration .................................................. 36
6.2 Ltable History Lengths ....................................................... 40
6.3 Paging Scheme Virtualized Design Configuration ................. 52

7.1 Default Ptable Configuration ................................................ 71
7.2 Packaging Scheme Configuration .......................................... 71
7.3 Final Packaging Scheme Configuration ................................. 77
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>TAGE Predictor with N Tagged Tables</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Phantom-BTB Architecture [10]</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>Organization of Tag and Data Arrays in CACTI [35]</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Access Latency for TAGE Modeled as an N-Way Set-Associative Cache (Tag Width: 5 bits, Data Width: 8 bits)</td>
<td>23</td>
</tr>
<tr>
<td>4.3</td>
<td>Access Latency for the Bimodal Table Modeled as an SRAM Array of Bytes</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Paging scheme increases the perceived length of a predictor table</td>
<td>26</td>
</tr>
<tr>
<td>5.2</td>
<td>Packaging scheme increases the perceived width of a predictor table</td>
<td>26</td>
</tr>
<tr>
<td>6.1</td>
<td>Paging Scheme Architecture</td>
<td>30</td>
</tr>
<tr>
<td>6.2</td>
<td>MPKI for Baseline TAGE at Three Different Sizes</td>
<td>34</td>
</tr>
<tr>
<td>6.3</td>
<td>MPKI for TAGE with a Single 32K-Entry Table</td>
<td>35</td>
</tr>
<tr>
<td>6.4</td>
<td>Maximum potential improvement achievable by adding a single 32K-entry table</td>
<td>36</td>
</tr>
<tr>
<td>6.5</td>
<td>Effect of page size on MPKI</td>
<td>38</td>
</tr>
<tr>
<td>6.6</td>
<td>Maximum potential improvement with a page size of 32 entries</td>
<td>39</td>
</tr>
<tr>
<td>6.7</td>
<td>Effect of page size on page swap frequency</td>
<td>41</td>
</tr>
<tr>
<td>6.8</td>
<td>Effect of page size on predictor tables’ MPKI contribution (N=1)</td>
<td>45</td>
</tr>
<tr>
<td>6.9</td>
<td>Effect of page size on predictor tables’ MPKI contribution (N=8)</td>
<td>45</td>
</tr>
<tr>
<td>6.10</td>
<td>Effect of instruction block size on page swap frequency (instruction block size = $2^X \times$ page size)</td>
<td>47</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.11</td>
<td>Effect of instruction block size on MPKI (instruction block size = $2^X \times \text{page size}$, N=12)</td>
<td>48</td>
</tr>
<tr>
<td>6.12</td>
<td>Effect of instruction block size on the fraction of pages never accessed (instruction block size = $2^X \times \text{page size}$)</td>
<td>49</td>
</tr>
<tr>
<td>6.13</td>
<td>Maximum potential improvement with an instruction block size of 256 B ($X = 3$)</td>
<td>50</td>
</tr>
<tr>
<td>6.14</td>
<td>Effect of latency on percentage improvement in MPKI</td>
<td>52</td>
</tr>
<tr>
<td>6.15</td>
<td>Potential improvement for virtualized design with 2K-entry dedicated tables</td>
<td>53</td>
</tr>
<tr>
<td>6.16</td>
<td>Effect of latency on L1 miss rate</td>
<td>55</td>
</tr>
<tr>
<td>6.17</td>
<td>Effect of latency on page swap frequency</td>
<td>56</td>
</tr>
<tr>
<td>6.18</td>
<td>Effect of latency on predictor table’s contribution to MPKI and final prediction (a) MPKI contribution (b) contribution to final prediction</td>
<td>57</td>
</tr>
<tr>
<td>6.19</td>
<td>Maximum potential improvement achievable by adding a single 32K-entry table – 4K-entry dedicated tables</td>
<td>59</td>
</tr>
<tr>
<td>6.20</td>
<td>Improvement loss due to page size, instruction block size, and latency constraints – 4K-entry dedicated tables</td>
<td>60</td>
</tr>
<tr>
<td>6.21</td>
<td>Accuracy improvement for virtualization of delay-constrained configurations</td>
<td>61</td>
</tr>
<tr>
<td>6.22</td>
<td>Effect of virtualization on L2 traffic for delay-constrained configurations</td>
<td>62</td>
</tr>
<tr>
<td>7.1</td>
<td>Packaging scheme architecture</td>
<td>67</td>
</tr>
<tr>
<td>7.2</td>
<td>Package number determined by $n$ bits of branch PC</td>
<td>68</td>
</tr>
<tr>
<td>7.3</td>
<td>MPKI for TAGE with a table of 1K packages added to a single predictor table</td>
<td>70</td>
</tr>
<tr>
<td>7.4</td>
<td>Maximum potential improvement with packaging scheme using 64 B packages and instruction block size</td>
<td>72</td>
</tr>
<tr>
<td>7.5</td>
<td>Package swap frequency for prediction and packaging buffers</td>
<td>73</td>
</tr>
<tr>
<td>7.6</td>
<td>Effect of instruction block size on prediction buffer package swap frequency</td>
<td>75</td>
</tr>
<tr>
<td>7.7</td>
<td>Effect of instruction block size on the fraction of packages never accessed</td>
<td>76</td>
</tr>
<tr>
<td>7.8</td>
<td>Effect of instruction block size on MPKI</td>
<td>77</td>
</tr>
</tbody>
</table>
List of Figures

7.9 Potential improvement for packaging scheme using an instruction block size of 1 KB ........................................... 78
7.10 Maximum potential improvement before considering L2 delay for paging and packaging schemes ........................................... 79
7.11 Accuracy of a 1K-entry PTB on the SPEC CPU 2000 benchmark set ........................................... 80

A.1 Bimodal Predictor [3] .................................................................................................................. 87
A.2 Two-Level Adaptive Predictor [14] ............................................................................................... 88
A.3 Gshare Predictor [3] .................................................................................................................. 89
A.4 2bc-gshare Predictor [3] ............................................................................................................ 90
A.5 2bc-gskew Predictor [4] ............................................................................................................ 91
A.6 YAGS Predictor [7] .................................................................................................................. 92
A.7 Perceptron Predictor [24] ........................................................................................................... 93
A.8 GEHL Predictor [25] ................................................................................................................ 94
Chapter 1

Introduction

1.1 Motivation

Modern microprocessors are continuously challenged to execute more instructions per second. To achieve this goal, microprocessors execute multiple instructions in parallel, exploiting Instruction Level Parallelism, the inherent potential for multiple instructions to be processed simultaneously. This Instruction Level Parallelism can be limited by the various dependencies between instructions. An important class of such dependencies is control flow dependencies, which determine the order of instruction execution. This class of dependencies is introduced by branch instructions, instructions that can alter the flow of execution, usually based on some conditions. When a conditional branch instruction is fetched, it can take several cycles before the outcome of its condition is evaluated, and until then, it is unclear which path the processor should follow. To avoid stalling for branches, microprocessors rely on speculation. Branch prediction is a form of speculation that allows the processor to speculatively fetch and execute instructions along a predicted path. The branch predictor uses statistical information based on past branch behaviour to determine the likely outcome of each branch. This technique enhances performance as it keeps the pipeline full, even in the presence of control hazards.

Branch prediction accuracy impacts processor performance as branch mispredictions represent a significant opportunity loss. If a branch is incorrectly predicted, a considerable number of cycles is wasted executing instructions on the wrong path, as well as recovering from such incorrect execution. Current microprocessors have large branch misprediction penalties. For instance, the newest Intel microprocessors, the Intel Sandy Bridge proces-
1.1. Motivation

Sensors, are reported to have a misprediction penalty of 14 cycles and a fetch width of four instructions per cycle [1]. Since younger instructions in the pipeline must be flushed on a misprediction, for these processors, each mispredicted branch introduces 56 bubbles into the pipeline. If 10 out of every 1000 instructions are mispredicted branches, on average 560 out of every 1560 instructions fetched are wasted executing wrong-path instructions, wasting more than a third of the fetch bandwidth. Although based on many simplified assumptions, this example illustrates that due to the large penalties associated with branch mispredictions, small improvements in accuracy can have a large impact on performance.

To predict a branch outcome, the branch address and some additional information are used to lookup predictions in one or more tables of counters. These tables store information about the past behaviour of branches. Since the tables have a limited number of entries, some branches will inadvertently be forced to use the same entries. This phenomenon is known as aliasing, which can, in principle, be constructive or destructive. However, aliasing has been shown to be predominantly destructive, significantly limiting the accuracy of branch predictors [2]. A variety of techniques have been proposed to limit aliasing and reduce its negative effects [3, 4, 5, 6, 7].

Increasing the branch predictor table size(s) is a simple way of reducing aliasing. However, larger tables require more area and have longer access latencies. As the branch predictor resides on the critical path of instruction fetch, it is desirable that it provides a prediction within a single cycle. Therefore, branch prediction designs are constrained not only by their chip area budget, but also by the delay to access their structures. It has been shown that increasing delay to improve prediction accuracy is almost never a good tradeoff [8]. This effect can be explained with the following equation, which roughly approximates the cost $C$ (in cycles) of executing a branch instruction:

$$C = d + (r \times p)$$  \hspace{1cm} (1.1)

where $d$ is the delay of the branch predictor, $r$ is the misprediction rate, and $p$ is the misprediction penalty. Because misprediction rates tend to be close to a few percent, changes
1.1. Motivation

in delay have a larger impact than small changes in the misprediction rate. For this reason, we are motivated to find methods that allow branch predictors to use larger tables without sacrificing the predictor access delay.

One such potential method is Predictor Virtualization (PV) [9]. PV can be applied to hardware optimization engines (known as predictors in PV terminology) that collect information about program behaviour at runtime and store it in on-chip tables. PV uses dedicated on-chip resources to record the active predictor entries. Whenever this dedicated storage is not enough, PV spills entry contents to the memory hierarchy. PV attempts to emulate large predictor tables without requiring large amounts of dedicated on-chip resources to be set aside at design time. PV offers the ability to dynamically adapt predictor table sizes based on the observed application behaviour, as well as, allowing the use of a feedback mechanism to turn the optimization on and off without wasting storage.

Applying Predictor Virtualization to branch outcome predictors could potentially be used to provide the best balance between predictor accuracy, area overhead, and latency. Augmenting the dedicated branch predictor tables with a large virtual second-level table is beneficial, not only because the second-level table does not come at the cost of larger dedicated storage, but also because, if orchestrated properly, the delay of the predictor can remain constant at the latency of accessing the smaller dedicated tables.

The main challenge with Predictor Virtualization is tolerating the variable and relatively long access latency introduced by placing the virtual storage in the memory hierarchy. Some predictors naturally benefit from virtualization as they can inherently tolerate higher latencies, because they themselves perform high-latency operations [9]. However, a straightforward application of PV techniques is not feasible for branch predictors, since branch predictors are delay sensitive. PV has also been successfully applied to branch target buffers, a predictor class that is not inherently tolerant of higher prediction latencies. For these predictors, the inherent spatial and temporal locality of their access stream is used to prefetch predictor entries, hiding the latency of virtual table accesses [10]. These predictors use branch addresses to access their structures, which inherit the spatial and
temporal locality present in program instructions. However, branch predictors’ access streams do not exhibit spatial or temporal locality since they use a hash of the branch address and other information to index the predictor tables. These hash functions are designed to randomize the predictor access stream as much as possible, in order to reduce aliasing. To take advantage of PV, the design of branch predictors will have to be revisited with virtualization in mind.

This thesis explores virtualization techniques as applied to branch outcome predictors. As most branch predictors were designed with area constraints in mind, it is useful to rethink branch predictor design in the absence of area constraints and in the context of virtualization; Virtualization makes area constraints a secondary factor by providing the predictor with a large virtual space. In this work, virtualization is applied to a state-of-the-art multi-table branch predictor, the TAged GEometric History Length predictor (TAGE) \[11\]. Our designs augment a single table in TAGE with a large virtual second-level table. We propose two schemes for the design of such a two-level predictor, a paging scheme and a packaging scheme.

1.2 Thesis Contributions

The contributions of this work are as follows:

- We propose a novel paging scheme for branch outcome prediction virtualization that introduces coarse-grain locality in the branch prediction access stream. In this scheme, the entries in a single predictor table are divided into pages shared by a group of branches in close proximity in the program code space. The first-level table caching the most recently used subset of the second-level pages (Chapter 6).

- We evaluate the paging scheme using instruction count as proxy for timing, on a set of commercial workloads. For a predictor with moderate table sizes (2K entries), prediction accuracy can be improved by 10.6% to 30.5%, depending on the number of dedicated tables, while adding an overhead of only 704 bits of extra dedicated stor-
1.2. Thesis Contributions

Alternatively, the virtualized design can be used to reduce the dedicated budget and complexity of the predictor without affecting accuracy. In this case, depending on the number of dedicated predictor tables, 20% to 50% of the tagged tables can be removed, while achieving same or better accuracy than a non-virtualized design.

For a predictor whose size is determined not by area constraints but by the delay to access its contents (delay-constrained), the accuracy of the best configuration of the baseline predictor can be improved by 8.7%. Alternatively, virtualization can be used to achieve the same accuracy as the baseline predictor while removing approximately 25% of the dedicated storage.

- As an alternative to paging, we propose a novel packaging scheme for branch outcome prediction virtualization. In this scheme, the evicted entries from the first-level predictor table are grouped into packages that are placed in the second-level table, and fetched in time to be used for prediction (Chapter 7).

- Before considering the effect of L2 delay on the predictor, we demonstrate that the packaging scheme achieves better accuracy (up to 9% more) than the paging scheme, especially for a predictor with fewer dedicated tables. However, the packaging scheme exhibits higher traffic to the L2 compared with the paging scheme. We leave investigation into the effect of L2 latency on the packaging scheme for future work.

The rest of this thesis is structured as follows. We present a background on branch prediction and Predictor Virtualization in Chapter 2. We review related work in Chapter 3. We describe the methodology in Chapter 4. Chapter 5 provides an overview of the two virtualization schemes. In Chapter 6, we present and evaluate the paging scheme. In Chapter 7, we present the packaging scheme and show the potential improvement that can be achieved with this scheme before considering the effects of latency. We conclude in Chapter 8.
Chapter 2

Background

2.1 Background on Branch Prediction

Virtually all modern processors contain a hardware branch prediction unit. The unit is generally organized as a branch outcome predictor and a branch target buffer (BTB). It is placed in the instruction fetch stages and is in charge of recognizing an incoming branch instruction, providing a prediction about the outcome of the branch (taken or not taken), and providing the target instruction address (in case of a predicted taken branch). The target of the branch is predicted by the BTB. The BTB is a cache-like structure, indexed by a portion of the branch address, that keeps branch target addresses as its entries. Each entry also typically includes a tag field. The branch outcome predictor can be coupled with the BTB or implemented as a separate structure. In the former case, only branches that hit in the BTB are predicted by the outcome predictor, while a static prediction algorithm is used on a BTB miss. Static prediction can be as simple as an always-predict-taken (or -not-taken) method or could rely on more sophisticated compiler techniques that are based on the behaviour of branches predictable at compile time [12]. As an alternative to static prediction, dynamic branch prediction can be used to predict branch outcomes by the hardware at execution time.

In this thesis, we focus on dynamic branch outcome prediction and use the term branch prediction to refer to this type of prediction from this point on. Although branch prediction encompasses the prediction of both the direction and target of branches, we will explicitly use the term branch target prediction when referring to the latter. To help put our work into context, this section reviews background on branch prediction schemes. We present
2.1. Background on Branch Prediction

a general overview of the evolution of branch predictors, while focusing on the branch prediction schemes that directly relate to our work.

Simple dynamic branch predictors were first proposed by Smith [13]. The main idea behind these schemes is to use the past behaviour of the branch to speculate about its future outcome. A hash of the branch address is used as an index into a table of counters that are incremented when a branch is taken, and decremented otherwise. Prediction comes from the high bit of the corresponding counter; 1 means predict taken, 0 means predict not taken.

In 1991, Yeh and Patt observed that the outcome of a given branch is often highly correlated with the outcomes of other recent branches [14]. They use the pattern formed by the history of branch outcomes to provide a dynamic context for prediction. As each branch outcome becomes known, one bit (1 for taken, 0 for not taken) is shifted into a pattern history register. A pattern history register is accessed by a hash of the branch address, and is in turn used to access a pattern history table of 2-bit saturating counters, resulting in a two-level predictor. In a later work, the authors provide a taxanomy of two-level adaptive branch prediction schemes [15].

Subsequent work focused on refining the schemes of Yeh and Patt. Several predictors were proposed that used de-interference techniques to address the issue of destructive aliasing, which represented a significant cause of mispredictions [5, 6, 16]. Hybrid predictors that combined two branch prediction schemes were also proposed, taking advantage of the strengths of each prediction scheme to increase the overall accuracy [3]. These predictors form the basis of all publically disclosed implementations of branch predictors on real processors [17, 18, 19, 20].

In 2001, Jimenez et al. introduced the Perceptron predictor, which uses simple neural elements, known as \textit{perceptrons}, to perform branch prediction [21]. The predictor maintains the organization of previous two-level schemes, but replaces the table of counters with a table of perceptrons. While the Perceptron predictor achieved better accuracy than previous two-level adaptive methods by allowing the use of longer history lengths at the
cost of linear resource growth (traditional methods required an exponential growth of resources), the predictor suffered from high latency. Later research would improve both the latency of the predictor, as well as its prediction accuracy [22, 23, 24].

The Championship Branch Prediction competitions, first held in December 2004, resulted in a new wave of innovative branch predictors. Combining the summation and training approach of neural predictors with the idea of hashing into several tables of counters, the Optimized GEometric History Length (O-GEHL) branch predictor won second place at the first Championship Branch Prediction (CBP-1) [25]. The O-GEHL predictor uses a set of history lengths that form a geometric series to index a set of predictor tables. The PPM-Like Predictor, a multi-table tag-based scheme based on prediction by partial matching (PPM) was introduced at the same competition, and placed fifth [26]. At CBP-2, the L-TAGE branch predictor, composed of a loop predictor and a TAgged GEometric history length (TAGE) predictor, placed first [11]. The main component of the predictor, TAGE, was built upon the PPM-Like predictor, while using geometric history lengths as in the O-GEHL predictor to access the different tables. The winner at CBP-3, held recently in June 2011, was the ISL-TAGE predictor [27], which augments the previously proposed L-TAGE predictor with a Statistical Corrector predictor. At the time of this writing, TAGE-based predictors represent the state of the art in branch outcome prediction.

2.1.1 Branch Prediction Mechanisms

This section reviews the L-TAGE predictor. A brief description of several other predictors is provided in Appendix A.

L-TAGE

The L-TAGE predictor is composed of a loop predictor and a TAGE predictor. The loop predictor simply tries to identify regular loops with constant iteration counts. The loop predictor provides the final prediction when a loop has been executed three times with the same number of iterations. Each entry in the loop predictor consists of the maximum
2.1. Background on Branch Prediction

The iteration count determined in previous runs of the loop, the current iteration count, a partial tag used to identify the loop branch, a confidence counter, and an age counter. The loop predictor is trained to recognize the maximum loop count. When the current iteration counter reaches the maximum loop count, it predicts the branch as not-taken.

The rest of this section focuses on the main predictor component, the TAGE predictor. The virtualization techniques described in this work are applied only to the TAGE component of the predictor.

Organization Figure 2.1 shows the organization of the TAGE predictor. The predictor consists of a base bimodal predictor and a set of (partially) tagged predictor tables. Each tagged table is indexed through an independent function of global and path histories and the branch address. Similar to the GEHL predictor, the global history lengths used by the tagged tables form a geometric series. The base predictor is in charge of providing a default prediction and is implemented as a simple PC-indexed 2-bit counter bimodal table, which shares a hysteresis bit among several counters to reduce storage [19]. An entry in the tagged components consists of a 3-bit counter that provides the prediction, an unsigned 2-bit useful counter \(u\), and a tag. The width of the tag varies for each table.

Index and Tag Hash Functions The index to each tagged table is a hash of the branch address, global and path histories, and the table number. The tag is a function of the global history and branch address only.

To allow the use of long history lengths, a folding mechanism is used to compress the history from its original length down to the number of bits used to index the table. For example, for a table of 1024 entries (which uses a 10-bit index), 30 bits of global history are folded as follows: \(h[9 : 0] \oplus h[19 : 10] \oplus h[29 : 20]\) down to 10 bits. The folding mechanism is implemented as a circular shift register and a few XOR gates [26].

Prediction At prediction time, all tables are accessed simultaneously to provide a prediction. The base predictor always provides a default prediction, while the tagged com-
2.1. Background on Branch Prediction

Figure 2.1: TAGE Predictor with N Tagged Tables

ponents provide a prediction only on a tag match. If there is no matching tag on any of the tables, the default prediction is used. Otherwise, two predictions from the two longest history length tables with a tag match (or bimodal, if there is only one tag match) are considered. The second prediction is called the alternate prediction. In general, the first prediction is used unless the entry is deemed as newly allocated. An entry is considered newly allocated, if its prediction counter is weak, or if a global 4-bit counter called use_alt, used to monitor the effectiveness of the alternative prediction, indicates so. This counter is updated whenever the alternate prediction differs from the main prediction: if the alternate prediction is correct, the counter is incremented; otherwise, it is decremented.

**Update** At update time, several steps are taken as follows:

- The prediction counter of the component providing the final prediction is updated.
  
  When the useful counter of the provider component is 0, the alternate prediction is also updated.

- The \( u \) counter of the provider is updated when the alternate prediction is different
2.2. Background on Predictor Virtualization

![Phantom-BTB Architecture](image)

Figure 2.2: Phantom-BTB Architecture [10]

from the final prediction.

- The use_alt counter is updated whenever the alternate prediction differs from the main prediction. If the alternate prediction is correct, the counter is incremented; otherwise, it is decremented.

- On a misprediction, a new entry is allocated on a longer history length table. Using a pseudo-random number generator, one of the three next tables is chosen with probabilities 1/2, 1/4, and 1/4, respectively. This randomness in the allocation policy is used to reduce the impact of aliasing. An allocated entry is initialized with the prediction counter set to weakly taken or not-taken as determined by the outcome of the branch. Counter $u$ is initialized to 0 (i.e. strongly not useful).

2.2 Background on Predictor Virtualization

Predictor Virtualization (PV) is a technique that can be applied to hardware optimizations that gather information about program behaviour at runtime and store it in on-chip buffers or lookup tables for future use. PV uses the memory hierarchy to transparently store predictor metadata, while using the dedicated on-chip resources to record the active entries of the predictor. Conceptually, PV allows for emulating large predictor tables without the cost of large amounts of dedicated on-chip resources.

This section reviews PV in the context of a virtualized BTB design, Phantom-BTB [10]. Similar to branch outcome prediction, BTB prediction is latency-sensitive; therefore, a
straightforward application of PV techniques is ineffective in both cases. The original PV study demonstrated its applicability to a data prefetcher, which proved inherently tolerant of the high prediction latency introduced by virtualization. Instead, Phantom-BTB takes a different approach by using a prefetch-based virtualized design as described next.

In Phantom-BTB, a first-level conventional BTB is augmented with a large second-level virtual table. While the first level uses dedicated storage and operates the same way as a conventional BTB, there is no dedicated storage for the second-level table. Instead, the second-level table entries are transparently allocated and stored on demand, at cache line granularity, in the L2 cache. Figure 2.2 shows the design of the Phantom-BTB predictor.

To compensate for the high latency of the virtual table, the virtual table is designed to facilitate the timely prefetching of the BTB entries. The virtual table stores groups of temporally correlated branches that have missed in the first-level BTB. As branches miss in the dedicated BTB, they are grouped into temporal groups. Each temporal group represents a virtual table entry that maps to a single L2 cache line. A temporal group is associated with a prefetch trigger: a region of code surrounding the branch previous to the group that also missed in the BTB. Any subsequent misses for a branch within this region triggers the prefetch of the temporal group.

BTB misses trigger prefetching of their associated temporal blocks into a prefetch buffer co-located with the dedicated BTB. The prefetch buffer and the BTB are accessed in parallel on each branch access. On a BTB miss and prefetch buffer hit, the corresponding branch is installed in the dedicated BTB, thus, increasing its perceived capacity.
Chapter 3

Related Work

3.1 Related Work on PV

Predictor Virtualization was first introduced as a generic framework that allows the existing memory hierarchy to be used to emulate large predictor tables [9]. The main goal of PV in this context was to reduce the amount of dedicated on-chip resources required for hardware predictor-based optimizations, while preserving their original effectiveness. In the same paper, the authors describe an application of PV techniques to a state-of-the-art data prefetcher, which is inherently tolerant of the higher prediction latency introduced by virtualization. The virtualized prefetcher matches the performance of the original scheme, while reducing the dedicated on-chip resources from 60 KB down to less than 1 KB.

A later work by the same authors applies virtualization techniques to a Branch Target Buffer, as described in Section 2.2 [10]. Although the design shares the same goals as PV, the inherent latency sensitivity of the BTB necessitates a prefetch-based virtualized design that relies on the temporal correlation in the BTB miss stream. The virtualized design improves IPC performance by an average of 6.9% over a conventional 1K-entry BTB, while adding only 8% extra storage overhead. The new design performs within 1% of a 4K-entry conventional BTB, while requiring 3.6 times less dedicated storage.

Finally, virtualization has been applied to the EXplicit dynamic branch predictor with ACTive updates (EXACT) [28]. EXACT employs a different approach to improving branch outcome prediction, based on the observation that global history alone can fail to distinguish between dynamic instances of branches. Instead, a combination of the branch address and the address of the load instruction on which the branch depends is used to
3.2 Related Work on Hierarchical Branch Predictors

uniquely identify the branch. A store to an address on which a dynamic branch depends may also change its outcome the next time it is encountered. As a result, such store instructions are allowed to actively update the prediction counter in the predictor. The branch predictor is augmented with two structures in charge of keeping track of the effect of load and store instructions. Virtualization is applied to the latter structure, the Active Update unit, which takes the address of a store instruction as index and outputs the branch address that is affected by the store, as well as the effects of the change. Since active updates are tolerant of 400+ cycles of latency, a straightforward application of PV is used on the Active Update unit. In the virtualized design, a 10KB dedicated first-level table is augmented with a 512KB virtual level-two table.

3.2 Related Work on Hierarchical Branch Predictors

Closely related to our work are delay-sensitive hierarchical branch predictors [8]. These predictors share the same goal as our work: to allow the use of a larger more accurate predictor without increasing prediction delay to more than a single cycle. These predictors were introduced in the context of addressing the problem of scaling branch predictors to future technologies. Although the aggressive clock rates assumed in the original study (~8GHz at 35nm) have not been realized to date, the conservative clock rates (~5GHz at 35nm) are close to what can be achieved at the time of this writing. The problem is only significant at the aggressive clock rate, at which point a table of only 512 entries can be accessed within one cycle. At the conservative clock rate, 16 K entries are still accessible within a cycle. Although the proposed designs are geared toward addressing the aforementioned problem, they represent valid design options for hiding the latency of large predictor tables. The paper introduces three schemes for overcoming delay: a caching approach, an overriding approach, and a cascading lookahead approach applied to the gshare branch predictor.

In the caching scheme, a small cache of branch prediction table entries (called PHTC) that is accessible in a single cycle, is backed by a larger second-level prediction table (PHT).
At prediction time, both tables are accessed in parallel. If the correct entry is found in the PHTC, a prediction can be made immediately. Otherwise, a third table called the Auxiliary Branch Predictor (ABP) is used to provide a prediction, so that the delay of the second-level table does not stall instruction fetch. Once the prediction is retrieved from the PHT, it replaces an entry in the PHTC.

The second approach combines the concepts of lookahead branch prediction [29] and cascading branch predictors [30]. In lookahead branch prediction, the natural spacing between branches is used to perform prediction for the next branch that is likely to arrive. For example, if branches are spaced so that the predictor is accessed only every other cycle, then the predictor can have a two-cycle latency without introducing additional delay. A cascading branch predictor implements a series of tables of ascending size and accuracy. In their scheme, the gshare predictor is arranged in a two-level cascading organization and is adapted to look one branch ahead. Prediction is begun simultaneously on both levels of the predictor. The second-level table is used when the next branch to be predicted is several cycles away, so that the access latency of the table fits in the space between the current prediction and the next. If the next branch arrives before the second level table can complete its access, then the prediction from the first table is used.

Finally, an overriding branch predictor provides two predictions. The first prediction comes from a fast PHT (PHT1) and the second prediction comes from a slower, but more accurate PHT (PHT2). To predict a branch, the first prediction is used and acted upon while the second prediction is still being made. If the second prediction differs from the first, the actions based on the first prediction are reversed and instructions are fetched using the second prediction; thus, the second predictor overrides the first predictor. A similar technique is used in the Alpha 21264 branch predictor [18], in which a branch predictor with a latency of 2 cycles overrides a less accurate instruction cache line predictor at the cost of a single stall cycle.

The last two schemes could be considered as an alternative to virtualization. However, in both cases the accuracy of the predictor depends on the accuracy of both the fast and
3.3 Related Work on Improving TAGE Prediction Accuracy

less accurate predictor and the larger more accurate one. Theoretically, these schemes could benefit from using a virtualized predictor (which can provide a prediction in a single cycle) as the first-level predictor. More importantly, unlike a virtualized design, these two schemes require large dedicated budgets for implementing the two predictor levels.

These predictors could also be considered as candidates for virtualization. However, the design of both the cascading lookahead predictor and the overriding scheme assume a latency of only a few cycles for the second-level table (2-4 cycles). With the latency of the L2 cache at more than 10 cycles, implementing their second-level table as part of the L2 cache would force the predictors to rely almost exclusively on the first-level predictor. Of the three schemes, the caching approach is closest to our virtualized design. The crucial difference is that in our design, an entire block of correlated entries is retrieved on a first-level table miss. In comparison, in the caching approach, only a single predictor entry is retrieved on a miss. The reported results indicate that fetching only a single entry from the second-level cache is insufficient, as the table is used for 7.5% of all branches, and this access results in a prediction different from the auxiliary predictor in 0.013% of all branches [8]. Therefore, this approach almost always relies on the auxiliary predictor rather than the second-level table, and when it does not the two predictions almost always agree.

3.3 Related Work on Improving TAGE Prediction Accuracy

Other predictors, such as ISL-TAGE [27] and EXACT [28], have been proposed to improve the accuracy of the TAGE predictor. These predictors augment TAGE with new structures that are geared toward predicting branches that cannot easily be predicted with the TAGE predictor. The TAGE predictor is apt at predicting branches that are highly correlated with the behaviour of other branches. By using a geometric set of history lengths, it efficiently captures correlation on recent branch outcomes as well as on very old branches. Our work is orthogonal to these works, since virtualization is applied to the core branch predictor structure, in order to increase its perceived capacity and reduce destructive aliasing.
Chapter 4

Methodology

This chapter explains the experimental methodology used in this work. The simulation infrastructure is described, followed by a summary of the parameters of the baseline predictor that we compare our virtualized designs to.

4.1 Simulation Infrastructure

To evaluate the designs explored in this work, we run simulations on the functional simulation infrastructure provided for the second Championship Branch Prediction Competition (CBP-2) [31]. The results are measured using functional simulation of the first two billion instructions of a set of three commercial workloads. For each of the workloads, a trace of branch addresses and outcomes is collected for a run of the first two billion instructions on Flexus, a full-system simulator based on Simics [32]. The simulator is configured as a functional in-order uniprocessor running the UltraSPARC III ISA.

4.1.1 Timing Model

In Section [6.3.6] we investigate the effect of the second-level predictor table latency on the overall prediction accuracy using a simple timing model. In this model, we use instruction count to approximate processor cycles and to emulate the second-level table delay. This model uses a different trace format that includes the number of instructions in the basic blocks encountered in the program, followed by the address and outcome of the branch at the end of the basic block. Although both sets of traces were collected for the same phase of the benchmarks, due to the move from Simics 2 to Simics 3, the traces differ slightly and so do the measurements of the related data. The presented results use these timing traces
4.1. Simulation Infrastructure

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Online Transaction Processing (OLTP)</strong></td>
<td></td>
</tr>
<tr>
<td>TPC-C</td>
<td>100 warehouses (10GB), 16 clients, 1.4 GB SGA</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Decision Support (DSS)</strong></td>
<td></td>
</tr>
<tr>
<td>TPC-H</td>
<td>Throughput oriented, 4 queries, 450 MB buffer pool</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Web Server (WEB)</strong></td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>16K connections, Fast CGI, worker thread modeling</td>
</tr>
</tbody>
</table>

Table 4.1: Workload Descriptions

Starting from Section 6.3.5.

To emulate the second-level predictor table delay, we employ a request queue model. We model the second-level table as having separate read and write ports, and manage a queue of requests initiated by the branch predictor for each of these ports (we do not model regular requests to the L2 cache). We service at most one read and one write request per cycle. We assume that the second-level table is pinned in the L2 cache. Therefore, each request contains the requested cache line index, in addition to being stamped with the cycle number in which the request is made. For write requests, a snapshot of the data at the time the request is made is also passed along with the request. To simulate a latency of \(N\) cycles, a request made at time \(t\) is serviced at the beginning of cycle \(t + N\). The simulator fetches four instructions per cycle (fetch width of four). We place a constraint that only a single conditional branch can exist in the fetch width, as we assume that the dedicated predictor tables have a single read port. Since each branch has single a delay slot for the SPARC ISA, there can be a maximum of two branches in a fetch width of four.

4.1.2 Benchmarks

Table 4.1 lists the simulated workloads, which include: the TPC-C v.3.0 online transaction processing (OLTP) workload running on IBM DB2 v8 ESE, the TPC-H workload running on IBM DB2 v8 ESE representing a decision support system (DSS), and finally representing a conventional web-server (WEB) workload, the SPECweb99 benchmark running over Apache HTTP Server v2.0. The web server is driven with separate simulated client systems, the results present the server activity.
We originally intended to use the Simplescalar simulation tool for our work, which has both functional and timing models [33]. However, since these commercial workloads could not be simulated on Simplescalar, we switched to the CBP infrastructure (Section 4.1). In Section 7.3.6, we present some results gathered prior to our move to the new infrastructure using the SPEC CPU 2000 benchmarks [34]. These benchmarks were simulated for a run of 1 billion instructions on the functional sim-bpred simulator of Simplescalar.

### 4.1.3 Branch Prediction Accuracy Metric

The primary metric used in this work to measure prediction accuracy is *mispredictions per kilo instructions* (MPKI). This metric is a better measure of prediction accuracy than misprediction rate, as it includes information about both the frequency of branches in the benchmark, as well as the predictor’s ability to correctly predict these branches. MPKI is also more representative of the performance impact of the predictor accuracy. For example, a poor misprediction rate on a program with few branches may not have a significant impact on performance.

### 4.1.4 Simulated Microprocessor

The only two processor parameters outside the branch prediction unit considered in this work are the L2 cache line size and latency, which we assume to be 64 bytes and 20 cycles respectively. All virtualized designs described in this work use a 64 KB second-level predictor, equivalent to 1K L2 cache lines.

### 4.2 TAGE Parameters

In this section, we summarize the parameters used for the TAGE predictor used in our experiments. We have used the parameters of the original TAGE implementation [11], unless otherwise specified.
4.2. TAGE Parameters

<table>
<thead>
<tr>
<th>Table</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag Width (bits)</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 4.2: Tag Width for Predictor Table Entries

<table>
<thead>
<tr>
<th>N</th>
<th>Geometric Function</th>
<th>Min/Max History Length (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-5</td>
<td>(L_i = L_1 \times 2^{i-1})</td>
<td>(L_1 = 8)</td>
</tr>
<tr>
<td>6-12</td>
<td>(L_i = h_{min} \times \left(\frac{h_{max}}{h_{min}}\right)^{\left(\frac{i}{N-1}\right)} + 0.5)</td>
<td>(h_{min} = 4, h_{max} = 640)</td>
</tr>
</tbody>
</table>

Table 4.3: Geometric Functions Used for History Lengths

4.2.1 Number of Tagged Tables

The original TAGE contains 12 tagged tables. In our experiments, we vary the number of tagged tables \((N)\), from 1 to 12. We do this since one of our goals is to use virtualization to reduce the dedicated budget and complexity of the predictor. In this way, we can determine if a non-virtualized design can be replaced by a virtualized design with fewer tables but equivalent accuracy.

We use the notation \(TAGE_{N,M}\) to refer to a TAGE with \(N\) tagged tables each with \(M\) entries. In the rest of this thesis, we use the terms table and component interchangeably when referring to TAGE’s tagged tables.

4.2.2 Entry Sizes

The entry sizes remain unchanged from the original predictor. The prediction and useful counters are 3 and 2 bits respectively. Tag width varies for each table, and is summarized in Table 4.2.

4.2.3 History Lengths

The original geometric function used for calculating history lengths takes a minimum and maximum history length, and computes a history length for each predictor table depending on the total number of tables. The minimum and maximum history lengths used in the original TAGE are 4 and 640 bits respectively. This allows a wide range of history lengths for a predictor with a reasonable number of tables. For a predictor with fewer tables, how-
ever, this results in very short and very long history lengths with few moderate values in between. To allow a better range of history lengths for smaller $N$, we tested the predictor with a simple geometric function: $L_i = L_1 \times 2^{i-1}$. A better accuracy was achieved with the new geometric function for up to five tagged components. Therefore, we use two sets of geometric functions: the new function for $N \leq 5$, and the original for $N > 5$ (Table 4.3).  

### 4.2.4 Table Sizes

For our baseline predictor, we allow the predictor capacity to be set by constraints on the predictor access time rather than any chip area limitations. The predictor is allowed as much capacity as it can access to make a prediction in a single cycle. We use the CACTI cache modeling tool [35] to estimate the latency of the predictor structures and choose the largest configuration that can provide a prediction within a single cycle. Specifically, we use CACTI to determine the number of entries on the predictor tables.

For a predictor with $N$ tagged components, the prediction time roughly consists of the time to compute the index/tag hash functions, access the predictor table, and perform a tag match (performed in parallel for all the tables). This is followed by the time to select the longest matching history length prediction and the alternate prediction, taking into account the confidence of each prediction and the global use_alt counter, as described in Section 2.1.1. To estimate the branch predictor access time, we optimistically assume that delay is dominated by the latency to access the tagged components, perform tag matching, and choose the longest history table with a tag match. In this way, we can model a predictor with $N$ tagged components as an $N$-way set-associative cache and estimate its access delay using CACTI. We use CACTI’s pure RAM interface (with no tags) to determine the maximum size of the bimodal table separately.

Figure 4.1 shows the basic logical structure of a uniform cache access organization in CACTI. CACTI uses separate arrays to model the tag and data in a cache. The address request to the cache is first provided as input to the decoder, which then activates a wordline in each of the tag and data arrays. The contents of an entire row are placed on the bitlines,
which are then sensed. The multiple tags thus read out of the tag array are compared against the input address to detect if one of the ways of the set does contain the requested data. This comparator logic drives the multiplexor that finally forwards at most one of the ways read out of the data array back to the requesting component.

This model of cache delay parallels our simplified model of the predictor access time. The delay for both models consists of the time to access the tag array, perform tag matching, and choose the final data among $N$ different options. There are two differences between an $N$-way cache and the simplified model of the predictor. First, the comparison logic for the two differs slightly. For a cache, a single tag hit must be determined, whereas for TAGE, the tables have associated priorities. As a result, even in the presence of multiple tag hits the longest history length table must be chosen. Second, in a set associative cache, the delay of the structure consists of the time to drive an entire row of $N$ tags. Since TAGE is organized as separate tables, each table is accessed separately, reading out a single tag.

Figure 4.2 shows the access latency of an $N$-way set-associative cache with up to 16 ways (in powers of 2 as allowed by CACTI), while varying table sizes from 1K up to 32K. Each entry is composed of a 5-bit tag (the minimum tag width considered in our experiments) and 8 bits of data. The actual size of the data portion in the predictor is 5 bits. However, CACTI places a limit on the minimum size of the data. Although not shown, we ensure that the reported access time is dominated by the tag array access time.
4.2. TAGE Parameters

Figure 4.2: Access Latency for TAGE Modeled as an N-Way Set-Associative Cache (Tag Width: 5 bits, Data Width: 8 bits)

Figure 4.3: Access Latency for the Bimodal Table Modeled as an SRAM Array of Bytes
4.2. TAGE Parameters

<table>
<thead>
<tr>
<th>Number of Tagged Tables (N)</th>
<th>Number of Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16K</td>
</tr>
<tr>
<td>2</td>
<td>8K</td>
</tr>
<tr>
<td>3-4</td>
<td>4K</td>
</tr>
<tr>
<td>5-8</td>
<td>2K</td>
</tr>
<tr>
<td>9-12</td>
<td>1K</td>
</tr>
</tbody>
</table>

Table 4.4: Delay-Constrained Table Size Configurations Established Using CACTI

We assume a processor with a 4 GHz clock rate in a process technology of 32 nm, resulting in a 250 ps clock cycle. Current high-end processors use approximately the same clock rate [36, 37]. To mitigate the effects of discrepancies introduced by our simplified model for the predictor access time, we allow delays within a 20% margin of the 250 ps cycle time, up to 300 ps. Based on Figure 4.2, Table 4.4 summarizes the table sizes that can be accessed in a single cycle for each $N$. We refer to these baseline configurations as delay-limited or delay-constrained, since the size of the predictor is determined not by area constraints, but by the delay to access the tables.

In Figure 4.3 the access latency of a pure SRAM array of 1B entries is used to determine the maximum size of a bimodal predictor that can be accessed within a single cycle. The figure shows that an array with a maximum of 16 K entries (for a total storage of 16 KB) can be accessed in a single cycle. We use this as the capacity of the bimodal table, allowing a maximum of 64K entries on this table.


Chapter 5

Virtualization Overview

This section overviews the presented virtualized designs. In our work, we abstract the design of a branch predictor that can be readily virtualized with the design of a two-level predictor where constraints are imposed so that the second level could be directly stored in the L2 cache. The second-level predictor table is not accessed directly by the processor's prediction mechanism. Instead, a separate mechanism is used to transfer data between the two levels. Data is transferred at the granularity of an L2 cache line. The latency of accessing data in the second level is equal to the delay of an L2 cache access. Without these constraints, the designs could be used to build traditional hierarchical branch predictors.

We propose two schemes for the design of such a two-level predictor: a paging scheme and a packaging scheme. For both schemes, virtualization is applied to a single predictor table. In Chapter 6, we describe the paging scheme. We present an experimental analysis of the different design parameters and evaluate the design in the presence of a large-delay second-level table. In Chapter 7, we present the packaging scheme, we describe the design and provide data to show the method's potential before considering the effects of the second-level table delay. Figures 5.1 and 5.2 show the high-level designs of the two schemes.

In the first scheme, the predictor table entries are grouped into pages shared by a group of branch instructions in close proximity in the program code space. Based on the current portion of the code, the corresponding page is fetched from the second level and is placed in the first-level table. The first-level table caches the most recently used subset of the second-level table pages. Effectively, this scheme creates multiple, virtual, mini prediction tables, where each block of instructions maps onto one of these tables.
In the second scheme, evicted entries from the first-level predictor table are grouped into packages that are placed in the second-level table. Similar to the paging method, a group of branch instructions spaced closely in the program share a package. Based on the current portion of the code, the correct package is placed in a buffer co-located with the first-level predictor table, and its data is used as part of the prediction mechanism.

The two methods can be viewed as increasing the perceived length (size) and width (associativity) of one of the predictor tables respectively. In the first scheme, the predictor
table’s perceived size is increased. The table is divided into chunks, a subset of which is cached in the dedicated predictor table. In the second scheme, packages augment the dedicated predictor table with extra positions for evicted entries. In this way, the table can be thought of as having multiple ways, and the entries in packages can be thought of as having been stored in virtual predictor table ways. In the paging scheme, the entries in the first level table are a subset of the second-level table, while the package entries in the packaging scheme supplement the dedicated table entries.
Chapter 6

Paging Scheme

6.1 Design

In the paging scheme, one of the tagged predictor tables is backed by a larger second-level table. The first-level table is accessible within a single cycle, while the second-level table has a larger latency. The two levels are divided into pages, while the other predictor components remain unchanged. The organization of entries in both levels remains the same as the original predictor. The first level table serves as a cache for the most recently accessed pages of the second level.

Branches are assigned specific pages on the predictor table. The instruction address space is conceptually divided into instruction blocks, and each instruction block corresponds to a page on the second-level predictor table. Branches in one instruction block use the corresponding page for making prediction and allocating new entries. The page number is the upper portion of the branch PC, and therefore, maintains the locality of the instruction stream at a coarser grain.

Prediction is performed as usual. The prediction mechanism is unaware of the existence of pages and is unaffected by the introduction of the second-level table. The only change to the prediction mechanism is the change in the index hash function for this component, so that each branch, based on its PC, accesses entries within a specific page. To make a prediction, only the first-level table is accessed directly. Updates are also made only on the first-level table. The second-level table is not accessed directly by the prediction mechanism. Instead, a virtualization engine swaps pages between the two levels as necessary.
6.2. Architecture

The virtualization engine is responsible for maintaining the correct pages on the first-level table and swapping pages between the two levels. For each branch, the engine determines the page number, and on a page miss, fetches the corresponding page from the second-level table. At the same time, the residing page, if dirty, is written back to the second level.

Since the prediction mechanism is unaware of the existence of pages, while waiting for the correct page to be fetched from the second-level table, a branch may inadvertently use data on a page different than its own. We will show in Section 6.3.4 that the entry’s tag provides a second layer of filtering, so that the effect of aliasing is insignificant in such cases. Similarly, a branch may allocate entries on a page different than its own. Conceptually, this allows repeating patterns to take advantage of entries allocated on other pages to make a prediction while waiting for the correct page.

6.2 Architecture

This section describes the architecture of the paged design. Our design adds a second-level table to the predictor and logic for swapping pages from the two levels of the predictor tables as necessary. We assume that the first level predictor table has ports to facilitate page swaps, separate from the original ports used for reading and updating it. Figure 6.1 shows the architecture of the paged design. We describe the components of the design next.

L1 and L2 tables A single predictor component is augmented with a large second-level table. We refer to the two levels of this predictor component as the L1 and L2 tables. We also refer to this component as a whole as long table, or ltable for short, since it appears virtually longer than the other predictor tables. The two levels are divided into pages. A page is the granularity at which data is swapped between the two tables, and in a virtualized design fits in a single cache block. The L1 table acts as a cache for the pages of the L2 that were most recently accessed. In caching terminology, the L2 is inclusive and writeback;
the L1 table is a subset of the L2, and data is written back to the second level at the time a new page is replaced in the L1.

**Virtualization Engine** The virtualization engine is responsible for maintaining pages in the two levels. The address of the branch is used to determine the page number, and the corresponding page is requested from the second-level table in case of a miss. On a page miss, the prediction is not stalled, as the prediction mechanism is unaware of the existence of pages. As a result, the paging mechanism is not on the critical path of making a prediction. In addition, pages are swapped to keep the first-level predictor table in sync with the current instruction block, regardless of whether ltable provides the final prediction or not. Consequently, the sequence of pages swapped is a characteristic of the application alone, and is independent of configuration of the predictor.

**Index Hash Function** The index hash function is modified so that each branch allocates and uses entries on a specific page. Figure 6.1 shows how the bits of the branch PC are used to compute the index hash function. In the figure, $p$ is defined as $\log_2$ of the page size, $b$ is defined as $\log_2$ of the instruction block size, $a$ is $\log_2$ of the number of pages that
fit in the L1, and \( n \) is \( \log_2 \) of the number of pages that fit in the L2 and determines the page number. The parameter \( X \) is used to vary the instruction block size as a function of the page size, such that \( b = p + X \). This variable is explained in more detail in Section 6.3.5.

For each branch, bits \([b + n - 1 : b]\) determine the corresponding page number. The page number is a function of the branch address only and preserves the locality of the branch stream at a coarser grain. The position of a page in the L1 table is determined by the lower \( a \) bits of the page number, bits \([b + a - 1 : b]\) of the branch address.

The index to the table is the \( a \) bits of the branch PC that determine the location of the corresponding page in the L1 table, concatenated with the page index (index within the page). The page index hash function parallels the original predictor hash function \([11]\), with minor modifications. The hash function uses the same information bits (PC, global and path histories) as the original hash function. If \( \log_2 \) of the number of entries on the table is \( t \), then the original hash function folds these bits to \( t \) bits. Instead in the new hash function, the information is compressed to \( p \) bits, the \( \log_2 \) of the page size.

**Auxiliary Table** The auxiliary table is in charge of keeping tags for the pages currently residing in the L1 table. The number of entries in this table is equal to the number of pages that fit in the ltable first-level table. Each entry consists of the page number of the page currently residing in the L1 table, as well as a single dirty bit, which indicates whether the page has been modified since it was fetched. Only dirty pages are written back to the L2. The auxiliary table has \( 2^a \) entries, where each entry consists of \( n \) bits for the page number tag and 1 bit for the dirty bit. Therefore, the auxiliary table represents an additional dedicated storage of \( 2^a(n + 1) \) bits. However, since we do not use the page numbers when making a prediction (we use them only to fetch pages from L2), the auxiliary table is not on the critical path of making a prediction.
6.3 Design Space Exploration

In this section, we describe the experiments used to explore the design space of the paged TAGE predictor and analyze the effects of each parameter. In order to allow these effects to be shown more clearly, we use a fixed size for all of the predictor tables (regardless of the number of tagged components), as opposed to using the delay-constrained configurations established using CACTI in Section 4.2.4. In Section 6.4, we apply the parameters determined in the design exploration phase to the delay-constrained configurations in order to show the benefits of virtualization.

In this section, the number of entries on the tagged components is fixed at 2K entries. This is the size determined by CACTI for the range of $5 \leq N \leq 8$. The reason for using this number is that the baseline TAGE$_{8,2K}$ predictor achieves the best MPKI of all the delay-constrained TAGE$_N$ configurations, as will be shown in Section 6.4. The size of the bimodal table is 16K entries, unchanged from the original TAGE predictor.

A summary of the rest of the section is presented here:

- In Section 6.3.1, we show that the accuracy of TAGE can be improved by increasing either the number of tables or the size of each table. This motivates the need for virtualization.

- In Section 6.3.2, we show that increasing the size of a single predictor component to 32 K entries can provide improved accuracy equivalent to doubling the size of all predictor components. We also determine the table that provides the best benefit from an increased capacity. This justifies our choice to virtualize one table only.

- In Section 6.3.3, we show the maximum potential improvement that can be achieved through virtualization of a single predictor table by implementing an idealized model of the predictor. This experiment establishes an upper bound on the accuracy improvement that we can expect.

- In Section 6.3.4, we evaluate the effect of restricting branches to a single mini-predictor
table page for various page sizes. We show that smaller page sizes have higher MPKI compared with larger pages, because of a decrease in the frequency of tag matches.

- In Section 6.3.5, we show that increasing the instruction block size can be used to reduce the page swap rate (traffic to and from the L2 table), but at the cost of increasing the MPKI.

- In Section 6.3.6, we examine the impact of the L2 table’s delay. We show that the paging scheme improves accuracy even with high latencies, such as 20 cycles. We present the accuracy of the final virtualized design, as well as the improvement it represents over the baseline predictor.

- Finally, in Section 6.3.7, we virtualize a predictor with double the initial table size as that used up to this point, in order to explore the sensitivity of the results to the CACTI estimates used for the initial table sizes. We show that although the potential for improvement is smaller in this case, virtualization is still beneficial.

### 6.3.1 TAGE Baseline and Potential for Virtualization

Figure 6.2 shows the average MPKI of the baseline predictor (TAGE$_{N,2K}$) as a function of $N$, the number of tagged tables. To show the potential for improvement as more capacity is allotted to the predictor tables, the figure also shows the average MPKI of TAGE$_{N,4K}$ and TAGE$_{N,8K}$.

The figure shows that the accuracy of the predictor can be improved both by increasing the table count and the table sizes. As more tables are added to the predictor, MPKI improves, but at a decreasing rate. Similarly, increasing the number of entries on the tagged components results in a better MPKI, but at a diminishing rate. The improvement achieved going from TAGE$_{N,2K}$ to TAGE$_{N,4K}$ is larger than going from TAGE$_{N,4K}$ to TAGE$_{N,8K}$, although the latter case requires a larger extra capacity (extra 4N K entries as opposed to an extra 2N K entries for the first case).
6.3. Design Space Exploration

6.3.2 Choosing $l_{table}$

Since the paging mechanism is applied only to a single predictor table, we need to ensure that increasing the perceived capacity of a single table provides enough improvement to justify a paged design. We also need to determine the table that would utilize the increased capacity most efficiently to improve accuracy. Intuitively, we expect one of the first few tables (that use shorter history lengths) to benefit most from an increased capacity. This is because the behaviour of most branches depends on the outcome of a few recent branches, while a smaller portion of branches depend on very long history lengths. In this section, we use an idealized model of the paged predictor to determine which table benefits most from an increased capacity. In the next section, we use these results to show the maximum potential gain of the paging scheme.

We model an idealized version of the predictor by replacing one of the predictor tables, $l_{table}$, with a large table equal in size to the L2 table. In this model, $l_{table}$ is not broken into pages. This represents an idealized version of the paged predictor, since a branch
can directly access any ltable entry as necessary, without any delay. The accuracy of this predictor represents an upper bound on the improvement that can be achieved with the paging scheme.

We set ltable size to 32 K entries to match the L2 size used in the rest of the experiments. In all of our experiments, we use an L2 table with 32 K entries, for a total storage of ~64 KB. The L2 size was chosen so that it is reasonably large compared to the L1.

Figure 6.3 shows the average MPKI of the idealized predictor for different values of \( N \). The x-axis shows which table is the ltable, while each line represents a different \( N \). The average MPKI of the baseline predictor (with no ltable) is also shown at \( N = 0 \) for comparison. In all cases, the first few tables benefit the most from an increased capacity. The best MPKI is roughly achieved with \( ltable = 2 \) for \( 2 \leq N \leq 6 \) (except for \( N = 5 \)), and \( ltable = 3 \) for \( N > 6 \). These results, which we refer to as the default ltable configuration, are summarized in Table 6.1 We use the default ltable configuration in the rest of this chapter.
### 6.3. Design Space Exploration

<table>
<thead>
<tr>
<th>Number of Tagged Tables (N)</th>
<th>ltable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 to 6</td>
<td>2</td>
</tr>
<tr>
<td>7 to 12</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6.1: Default Ltable Configuration

---

#### Figure 6.4: Maximum potential improvement achievable by adding a single 32K-entry table

### 6.3.3 Maximum Potential Improvement with the Paging Scheme

In this section, we use the idealized predictor with the default ltable configuration (Table 6.1) to show the upper bound on the potential improvement that can be achieved through the paging scheme. To assess the efficiency of using a single large predictor table, we compare this ideal improvement with the improvement that can be achieved by doubling the size of all predictor tables.

In Figure 6.4, two sets of plots are shown: The solid plots show the average MPKI of the baseline predictor ($TAGE_{N,2K}$), the idealized predictor, and the $TAGE_{N,4K}$ predictor.
The dashed plots show the percentage improvement in MPKI for the latter two predictors over the baseline (second y-axis).

The figure shows that the maximum potential improvement for the paging scheme is in the range of ~16% to ~44% for different $N$. As $N$ increases and the dedicated capacity of the predictor grows larger, the potential improvement decreases.

Next, we evaluate the efficiency of using a single large predictor table. The idealized predictor adds a fixed extra storage of 62 KB to a single predictor table. In comparison, the $TAGE_{N,4K}$ predictor adds an extra 2 K entries (~4 KB) to each tagged table, for a total of $4N$ KB. It can be observed from the figure that a single large table does not represent the most efficient use of storage. Although the idealized predictor achieves much better accuracy for smaller $N$, the accuracy of the two predictors starts to grow closer. At $N = 12$, the extra ~48 KB of $TAGE_{N,4K}$ spread across all the tables results in better accuracy than the extra ~62 KB added to a single predictor table for the idealized paged predictor.

Although the extra capacity is not used most efficiently, the margin of improvement is large enough to justify the choice of exploring paging mechanisms with a single L2 table. Using a single L2 table is less complex compared to designs that break the extra storage in the form of multiple L2 tables. We leave the exploration of such designs for future work.

### 6.3.4 Page Size

In the previous section, we showed the maximum improvement that can be achieved with the paging scheme in the absence of page size and delay constraints. In this section, we add page size constraints to the idealized predictor model by dividing the table into pages and restricting branches to access entries within a single page. The entries in the table can still be accessed directly without any delay. We evaluate the effect of page size on accuracy, as well as on the frequency of page swaps. We conclude the section by providing a detailed analysis of the behaviour of the predictor in the presence of page size constraints. In this section, the instruction block size is equal to the page size.
6.3. Design Space Exploration

Effect of Page Size on MPKI

In order to show the impact of page size on the predictor accuracy, we run experiments for ltable having a single page of 32 K entries up to 32 K pages of 1 entry each. Figure 6.5 shows the predictor’s MPKI as a function of the page size. Each line in the figure shows the MPKI for a different $N$. The figure shows that smaller page sizes have a higher MPKI than larger page sizes. However, the effect of page size on MPKI is minor for pages of 256 entries and higher. For smaller page sizes, the MPKI starts to grow at an increasing rate, and the effect is generally more pronounced for smaller $N$.

One reason that smaller page sizes adversely affect MPKI is because history bits are folded into fewer bits to form the page index. History serves two purposes: First, it carries information about the past behaviour of branches, and the fewer bits it is compressed to, the higher are the chances of information loss. Second, XORing the branch address with history to index a predictor table helps to allow even distribution of entries on the table. The fewer bits of history that are XORed with the branch address to form the index, the
6.3. Design Space Exploration

Figure 6.6: Maximum potential improvement with a page size of 32 entries

higher the chances of uneven distribution of entries on the table. Between the two effects, the second one is especially important, and is illustrated in the figure by comparing the predictor accuracy for 1-entry and 2-entry pages. In the case of 1-entry pages, a branch always maps to the same entry on the predictor table, based solely on its PC. For 2-entry pages, a branch is allowed to access one of two entries on the table. Since the tag width remains constant for all page sizes, the same amount of history information is embedded in each entry's tag for both cases. It can be seen in the figure that the MPKI for 1-entry pages is much worse than if only a single history bit was used in the index hash function, as in the case of 2-entry pages. For this reason, using too small a page size can have a detrimental effect on accuracy, even if the aggregate capacity of the predictor is large.
6.3. Design Space Exploration

<table>
<thead>
<tr>
<th>Number of Tagged Tables (N)</th>
<th>Ltable</th>
<th>Global History Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2 to 5</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 6.2: Ltable History Lengths

Choosing Page Size

To simplify the design of the virtualized predictor, we choose the page size such that it fits in a single 64 B cache line, the typical line size of L2 caches today. Since each entry is ~2 B, this corresponds to a 32-entry page size.

Figure 6.6 shows the average MPKI of the predictor with 32-entry pages. Also shown for comparison are the average MPKI of the baseline predictor \(TAGE_{N,2K}\) and the idealized predictor (with no page constraints, Section 6.3.3). The two dashed plots show the percentage improvement in the accuracy of the idealized predictor and the paged predictor over the baseline. The difference between the two lines represents the improvement loss due to page size constraints.

The figure shows that the accuracy of the paged predictor is close to the idealized predictor accuracy. The largest difference between the two designs is at \(N = 7\). One reason \(N = 7\) suffers most from paging constraints is because \(N = 7\) uses the longest history for ltable (Table 6.2).

In conclusion, index limitations imposed by page size constraints can have a large negative impact on accuracy, even for a predictor with tagged entries and with a large aggregate capacity. Tables with smaller history lengths make better candidates for virtualization, as the adverse effects of paging are smaller for shorter history lengths. Finally, the impact of dividing a single table into pages is less noticeable the more tables the predictor contains.
6.3. Design Space Exploration

**Effect of Page Size on Page Swap Frequency**

In this section, we examine the effect of page size on the page swap frequency. The page swap frequency determines the impact of virtualization on the traffic to the L2 cache. Increasing L2 traffic is undesirable for performance and power efficiency reasons. However, past work has shown that today’s L2 caches can absorb considerably higher traffic than that induced by typical applications [10]. We calculate the page swap frequency as the total number of page-ins to the L1 over the total number of branch predictions made (the page swap frequency has a unit of swaps per conditional branch).

Figure 6.7 shows the page swap frequency as a function of the page size for each benchmark, as well as on average. The page size is varied from 2 K entries to 16 entries per page. Larger page sizes are not shown since the L1 table is 2K entries, while smaller page sizes are not considered due to their high MPKI. The page swap rate is independent of $N$, since it is determined by the stream of branch addresses (and, consequently, pages) encountered in the program. The page swap rate is a property of the benchmark itself and is a function
of the entropy present in the branch address bits.

The figure shows that as page sizes become smaller, the swap rate first decreases and then starts to grow. Two opposing effects are at play: with smaller page sizes, more pages can fit in the L1 table, decreasing the need for a swap. This is the dominant effect up to a page size of 512 entries. On the other hand, as pages become smaller, more bits are used to determine the page number. These bits come from the lower portions of the PC, which generally have higher entropy, resulting in a higher entropy in the sequence of pages encountered.

At the page size of interest for our virtualized design (32-entry pages), the average page swap rate is 32.8%. This means that a page would have to be swapped approximately every three conditional branches. Lowering the swap rate is beneficial for two reasons: First, it reduces the impact of virtualization on the traffic to the L2 cache. Second, a design with a lower swap rate is potentially less likely to be affected by the L2 table's large latency. We address the issue of lowering the page swap frequency in Section 6.3.5.

Analysis of Effect of Page Size Constraints

In this section, we provide an analysis of the impact of paging on the predictor by examining the specific behaviour of ltable and its contribution to final prediction in the presence of page size constraints. The overall MPKI results presented in this section are the same as those in Figure 6.5 shown for each benchmark and at specific \( N \) to aid our analysis.

Figure 6.8 shows the breakdown of the MPKI contributed from each of the bimodal and the single tagged table at different page sizes, for \( N = 1 \). The graph also shows the percentage of branches for which the tagged component provides the final prediction, with the remainder of the predictions coming from the bimodal table.

The MPKI for a predictor is the sum of the MPKI contribution from each of the predictor tables. In addition, the MPKI of each predictor table and the table's misprediction rate are related by the following equation,
6.3. Design Space Exploration

\[ MPKI_i = \frac{B}{10 \times I} (C_i \times R_i) \]  \hspace{1cm} (6.1)

where \( MPKI_i \) is the MPKI contribution of table \( i \), \( C_i \) is the contribution rate of the table (in percent), \( R_i \) is the misprediction rate of the table (in percent), and \( B \) and \( I \) are the total number of conditional branches and instructions in the benchmark, respectively.

The contribution rate of a table is calculated as the number of times the table provides the final prediction (regardless of the outcome) over the total number of predictions made. In addition, for a single benchmark, this relation holds:

\[ \frac{MPKI_1}{MPKI_2} = \frac{C_1}{C_2} \times \frac{R_1}{R_2} \]  \hspace{1cm} (6.2)

Therefore, the relationship between the misprediction rate of a table at two different page sizes can be inferred from the graph, based on the relationship between the MPKIs and the contribution rates.

The figure shows that as pages get smaller, the overall MPKI starts growing compared with a non-paged design. However, the MPKI contribution of the tagged table starts to decrease, while the bimodal table’s MPKI contribution increases. At the same time, the contribution rate of the tagged component drops. This means that the decrease in the MPKI of the table is due to the drop in the contribution of the table in providing the final prediction (and not because of an increased prediction accuracy of this table, for example). Although not shown, the prediction accuracy of the table, which can be inferred from the graph according to Equation 6.2, stays relatively constant. Therefore, limiting a branch to the entries on a single page does not result in an increased level of aliasing on the tagged table. Instead, the predictor finds a tag match less frequently and, consequently, the contribution of the table to the final prediction drops. As this happens, the burden of providing the prediction falls on the bimodal table, which falls short of making a correct prediction.

This finding indicates that the index and tag hash functions in TAGE can identify
6.3. Design Space Exploration

branches well, since the increase in MPKI with smaller page sizes is not due to a poorer accuracy of the table itself. Other predictors, especially those with tagless entries, would not be as readily compatible with a paging scheme.

Figure 6.9 shows the same breakdown of MPKI contribution for \( N = 8 \). Similar to the previous graph, a breakdown of the MPKI contribution from each of the ltable and bimodal tables is shown. Also shown is the aggregate MPKI contribution from the 7 remaining tagged components. Again, a similar pattern is present: as the page size shrinks, the percentage of branches for which the ltable provides the final prediction decreases, resulting in a lower contribution to the overall MPKI. As this table fails to provide predictions, the responsibility of making a prediction falls on the other tagged tables, as well as the bimodal table. The increase in the overall MPKI is a result of the increase in the MPKI contribution of both the bimodal table and the remaining tagged tables. However, the combined increase coming from the remaining 7 tagged components is less than that coming from the bimodal table. In addition, the increase in the MPKI due to paging is not as severe as the previous case \( (N = 1) \), because the other tagged components are more apt at compensating for the decline in the contribution of ltable to the final prediction.

As a sidenote, an observation can be made regarding the contribution rates of ltable at different values of \( N \). With a single tagged component \( (N = 1) \), the percentage of branches for which ltable provides the final prediction starts at more than 50%. The contribution rate decreases with more than a single tagged component, down to approximately 22% for \( N = 8 \). The contribution rate of ltable is not only a function of the number of tagged components on the predictor, but also depends on the history length used for the table (Table 6.2). The contribution rate of each table depends on the number of branches that can be satisfactorily predicted with the history length assigned to the table.

The predictor tables do not contribute equally to the final prediction. A minimum of 40% of the predictions come from the bimodal table, as will be seen in Section 6.3.6. With the contribution rate of ltable at around 25%, every fourth branch is predicted by ltable on average. When considering timing effects, this means that there is on average a three
6.3. Design Space Exploration

Figure 6.8: Effect of page size on predictor tables’ MPKI contribution (N=1)

Figure 6.9: Effect of page size on predictor tables’ MPKI contribution (N=8)
branch gap between two predictions provided by ltable, allowing time for the correct page to be fetched from the second-level table. Section 6.3.6 provides more detail on how the contribution rate of ltable factors into the sensitivity of the predictor to L2 delay.

6.3.5 Instruction Block Size

In the previous section, we assumed that the instruction block size and the page size are equal. For our virtualized design, we chose a page size of 32 entries, and showed that for this page size, a page would have to be swapped into the L1 table once every three conditional branches.

In this section, we vary the instruction block size in order to lower the page swap frequency. We use the parameter $X$ (Figure 6.1) to vary the instruction block size as a function of the page size. We evaluate the effect of varying $X$ on page swap frequency as well as on the predictor accuracy. We comment on the L2 table utilization rate for different instruction block sizes, by looking at the fraction of pages that are never swapped into the L1. We will show that using too large an instruction block can lead to wasting a large portion of the L2.

The predictor model remains unchanged from the previous section.

Effect of Instruction Block Size on Page Swap Frequency

Figure 6.10 shows the average page swap frequency as a function of $X$, for three different page sizes. For a virtualized design, we are interested in 32-entry pages, but we also show the effect of $X$ on a larger (128-entry) and smaller (16-entry) page size for completeness.

For all three page sizes, a larger $X$ leads to a lower page swap frequency. As $X$ is increased, and more branches share a page, the page swap frequency drops, but at a slower rate. Therefore, a larger instruction block size can be used to reduce page swap frequency. Next, we evaluate the impact of using larger instruction blocks on the predictor’s MPKI.
6.3. Design Space Exploration

Figure 6.10: Effect of instruction block size on page swap frequency (instruction block size = $2^X \times$ page size)

**Effect of Instruction Block Size on MPKI**

Figure 6.11 shows the effect of varying $X$ on the predictor MPKI, for the three page sizes used in the previous experiment. The effect is shown on a predictor with $N = 12$ as a representative, although the same effect is present for other values of $N$.

The figure shows that for all three page sizes, a larger $X$ leads to a worse MPKI. As $X$ is increased, and more branches share a page, the MPKI grows at an increasing rate. Therefore, using a larger instruction block size to reduce page swap frequency comes at the cost of an increased MPKI.

Based on the two figures (Figures 6.10 and 6.11), smaller values of $X$ represent a better tradeoff in increasing MPKI for a lower page swap frequency, as they result in a larger drop in page swap frequency for a smaller increase in MPKI.
6.3. Design Space Exploration

Figure 6.11: Effect of instruction block size on MPKI (instruction block size = \(2^X \times\) page size, \(N=12\))

**Effect of Instruction Block Size on L2 Table Utilization**

In this section, we examine the effect of the instruction block size on the utilization of the L2 table pages. As \(X\) is increased and higher portions of the PC are used for the page number, some page numbers may not be encountered in the program run. Figure 6.12 shows the percentage of such pages as a function of \(X\), for the three page sizes used in the previous experiments. Similar to the page swap frequency, this number is a property of the benchmark and is independent of \(N\).

The figure shows that up to \(X = 3\), almost all pages are accessed for all three page sizes. As \(X\) is increased, the fraction of pages that are never accessed grows larger, especially for smaller page sizes. This decrease in page utilization for larger \(X\) values is one of the reasons for the MPKI increase observed in Figure 6.11. For a certain L2 size, larger instruction block sizes lead to a lower page swap rate, but a portion of the pages remain unused.

An alternative method to reduce the page swap frequency with less impact on the page utilization rate is to incorporate branch history in the page fetch mechanism. For example,
two bits of history can be XORed with the upper bits of the page number (still determined solely by the upper bits of the PC) to allow each page four dynamic instances. Page swaps are triggered by page number changes, but at fetch time, one of the four instances of the page is fetched based on the current history. This scheme has the same page swap rate as a scheme with no history, but a lower percentage of pages remain untouched. We leave investigation into this method for future work.

Choosing Instruction Block Size

For our virtualized design, we choose $X = 3$, resulting in an instruction block size eight times larger than the page size. This means that branches in a 256 B block of code share a single 32-entry page on the predictor table. Figures 6.10 and 6.11 showed that the first few values of $X$ represent the best tradeoff in MPKI versus page swap frequency. For $X = 3$, almost all pages are utilized, while the page swap frequency is cut down to a third of its original value for $X = 0$ (from $\sim 33\%$ to $\sim 11\%$).
Figure 6.13: Maximum potential improvement with an instruction block size of 256 B ($X = 3$)

Figure 6.13 shows the average MPKI of this predictor for different $N$. Also shown for comparison are the average MPKI of the baseline predictor ($TAGE_{N,2K}$) and the predictor with equal instruction block and page sizes ($X = 0$). The two dashed plots show the percentage improvement in the accuracy of the predictor with the two different instruction block sizes over the baseline. The difference between the two lines represents the improvement loss due to using a larger instruction block size.

The figure shows that using a 256 B instruction block size removes ~1%-3% of the improvement that can be achieved using a 32 B instruction block size. The improvement loss decreases with larger values of $N$.

In the remainder of this chapter, we use a page size of 32 entries and an instruction block size of 256 B, unless otherwise stated.
6.3. Design Space Exploration

6.3.6 L2 Latency

Up to this point, we have used a predictor model that allows access to the L2 table without any delay. In this section, we evaluate the effect of L2 latency on the accuracy of the paged predictor. The timing model described in Section 4.1.1 is used to emulate the effect of L2 delay.

In this section, we first demonstrate the effect of latency on the predictor accuracy. In order to measure the impact of the delay magnitude, we vary latency from 0 (no latency) to 20 cycles in increments of 5 cycles. We find that delay has a negative impact on accuracy; However, the impact is not proportional to the magnitude of the delay. Next, we show that the virtualized design improves accuracy even with the high latencies used to represent the L2 cache latency. We assume a latency of 20 cycles for the L2 cache (the access delay of a dedicated L2 table of the same size is estimated by CACTI as 2 cycles). In addition, we look at the L1 miss rate and page swap frequency for different latencies. We show that larger delay increases the L1 miss rate, but not the page swap frequency. We conclude the section by providing a detailed analysis of the behaviour of the predictor in the presence of delay constraints.

Effect of L2 Latency on MPKI

Figure 6.14 shows the paged predictor’s percentage improvement in MPKI over the baseline predictor (\(TAGE_{N,2K}\)) for different values of L2 latency, and \(N\). For this figure, we opted to show improvement values rather than MPKI, since the MPKI values were very close. The percentage improvement is a direct function of the MPKI.

The figure shows that increasing latency has a negative impact on accuracy improvement. The largest drop in improvement is between no latency and a latency of 5 cycles. Larger latencies result in worse improvement, but the drop is smaller than that of going from no latency to a 5-cycle latency. The results for the L2 cache latency (20 cycles) are summarized next.
6.3. Design Space Exploration

![Graph showing the effect of latency on percentage improvement in MPKI]

Figure 6.14: Effect of latency on percentage improvement in MPKI

<table>
<thead>
<tr>
<th>Page Size</th>
<th>32 Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Block Size</td>
<td>256 Bytes</td>
</tr>
<tr>
<td>L2 Size</td>
<td>1K Pages</td>
</tr>
<tr>
<td>L2 Latency</td>
<td>20 Cycles</td>
</tr>
</tbody>
</table>

Table 6.3: Paging Scheme Virtualized Design Configuration

Virtualized Design Accuracy and Improvement Over Baseline

In this section, we present MPKI results for the virtualized design and provide a comparison with the MPKI of the baseline predictor ($TAGE_{N,2K}$). In addition, in order to measure the improvement loss due to the L2 table latency, we compare the accuracy of the virtualized predictor with that of a predictor with a zero-latency L2 table. The parameters of the virtualized design are summarized in Table 6.3.

Figure 6.15 shows the average MPKI of the virtualized predictor for different $N$. Also shown for comparison are the average MPKI of the baseline predictor and the predictor with a zero-latency L2 table. The two dashed plots show the percentage improvement in MPKI for the two different values of L2 latency over the baseline. The difference between
Next we compare the accuracy of the virtualized design with that of the baseline predictor. We described in Chapter 4 that virtualization can be used either to improve accuracy while keeping dedicated storage constant, or to reduce dedicated storage costs while maintaining the same prediction accuracy. These two goals are evaluated next.

The figure shows the accuracy improvement over the baseline for each $N$. The improvement is larger for smaller $N$, where the extra capacity of the L2 table is comparatively larger than the aggregate dedicated L1 capacity (The aggregate L1 capacity at each $N$ is ~2N KB). The improvement ranges from 30.5% for $N = 1$ to 10.6% for $N = 12$.

It can be observed from the figure that the virtualized design can also be used to reduce the number of predictor tables, while maintaining the same accuracy as a non-virtualized
6.3. Design Space Exploration

design. Comparing the MPKI values for the virtualized and baseline predictors, we see that for $N = 2$ and $N = 3$, one table can be eliminated in the virtualized design, while achieving better accuracy. For larger $N$, at least two tables can be removed while maintaining accuracy. Depending on the number of dedicated predictor tables, 20% to 50% of the tagged tables can be removed, while achieving same or better accuracy than a non-virtualized design.

**Effect of L2 Latency on L1 Miss Rate and Page Swap Frequency**

In this section, we evaluate the effect of latency on the L1 miss rate, as well as on the page swap frequency. The L1 miss rate is calculated as the number of instances where a wrong page is accessed on the L1 table over the total number of predictions made. The page swap frequency is measured as the total number of page swaps into the L1 over the total number of branch predictions made, and determines the traffic to the L2 cache. For a predictor with no latency, the two metrics are equivalent. For larger latencies, we expect the L1 miss rate to be higher than the page swap frequency, as a branch may miss multiple times before the correct page is fetched. Both metrics are properties of the benchmark and are independent of $N$.

Figure 6.16 shows the L1 miss rate for each benchmark, as well as on average, for different values of latency. The figure shows that the L1 miss rate is higher for larger latencies. At latencies of 5 and 20 cycles respectively, the L1 miss rate is approximately double and triple the miss rate for no latency.

For the L1 miss rate, a miss is counted regardless of whether the access is used for the final prediction or not. As a result, although the miss rate is high, the percentage of times a miss affects prediction is smaller. If 20% of the final predictions are provided by ltable, then at a latency of 20 cycles (with 31% L1 miss rate), the percentage of times a miss affects the final prediction is $0.20 \times 0.31 = 0.061$, approximately 6%.

Figure 6.17 shows the page swap frequency for each benchmark, as well as on average, for different values of latency. It can be observed from the figure that the page swap
Figure 6.16: Effect of latency on L1 miss rate

frequency is equal to the L1 miss rate for a zero-latency L2 table. For other latencies, the page swap frequency is smaller than the L1 miss rate, and is approximately equal to the ideal page swap frequency. Although for higher latencies, the predictor incurs multiple misses before a page is fetched (higher L1 miss rate), the number of page swaps remains relatively constant.

The figure shows that the page swap frequency decreases marginally with longer latencies. This effect can be explained with the following example: If the L1 table has a capacity of a single page, and currently holds page A, for a sequence of page accesses \{B, A\}, the sequence of page-ins for the zero-latency predictor is also \{B, A\}. For a predictor with a non-zero latency, the sequence of page-ins is \{B\}, since page A already resides in the L1 at the time of the second access.

**Analysis of Effect of L2 Latency Constraints**

In this section, we discuss different factors that influence the sensitivity of the paged design to the L2 table delay. We examine how latency affects the breakdown of MPKI contri-
6.3. Design Space Exploration

Figure 6.17: Effect of latency on page swap frequency

bution from each of the predictor tables by comparing a zero-latency L2 with a 20-cycle L2 delay. The results are presented for a paged design with a 32 B instruction block size.

Figure 6.18 (a) shows the breakdown of the MPKI contributed from the bimodal table, ltable, and the other tagged tables for different N at two L2 latencies, 0 and 20. The figure shows that for all N, the MPKI contribution of ltable is smaller at a latency of 20, while the MPKI contribution of bimodal is larger. The aggregate MPKI contribution of the other tagged tables is also generally smaller compared with zero-latency L2.

The difference in MPKI at different latencies depends on several factors. One important factor is the original ltable contribution (at latency 0), which affects the sensitivity of the predictor to latency for two reasons: First, the higher the number of predictions made by this table, the higher the probability that predictions will have to be delegated to other tables in case of adverse effects of latency. Second, ltable’s original contribution rate determines the spacing between the time two predictions are required from the table. The higher the contribution rate, the smaller the number of cycles between two such predictions, and the less delay that the predictor can tolerate in fetching pages. Figure 6.18 (b)
Figure 6.18: effect of latency on predictor table’s contribution to MPKI and final prediction
(a) MPKI contribution (b) contribution to final prediction
6.3. Design Space Exploration

shows the contribution of each of the ltable, bimodal table, and other tagged tables to the final prediction at the two latencies.

The figure shows that the smaller the original contribution of ltable to final prediction (at 0 latency), generally the less the difference there is between ltable’s MPKI contribution at the two latencies in Figure 6.18 (a). The number of tagged tables also influences this difference, since it determines how the new misses are offloaded to the other predictor tables. The effect of latency can be explained most clearly at $N = 1$, where any misses in the tagged table are offloaded to the bimodal predictor. The figure shows that the effect of latency is largest at $N = 1$. The original contribution of ltable is also largest at this point. The smallest difference in MPKI contribution of ltable (and the overall MPKI) for the two latencies is at $N = 7$, and Figure 6.18 (b) shows that the contribution of ltable is smallest at this point.

The number of cycles between the time a page is fetched and when the first prediction from that page is needed is another factor that impacts the results. In a page fault scheme, a page is normally fetched at the time when its data is needed. However, since the predictor has multiple tables, the data in a requested page is not necessarily needed at the time the request is made. If ltable is not the component providing the final prediction for the branch that caused the page fetch, then the predictor is not affected by the latency to fetch the page. A simple example of this case is if we consider page fetch and the contribution of ltable to final prediction as periodic functions of time with the same period but a phase difference of 20 cycles. In this case, the introduced latency would have no effect on the predictor MPKI.

Finally, our choice to allow updates and allocates on a wrong page also plays a role in the impact of latency on prediction accuracy. This effect can be explained with the following example: If the L1 table has a capacity of a single page, and the L2 table contains two pages, then an L1 miss rate of 100% does not change the accuracy of the predictor. In this case, each branch always uses the other page to make predictions and allocate entries on the table. Therefore, the accuracy of the predictor remains unaffected, since the only
6.3. Design Space Exploration

In this section, we evaluate the sensitivity of the results to the initial table sizes by virtualizing a predictor with double the initial size as that used up to this point as an example. For the baseline predictor, we fix the number of entries on the tagged components at 4K entries ($TAGE_{N,4K}$). The same parameters as the previous section (Table 6.3) are used for the virtualized design.

Figure 6.19 shows the upper bound on the potential improvement that can be achieved through the paging scheme for this table size, using the idealized predictor of Section 6.3.3. Also shown for comparison, is the improvement that can be achieved by doubling the size of all predictor tables ($TAGE_{N,8K}$). This figure is similar to Figure 6.4 for the previous table size.

Comparing Figures 6.19 and 6.4 for the two table sizes, we see that the improvement
Figure 6.20: Improvement loss due to page size, instruction block size, and latency constraints – 4K-entry dedicated tables

achieved by doubling the table sizes is smaller in this case, although the extra capacity added to the predictor is twice as much for each $N$.

Similarly, the maximum potential improvement for the paging scheme is smaller for this table size. For example, at $N = 6$, the maximum potential improvement is 16%, compared to 28% for the previous table size. At $N = 12$, the maximum potential improvement for this table size is 8%, half of that for the previous table size. Therefore, a larger size for the first level reduces the maximum potential for virtualization even in the absence of constraints.

Figure 6.20 summarizes the effect of page size, instruction block size, and latency constraints on the accuracy of the predictor. The segments show (from top to bottom) the percentage improvement lost due to page size, instruction block size, and latency constraints respectively. The remainder shows the percentage improvement of the final virtualized design over the baseline. The sum of the bars adds up to the maximum potential
Figure 6.21: Accuracy improvement for virtualization of delay-constrained configurations

Comparing the figure with Figure 6.15 for the previous table size, we observe that the improvement achieved through virtualization is much smaller in this case. Although small, virtualization is still beneficial, providing an improvement in the range of 19.2% for \( N = 1 \) to 4.1% for \( N = 12 \).

### 6.4 Virtualized Design Accuracy and Improvement Over Baseline for Delay-Constrained Configurations

In this section, we present MPKI results for the virtualization of the delay-constrained configurations (Table 4.4), and provide a comparison with the non-virtualized predictor. For the delay-constrained configurations, the size of the first-level tagged tables starts at 16 K entries at \( N = 1 \), and is successively cut in half at \( N = 2 \), \( N = 4 \), and \( N = 8 \). The
6.4. Virtualized Design Accuracy and Improvement Over Baseline for Delay-Constrained Configurations

The best accuracy for the non-virtualized design is achieved at \( N = 8 \). For this point, the accuracy can be improved by 8.7\% using the virtualized design. Alternatively, a vir-
6.5 Summary

The virtualized design with 5 tagged tables can be used to achieve the same accuracy as the best baseline predictor. This reduces the dedicated storage of the predictor by ~12 KB, 25% of the original dedicated predictor size.

To show the effect of virtualization on the L2 traffic, Figure 6.22 shows the L2 read and write rates for each $N$, measured as the total number of L2 reads or writes over the total number of conditional branches. The measured rates have a unit of read or write per conditional branch. The L2 read rate is the same as the page swap frequency in previous sections. L2 writes occur only for dirty pages, and as a result, the L2 write rate is smaller. The figure shows that the maximum read and write rates are ~13.4% and ~5.2% both at $N = 12$. For the virtualized design at $N = 8$, the L2 read rate is ~10.8%, while the write rate is ~3.7%.

6.5 Summary

In this section, we presented the paging scheme, where the entries in the predictor table are divided into pages shared by a group of branch instructions in close proximity in the program code space. Exploring the design space of this scheme, we found that virtualizing shorter history length tables provides the best benefit in improving prediction accuracy. We found that smaller page sizes have higher MPKI compared with larger pages. We demonstrated that by sharing a page among branches in a larger instruction block, the traffic to and from the L2 can be decreased almost arbitrarily, but at the cost of an increased MPKI. Virtualizing only a single predictor component allowed more tolerance to the L2 delay, as the table does not contribute to every prediction. Finally, we showed that the virtualized design remains effective even for larger dedicated table sizes, although page size limitations are worse in this case.
Chapter 7

Packaging Scheme

In this section, we present a different virtualization scheme for the TAGE predictor, the packaging scheme. We present preliminary data demonstrating the potential of the scheme by showing the improvement margin available before considering second-level predictor table delay. We leave investigation into the effects of delay and methods for mitigating these effects for future work.

7.1 Design

In the packaging scheme, the second-level predictor table stores packages, similar to the Phantom-BTB design presented in Section 2.2. These packages contain predictor entries that can be used as part of the prediction mechanism. For a packaging scheme, two issues need to be addressed: the grouping of data in the second-level table and the fetch trigger. The grouping refers to the organization of data in packages that are fetched from the second-level table according to a fetch trigger. Phantom-BTB is an example of such a packaging scheme. In Phantom-BTB, packaging is done on BTB table misses. As branches miss in the dedicated BTB, they are packaged into a temporal group. For the fetch trigger, each temporal group is associated with a region of code surrounding the branch previous to the group that also missed in the BTB. Any subsequent misses for a branch within this region triggers the prefetch of the temporal group.

In our packaging scheme, we borrow the idea of using the upper bits of the branch address to associate branches with pages in the paging scheme. Similarly, we use the upper bits of the branch PC to associate a branch with a specific package in the second-level table. Whenever an entry is evicted from the dedicated predictor table, it is grouped into its
corresponding package. In order to determine which package an evicted entry should be placed in, we need to know the address of the branch to which the entry belongs. However, in TAGE and in branch outcome predictors in general, both the index to the predictor table and the entries’ tag fields are a hash of the branch address and other information such as global and path histories. Therefore, the address of the branch can not be inferred from the entry’s tag or its position in the predictor table. As a result, we augment the entries in the predictor table with a field containing the package number of the branch that first allocated the entry. This package number is used to determine the package to which the entry should be placed, in the event of eviction.

Based on the address of the current branch, a corresponding package is fetched from the L2 table, so that its entries can be used as part of the prediction mechanism. In general, the two packages used for storing evicted entries (determined based on the address of the branch that initially allocated the entry) and for contributing to prediction (determined based on the current branch address) are two different packages.

The packaging scheme takes advantage of the spatial and temporal locality present in the branch address stream. The entries in a package are evicted entries belonging to a set of branches that are close together in the instruction address space (have the same upper PC bits). Based on the current portion of the code, the corresponding package is fetched and its entries are used to contribute to the final prediction.

Similar to the paging scheme, packaging is applied to a single tagged component in the TAGE predictor, which we call the ptable. The package number field can be stored in a separate array so as not to increase the access delay of the predictor table. Since this field is only used to group evicted entries into their corresponding packages, it is not on the critical path of making a prediction. Alternatively, it may be possible to merge the package number with the entry’s tag field by limiting the XOR portion of the tag, similar to way we limited the XOR portion of the table index in the paging scheme. It is necessary to evaluate the sensitivity of the predictor to the tag hash function in order to determine the feasibility of this option, which we leave for future work.
7.2. Architecture

An alternative to packaging evicted entries is to place *newly allocated* entries into the packages, whenever the predictor table does not have enough space. The advantage of this scheme is that it would not be necessary to keep track of the package number of the branch that initially allocated the entry. We based our decision to package entries upon eviction on the assumption that newly allocated entries have a higher probability of being useful in the near future, especially as evicted entries are generally chosen because they have a *useful bit* of 0, as explained in Section 2.1.1. With this scheme, we allow the dedicated predictor table to be kept up-to-date and adapt to the different phases of the program. Instead, the second-level storage is used to keep track of evicted entries that *may* be useful in the future. With a “package on allocate” scheme, the accuracy of the predictor would be more sensitive to the delay of the second-level table, since it would rely on the second-level table to keep track of the most recent updates to the predictor table. In the eviction scheme, in the worst case that no packages can be fetched in time to contribute to the final prediction, the accuracy of the predictor will be no worse than without packages. Therefore, although it may be feasible to explore packaging entries upon allocation, the packaging of evicted entries represented a better initial design point to us.

7.2 Architecture

The architecture of the packaging design is depicted in Figure 7.1. In the rest of this section, we use the terms ptable and L1 table interchangeably to refer to the dedicated predictor table to which the packaging scheme is applied. We describe the components of the design next.

**L2 Table** Unlike the paging mechanism, the organization of entries in the second-level table in the packaging scheme differs from that of the first-level table. The L2 table is organized into packages that are equal in size to the L2 cache line, and each package contains as many entries as can fit within this amount of storage. Each package entry contains the L1 table entry fields and the L1 table index of the entry. In TAGE, since the index and tag
hash functions are computed differently, two entries in different positions on the predictor table may have the same tag. Therefore, the combination of each entry’s index and tag is used to uniquely identify each dynamic instance of a branch. The entries in the packages do not include the package number field from the first-level table, since the package number is implicit in the package in which an entry resides.

**Fetch Engine** The fetch engine is used to fetch packages from the second-level table for the purpose of being used in the prediction mechanism. These packages are placed in the prediction buffer. For each branch, the dedicated predictor table and the prediction buffer are accessed in parallel to provide a prediction.

To determine a hit in the prediction buffer we require that both the index and the tag of the entry match the computed index and tag respectively. In case of a miss in the dedicated predictor table and a hit in the prediction buffer, the entry in the buffer is used to provide
7.2. Architecture

Figure 7.2: Package number determined by $n$ bits of branch PC

the prediction for the ptable component. The final prediction is selected among all the TAGE tables as before. If the final prediction was provided by the prediction buffer, then the entry is updated in-place in the prediction buffer.

Package Generation Unit The package generation unit is in charge of placing evicted entries in their respective packages. The packaging buffer is used to store the package in which an evicted entry is placed for the current branch.

As described before, the packages in the packaging and prediction buffers are generally different. The package in the prediction buffer is determined by the package number of the current branch, while an evicted entry is placed into a package belonging to the branch that initially allocated it. We will see in Section 7.3.3 that the swap rate for packages in the packaging buffer is very small, so that the major portion of the traffic to and from the L2 comes from the prediction buffer.

Package Number Figure 7.2 shows how the branch address is used to determine the package number for the branch. Similar to the paging scheme, the instruction address space is conceptually divided into instruction blocks, where each instruction block is assigned its own package on the second-level predictor table. Branches in one instruction block share a package to store their evicted entries. In the figure, $b$ is $\log_2$ of the instruction block size and $n$ is $\log_2$ of the number of packages that reside in the L2. The package number is determined by bits $[b + n - 1 : b]$ of the branch address.

Since each entry in the L1 is augmented with the package number, this scheme has an overhead of $M \times \log_2(T)$ bits of extra dedicated storage, where $M$ is the number of L1 entries, and $T$ is the number of packages in the L2.
7.3 Experimental Results

In this section, we experimentally evaluate the potential benefit of the packaging scheme, ignoring the effect of the L2 table latency. The experiments are organized to parallel those for the paging scheme (Chapter 6).

Similar to the experiments presented in Section 6.3 for the paging scheme, we use a fixed size of 2K entries for all tagged tables. The size of the bimodal table is 16K entries.

In all the experiments presented in this chapter, we use an L2 table with 1K 64 B packages, for a total storage of 64 KB. The package size is chosen such that it fits in a single L2 cache line. The L2 size is chosen equal to the L2 size for the paging scheme, to allow for a fair comparison of the two schemes. The packaging scheme has the advantage that the L2 represents storage additional to the L1 table. As a result, any L2 size can be used to provide potential gains. Unlike this scheme, the L2 table has to be larger than the L1 in the paging scheme, since the L2 is a superset of the L1.

A summary of the rest of the section is as follows:

• In Section 7.3.1, we determine the table that provides the best benefit from the extra capacity in the form of packages.

• In Section 7.3.2, we show the maximum potential improvement that can be achieved through virtualization with the packaging scheme in the absence of L2 delay constraints.

• In Section 7.3.4, we increase the instruction block size in order to reduce the package swap rate (traffic to and from the L2 table).

• Finally, in Section 7.3.6, we discuss a prefetching mechanism that can be used to prefetch packages in order to potentially reduce the effects of L2 delay.
7.3. Experimental Results

In this section, we determine which table benefits most from the extra capacity provided in the form of packages. We use a predictor model that allows access to the packages without any delay. In the next section, we use these results to show the maximum potential gain of the packaging scheme.

In this model of the predictor, one of the tables is augmented with an L2 table organized into packages. The packages are placed in the prediction and packaging buffers without any delay. We refer to this model as the zero-delay packaged design, and use this model in the rest of this chapter. The accuracy of this predictor represents an upper bound on the improvement that can be achieved with the packaging scheme.

In this section, we use an instruction block size of 32 B, same as the initial instruction block size used in the paging scheme, to allow for a direct comparison of the two designs.

Figure 7.3 shows the average MPKI of the zero-delay packaged design for different values of $N$. The x-axis shows which table was augmented with an L2 table, while each line
Table 7.1: Default Ptable Configuration

<table>
<thead>
<tr>
<th>Number of Tagged Tables (N)</th>
<th>Ptable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 5</td>
<td>1</td>
</tr>
<tr>
<td>6 to 12</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 7.2: Packaging Scheme Configuration

<table>
<thead>
<tr>
<th>Package Size</th>
<th>22 Entries (~64 B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Block Size</td>
<td>32 Bytes</td>
</tr>
<tr>
<td>L2 Size</td>
<td>1K Packages</td>
</tr>
</tbody>
</table>

represents a different $N$. The average MPKI of the baseline predictor (with no ptable) is also shown at $N = 0$ for comparison. This figure is similar to Figure 6.3 used for choosing ltable in the paging scheme. We expect the two figures to show similar results, since both experiments effectively increase the capacity of a single predictor table, only in two different ways.

It can be observed from the figure that in all cases, the first few tables benefit the most from the additional capacity. The best MPKI is roughly achieved with $ptable = 1$ for $N \leq 5$, and $ptable = 2$ for $N > 5$. These results, which we refer to as the default ptable configuration, are summarized in Table 7.1. We use the default ptable configuration in the rest of this chapter.

For the default ptable configuration, each package entry is 23 bits (12 bits for the predictor table entry, plus 11 bits to store the entry’s index). As a result, each package (64 B) contains 22 such entries.

7.3.2 Potential Improvement with the Packaging Scheme

In this section, we use the zero-delay packaged predictor with the default ptable configuration (Table 7.1) to show the potential improvement that can be achieved through the packaging scheme. Table 7.2 summarizes the parameters of the packaged design. We compare the results in this section with the results for the zero-delay paging scheme with the same page size (64 B) and instruction block size (32 B) (Section 6.3.4). Figure 7.4 shows the average MPKI of the zero-delay packaged predictor. Also shown
7.3. Experimental Results

![Graph](image)

Figure 7.4: Maximum potential improvement with packaging scheme using 64 B packages and instruction block size for comparison is the average MPKI of the baseline predictor ($TAGE_{N,2K}$). The dashed plot shows the percentage improvement in the predictor accuracy over the baseline. The figure shows that the potential improvement for this scheme is in the range of ~15% to ~48% for different $N$. The potential is smaller for larger $N$. The discontinuity at $N = 6$ marks the point where ptable changes from the first tagged table to the second table.

Next, we compare the potential improvement of the packaging scheme (Figure 7.4) with the potential improvement of the paging scheme in Figure 6.6. The improvement range for this scheme is between ~15% to ~48%, compared with the range of ~15% to ~41% for the paging scheme. The two schemes achieve similar results for $N = 12$. However, the packaging scheme is always better than the paging scheme for other values of $N$. The difference between the potential improvement of the two schemes is larger for smaller $N$. For example at $N = 8$, this scheme results in a potential improvement of 20.3%, while the paging scheme results in an 18.3% potential improvement.
7.3. Experimental Results

In the next section, we measure the frequency of package swaps and show that the swap frequency is larger than the page swap frequency of the paging scheme.

7.3.3 Package Swap Frequency

In this section, we measure the package swap frequency for both the prediction and packaging buffers. The package swap frequency determines the impact of virtualization on the traffic to the L2 cache. For each structure, we calculate the package swap frequency as the total number of package fetches over the total number of branch predictions made (the package swap frequency has a unit of swaps per conditional branch). We assume that the two structures always fetch different packages. As a result, the total package swap rate is the sum of the two buffers’ swap rate.

In our experiments, we use a victim buffer to store a single evicted package from the prediction buffer. This method is employed to reduce the prediction buffer package swap frequency. Package swaps between the victim and prediction buffers are not counted as part of the structure’s package swap frequency.
Figure 7.5 shows the package swap frequency for the packaging and prediction buffers, as a function of $N$. The prediction buffer package swap rate is independent of $N$, since it is determined by the stream of branch addresses (and, consequently, packages) encountered in the program. Similar to page swap frequency, the prediction buffer package swap frequency is a property of the benchmark itself and is a function of the entropy present in the branch address bits. In contrast, the packaging buffer swap rate differs for each $N$, as it is determined by the pattern of evictions from the L1 table.

The figure shows that the prediction buffer package swap frequency is 56.8%. For an equivalent configuration, the paging scheme swap frequency is 32.8% (Figure 6.7). The page swap frequency is lower, since the paging scheme allows multiple pages (64 pages in our case) to simultaneously reside in the L1 table.

Interestingly, the package swap frequency for the packaging buffer is rather small. For this buffer, the swap frequency decreases with larger $N$, with a maximum of 0.19% at $N = 1$. Since this number is much smaller relative to the prediction buffer swap rate (56.8%), we ignore its effects on the L2 traffic. From this point on, we use the prediction buffer swap rate to measure the impact of virtualization on the L2 traffic, and use the term package swap frequency to refer to the prediction buffer package swap frequency.

We address the issue of lowering the package swap frequency next.

### 7.3.4 Instruction Block Size

In the previous section, we showed that for a 32 B instruction block size, a package would have to be swapped into the prediction buffer more often than every other conditional branch. In this section, we vary the instruction block size in order to lower the package swap frequency. We evaluate the effect of varying the instruction block size on package swap frequency as well as on the predictor accuracy.

The instruction block size is varied between 32 B and 16 KB, in effect having branches in that region of code share the same package for their evicted entries. The experiments in this section are similar to those presented in Section 6.3.5 to lower the page swap frequency.
7.3. Experimental Results

Figure 7.6: Effect of instruction block size on prediction buffer package swap frequency for the paging scheme.

**Effect of Instruction Block Size on Package Swap Frequency**

Figure 7.6 shows the package swap frequency as a function of the instruction block size, for each benchmark and on average. The figure shows that as the instruction block size is increased (and more branches share a package), the package swap frequency drops, but at a slower rate. However, even with a 16 KB instruction block size, the package swap frequency does not drop below 11%. The paging scheme achieves the same swap frequency with an instruction block size of 256 B (Figure 6.10).

Next, we examine the effect of the instruction block size on the utilization of the L2 table packages. As the instruction block size is increased and higher portions of the PC are used for the package number, some package numbers may not be encountered in the program run. Figure 7.7 shows the percentage of such packages as a function of the instruction block size. For the two schemes respectively, the percentage of package and page numbers not encountered in the program run are the same.
Figure 7.7: Effect of instruction block size on the fraction of packages never accessed

The figure shows that as instruction block size is increased, the percentage of packages that are never accessed grows larger. For an instruction block size of 16 KB, more than half the packages are never accessed in the entire program run.

In conclusion, a larger instruction block size can be used to reduce package swap frequency, but not beyond a certain value. At the same time, larger instruction block sizes lead to an underutilized L2 table. In Section 6.3.5, we described a method for decreasing the swap rate with less impact on L2 utilization by incorporating branch history information in the page fetch mechanism. The same method can be used for fetching packages.

Effect of Instruction Block Size on MPKI

Figure 7.8 shows the effect of varying the instruction block size on the average MPKI, for different N. The figure shows that a larger instruction block size leads to a worse MPKI, and the effect is more pronounced for smaller N. Up to an instruction block size of 256 B, the effect on MPKI is minor. One reason for this may be that up to this point, all packages are still in use (Figure 7.7).
7.3. Experimental Results

Based on Figures 7.6 and 7.8, it can be seen that lowering the package swap rate comes at the cost of an increased MPKI. The first few values of the instruction block size represent the best tradeoff between the two.

7.3.5 Improvement for Packaging Scheme Before Considering L2 Delay

In this section, we summarize the MPKI results for the packaging scheme before considering the L2 delay effects. Based on the previous section, we choose an instruction block size of 1 KB. For this size, the package swap frequency is cut down to a third of its original value with a 32 B instruction block size (from ~56.8% to ~17.8%). For this instruction block size, an average of ~6.5% of the 1 K packages are never utilized. Table 7.3 summarizes the parameters for this design.

Table 7.3: Final Packaging Scheme Configuration

<table>
<thead>
<tr>
<th>Package Size</th>
<th>22 Entries (~64 B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Block Size</td>
<td>1 KB</td>
</tr>
<tr>
<td>L2 Size</td>
<td>1K Packages</td>
</tr>
</tbody>
</table>

Figure 7.8: Effect of instruction block size on MPKI
Figure 7.9: Potential improvement for packaging scheme using an instruction block size of 1 KB

Figure 7.9 shows the average MPKI of this predictor for different $N$. Also shown for comparison are the average MPKI of the baseline predictor ($TAGE_{N,2K}$) and predictor with 32 B instruction block size (Section 7.3.2). The two dashed plots show the percentage improvement in the accuracy of the predictor with the two different instruction block sizes over baseline. The difference between the two lines represents the improvement loss due to using an instruction block size 32 times larger than the original. This instruction block size removes ~2%-3% of the improvement that can be achieved using a 32 B instruction block size.

Comparing the packaged design with the baseline predictor, the figure shows the accuracy improvement over baseline for each $N$. The improvement is larger for smaller $N$, where the L2 capacity is comparatively larger than the aggregate dedicated L1 capacity (The aggregate L1 capacity at each $N$ is ~2$N$ KB). The improvement ranges from 45.7% for $N = 1$ to 13.9% for $N = 12$. 
7.3. Experimental Results

Figure 7.10: Maximum potential improvement before considering L2 delay for paging and packaging schemes

Figure 7.10 compares the percentage improvement in accuracy for the packaging and paging schemes for different $N$. The percentage improvement for paging scheme has been reproduced from Figure 6.13. It can be seen in the figure that packaging always achieves better accuracy improvement than paging. The difference between the two schemes is generally larger for smaller $N$. The two schemes perform almost similarly for $N = 12$.

In conclusion, with no L2 delay, the packaging scheme can achieve better accuracy than the paging scheme, especially for smaller $N$. We expect the difference between the two schemes to be greater for larger L1 sizes, since losses in the paging scheme due to page size constraints are more significant for larger L1 sizes. Compared with the paging scheme, however, this scheme adds a larger dedicated storage overhead. In addition, it has a larger swap frequency, causing larger traffic to and from the L2. Due to the higher swap frequency, it may also be more sensitive to L2 delay. The impact of L2 delay on the scheme will also depend on the contribution of ptable to final prediction, and specifically the
7.3. Experimental Results

Figure 7.11: Accuracy of a 1K-entry PTB on the SPEC CPU 2000 benchmark set

contribution of the prediction buffer. The high package swap frequency may be relieved by allowing more packages in the prediction buffer. Prefetching of packages may also be helpful to lower the impact of L2 delay. In the next section, we introduce a simple structure, the Package Target Buffer, which can be used to prefetch packages, and which may help reduce the sensitivity of the scheme to the L2 delay.

7.3.6 Prefetching

In this section, we discuss a prefetching mechanism that may be useful in decreasing the sensitivity of the scheme to the L2 delay. We propose a new structure called the Package Target Buffer (PTB), which keeps track of the targets of packages. It is indexed with the current package number in order to predict the next package number. The PTB is similar to the BTB, which is used to predict targets of branches.

For our virtualized design (with a total of 1K packages), we evaluate a 1K-entry PTB. Each PTB entry is 10 bits ($\log_2(1K)$) for a total storage of 1.25 KB. Figure 7.11 shows the accuracy of this PTB on the SPEC CPU 2000 benchmarks. The figure shows that the PTB
has an average accuracy of 70%, achieving almost perfect accuracy for three of the benchmarks. This motivates the use of the PTB for prefetching of packages.

Since page and package numbers are essentially the same (upper bits of the branch PC), this prefetching mechanism can also be applied to the paging scheme.

7.4 Summary

In this section we presented the packaging scheme, where evicted entries from the first-level predictor table are grouped into packages that are placed in the second-level table. Unlike the paging scheme, the packaging scheme does not impose limitations on the number of L1 predictor table entries that can be accessed by each branch. Instead, it allows branches to access any entry in the L1 table, while using the second level as additional storage to retain evicted entries. Before considering the effects of L2 delay, we showed that the packaging scheme achieves better accuracy than the paging scheme. However, since only a single package is allowed to be active in the L1 at one time, the packaging scheme has higher package swap rate. In addition, for the same L2 size, the packaging scheme requires a larger dedicated storage overhead than the paging scheme, since it extends each entry with the package number of the branch that allocated the entry.
Chapter 8

Conclusion

In this section we summarize the preceding chapters, comment on limitations, and discuss future work.

8.1 Summary

Applications with large instruction footprints, such as commercial workloads, expose the tradeoff among accuracy, latency and hardware cost in branch predictor design. To address this tradeoff, this work introduces novel virtualization techniques for branch outcome predictors. Virtualization is applied to a state-of-the-art multi-table branch predictor, the TAgged GEometric History Length (TAGE) predictor. One of the dedicated predictor tables is augmented with a large virtual second-level table allocated in the second-level caches, increasing its perceived capacity. Chapter 5 presented an overview of the two virtualization methods proposed in this work: the paging scheme (Chapter 6) and the packaging scheme (Chapter 7).

8.1.1 Paging Scheme

In the paging scheme, the entries in the predictor table are divided into pages shared by a group of branch instructions in close proximity in the program code space. Based on the current portion of the code, the corresponding page is fetched from the second level and is placed in the first-level table. The first-level table caches the most recently used subset of the second-level table pages.

Exploring the design space of this scheme, we found that virtualizing shorter history length tables provides the best benefit in improving prediction accuracy. We evaluated the
8.1. Summary

effect of restricting branches to a single predictor table page for various page sizes. We
found that smaller page sizes have higher MPKI compared with larger pages. We showed
that the higher MPKI of smaller pages is not due to an increase in aliasing, but because of
a decrease in the frequency of tag matches. In addition, the effect of page size constraints
are smaller on a predictor with more tagged tables. We demonstrated that by sharing a
page among branches in a larger instruction block, the traffic to and from the L2 can be
decreased almost arbitrarily, but at the cost of an increased MPKI. Finally, we found that
virtualizing only a single predictor component allowed more tolerance to the L2 delay, as
the table does not contribute to every prediction.

Experimental results showed that the virtualized design remains effective even for
larger dedicated table sizes, although page size limitations are worse for larger tables.
We found that the relative size of the L2 to L1 has to be large in order to compensate for
losses due to page size constraints.

In our final virtualized design, branches in a 256 B block of code shared a single page
of 32 entries, which fits in a single L2 cache line. We used a second-level virtual table
containing 1K 64 B cache lines, for a total storage of 64 KB. The virtualized design has an
overhead of 704 bits of extra dedicated storage.

For a predictor with moderate table sizes (2K entries on the tagged components and
a bimodal table of 16K entries), virtualization improves accuracy in the range of 10.6%
to 30.5%, depending on the number of dedicated tables. Alternatively, the virtualized
design can be used to reduce the dedicated budget and complexity of the predictor without
affecting accuracy. In this case, depending on the number of dedicated predictor tables,
20% to 50% of the tagged tables can be removed, while achieving same or better accuracy
than a non-virtualized design.

For a predictor whose size is determined not by area constraints but by the delay to
access its contents, the best accuracy of the baseline predictor is achieved with 8 tagged
tables and can be improved by 8.7% using virtualization. Alternatively, the same accuracy
as the baseline predictor can be achieved using a virtualized design with 5 tagged tables,
8.1. Summary

removing approximately 25% of the dedicated storage.

8.1.2 Packaging Scheme

In the packaging scheme, evicted entries from the first-level predictor table are grouped into packages that are placed in the second-level table. Similar to the paging method, a group of branch instructions spaced closely in the program share a package. Based on the current portion of the code, the correct package is placed in a buffer co-located with the first-level predictor table, and its data is used as part of the prediction mechanism.

For the packaging scheme, we found that shorter history length tables benefit the most from the extra capacity. We showed that increasing the instruction block size can be used to reduce traffic to and from the L2 table, but only to a certain point.

In our final design, branches in a 1 KB block of code shared a single package of 22 entries, which fits in a single L2 cache line. The design adds an overhead of 20 K bits of extra dedicated storage. We used a second-level virtual table containing 1K 64 B cache lines, for a total storage of 64 KB. Before considering the effects of the L2 table delay, the packaging scheme improves accuracy in the range of 13.9% to 45.7%, depending on the number of dedicated tables.

8.1.3 Comparison of the Two Schemes

The paging and packaging schemes can be viewed as increasing the perceived length (size) and width (associativity) of one of the predictor tables, respectively. In both schemes, the upper bits of the branch address are used to determine pages or packages shared by a group of branch instructions in close proximity in the program code space.

The paging scheme imposes limitations on the number of L1 predictor table entries that can be accessed by each branch to within a single page. The packaging scheme removes this limitation by allowing branches to access any entry in the L1 table, while using the second level as additional storage to retain evicted entries. Before considering the effects of L2 delay, the packaging scheme can achieve better accuracy than the paging
scheme, especially for a predictor with fewer dedicated tables. However, since only a single package is allowed to be active in the L1 at one time, the packaging scheme has higher package swap rate. Not only does this result in more traffic to the L2, but it may also increase the sensitivity of the scheme to the L2 delay. Evaluation of the effect of latency on the packaging scheme was left for future work. In addition, for the same L2 size, the packaging scheme requires a larger dedicated storage overhead than the paging scheme, since it extends each entry with the package number of the branch that allocated the entry.

8.2 Limitations and Future Work

In this work, we have used functional simulation to evaluate the accuracy of the proposed branch predictor designs. This is typical of branch prediction studies. For the paging scheme, we used instruction count to approximate processor cycles and to emulate delay of the predictor L2 table. Cycle-accurate simulation would allow us to determine the impact of our design on the L2 cache traffic, as well as the effect of the improved accuracy on the overall processor performance. Furthermore, it would allow us to determine predictor L2 table sizes which benefit prediction accuracy without contending for the L2 cache data. For the packaging scheme, evaluation of the effect of predictor L2 table delay is left for future work.

To enhance the accuracy of the proposed designs, several optimizations can be explored. Extending the schemes to multiple predictor tables, incorporating history bits into page/package numbers (Section 6.3.5), and prefetching of pages/packages (Section 7.3.6) are examples of such enhancements.

Our schemes can also be used to implement traditional hierarchical branch predictors. The design of these predictors can be explored without the L2 cache line size and delay constraints. For the paging scheme, we have tried to show the effects of all parameters without the limitations of the L2 cache constraints, which can be used as a basis for evaluating these predictors.
Appendix A

Branch Prediction Schemes

A.1 Bimodal

The behavior of typical branches is far from random. Most branches are either usually taken or usually not taken. Bimodal branch prediction takes advantage of this bimodal distribution of branch behaviour and attempts to distinguish usually taken from usually not-taken branches [3]. Figure A.1 shows a typical organization of the bimodal predictor.

In this scheme, a table of counters is indexed by the low order bits of the branch address. For each taken branch, the appropriate counter is incremented. Likewise for each not-taken branch, the appropriate counter is decremented. Each counter is two bits long and is saturating. In other words, the counter is not decremented past zero, nor is it incremented beyond three. The most significant bit determines the final prediction. Repeatedly taken branches will be predicted to be taken, and repeatedly not-taken branches will be predicted to be not-taken. By using a 2-bit counter, the predictor can tolerate a branch going an unusual direction one time and keep predicting the usual branch direction. This is especially useful for loop branches which are taken multiple times and not-taken only once when exiting the loop. With a 1-bit counter, each loop branch would incur two mispredictions: on the loop exit and the first loop iteration the next time, while with a 2-bit counter only the loop exit is mispredicted.

A.2 Two-Level Adaptive Predictors

Two-Level Adaptive Branch Prediction [14] uses two levels of branch history information to make predictions. The first level is the history of the outcome of recently encountered
A.2. Two-Level Adaptive Predictors

branches. The second level is the branch behavior for the occurrences of the specific pattern of branch outcomes. The two levels of information are maintained in two major data structures: the branch history register (HR) and the pattern history table (PHT). The history register is a shift register which shifts in bits representing the outcome of the most recent branches. The branch predictor may use an array of history registers each tracking the local history of a specific branch. This array of history registers is known as the branch history register table (HRT). The history register is used as an index into the PHT to select a 2-bit saturating counter that will provide the final prediction.

Depending on the number of history registers, three main classes of two-level adaptive predictors are developed as follows: First, the Per-Address Schemes (PA), in which an array of history registers each keeping track of the outcome of a single branch (local history) is indexed by the branch address. Second, the Per-Set Schemes (SA), in which an array of history registers each keeping track of the outcome of a set of branches is indexed by a portion of the branch address. The number of branch history registers in this scheme is smaller than the PA schemes. And Third, the Global Schemes (GA), in which a single global history register is used to keep track of the outcome of all branches encountered.

Each of the three classes can be further divided into sub-schemes according to the number of pattern history tables used. For example, for the PA scheme three sub-schemes exist: In the PAg scheme, the HR determined by the address of the branch is used to access
A.3. Gshare

In the GAg scheme, the index to the PHT is based solely the outcome of most recently encountered branches. Instead, in the Gshare predictor the branch history is XORed with
Figure A.3: Gshare Predictor [3]

A portion of the branch address to create a hash function to access the PHT [3]. The XOR operation allows accesses to the PHT to be evenly distributed, which would otherwise be unequally distributed due to the non-uniform nature of branch histories. In this way, the Gshare predictor increases accuracy by reducing the probability that two different branches will interfere with each other in the PHT. Figure A.3 shows the Gshare predictor.

A.4 2BC-Gshare

The advantages of different branch prediction schemes can be combined in a hybrid branch predictor, resulting in better prediction accuracy. One such combination of branch predictors is a bimodal predictor and a gshare predictor, known as 2bc-gshare. In this combination, global information can be used if it is worthwhile, otherwise the usual branch direction as predicted by the bimodal scheme can be used. This predictor contains the two bimodal and gshare predictor tables, plus an additional table, known as the meta predictor, which serves to select the best predictor to use. The meta predictor also uses simple 2-bit up/down saturating counters to keep track of which predictor is more accurate for the branches that share that counter. The final prediction is chosen between the bimodal and gshare predictions based on the decision of the meta predictor. Figure A.4 shows the high-level structure of the 2bc-gshare predictor.
A.5 The Skewed Predictor and 2BC-Gskew

The basic principle of the skewed branch predictor [4] is to use three branch predictor tables each indexed by a different and independent hash function of the same vector of information, the branch address and global history, in order to reduce the impact of destructive aliasing. A prediction is read from each of the predictor tables and a majority vote is used to select the final prediction. The rationale for using different hashing functions for each table is that two branches that are aliased with each other on one table are unlikely to be aliased on the other tables. This means that a destructive aliasing of the two branches may occur in one table, but the overall prediction is still likely to be correct.

The 2bc-gskew predictor is a hybrid predictor that combines the skewed predictor with a bimodal predictor, as shown in Figure A.5. The predictor consists of four tables of 2-bit saturating counters. The first table is the bimodal predictor, which is also used as the first table of the skewed predictor. Two more tables are used in the skewed predictor. The fourth table is the meta predictor, which chooses whether the final prediction is provided by the bimodal predictor or the skewed predictor. The bimodal predictor’s prediction is one of the votes in the skewed predictor even if the final prediction is not provided by the bimodal table.

The predictor uses a partial update method to update the table entries. On a wrong
prediction, all predictor tables are updated. On a correct prediction, however, only the
tables that participated in the correct prediction are updated. The reasoning behind the
partial update policy is that a table giving an incorrect prediction probably holds informa-
tion which belongs to a different branch. As a result, this table is not updated. The meta
predictor is updated when the two predictors disagree.

A.6 YAGS

The YAGS predictor is shown in Figure A.6. The YAGS predictor uses filtering and de-
interference techniques in order to reduce the impact of destructive aliasing on the pre-
dictor [7]. A bimodal predictor is used to store the bias of a branch, the direction usually
taken by the branch (filtering mechanism). Instances of the branch that do no comply with
this bias are recorded in two tagged tables called taken and not-taken direction caches.
Figure A.6: YAGS Predictor [7]

The use of small tags (6-8 bits) allows multiple instances of branches to be differentiated from one another (de-interference). The tags store the least significant bits of the branch address.

To make a prediction, the bimodal predictor is accessed to provide the default behaviour of the branch. If it indicates taken, then the not-taken cache is accessed to check if this is a special case where the prediction does not agree with the bias. If there is miss in the direction cache, the default prediction coming from the bimodal table is used. If there is a hit, then the not taken cache supplies the final prediction. A similar set of actions is taken if the bimodal predictor indicates not-taken, but in this case the check is done in the taken cache.

If the outcome of a branch differs from the prediction of the bimodal predictor, the corresponding direction cache is updated. A direction cache is also updated if a prediction from it is used. The bimodal predictor is updated if it correctly predicts a branch or the direction caches were incorrect.
A.7 Perceptron

Figure A.7 shows the perceptron predictor [21]. A perceptron is a vector of \( h + 1 \) small integer weights, where \( h \) is the history length of the predictor. The global history is kept in a shift register and the perceptrons are kept in a table analogous to the shift register and table of counters in traditional global two-level predictors.

To predict a branch, the lower bits of the branch address are used to select a perceptron from the table. The output of the perceptron is computed as the dot product of the perceptron and the history shift register, with the 0 (not-taken) values in the shift register interpreted as -1. Added to the dot product is an extra bias weight in the perceptron, which takes into account the bias of the branch. If the result of the dot product is greater than or equal to 0, then the branch is predicted taken; otherwise, it is predicted not-taken. Negative weight values denote inverse correlation between the outcome of the branch and the corresponding history bit. The magnitude of the weight indicates the strength of the positive or negative correlation.

At update time, the perceptron is trained on a misprediction or when the magnitude of the perceptron output is below a specified threshold value. Upon training, both the bias weight and the \( h \) correlating weights are updated. The bias weight is incremented or decremented if the branch is taken or not taken, respectively. Each correlating weight in the perceptron is incremented if the predicted branch has the same outcome as the corre-
A.8 O-GEHL

sponding bit in the history register and decremented otherwise with saturating arithmetic. If there is no correlation between the predicted branch and a branch in the history register, the corresponding weight will tend toward 0. If there is high positive or negative correlation, the weight will have a large magnitude.

A.8 O-GEHL

The main component of the O-GEHL predictor [25] is the GEHL predictor which is composed of several predictor tables indexed through independent functions of branch history and branch address as illustrated in Figure A.8. The set of global history lengths used for the predictor tables forms a geometric series i.e. $L(i) = a^{i-1} \times L(1)$. This allows the predictor to capture correlation on very old branches by using long history lengths, while still devoting most of the predictor resources to short history lengths. Similar to the perceptron predictor, the prediction is computed through the addition of the predictions read from each of the predictor tables. The O-GEHL predictor further optimizes the predictor through the addition of dynamic history length fitting [38] and dynamic threshold fitting. In this section, we focus on the structure of the GEHL predictor.

The predictor features $M$ distinct predictor tables, indexed with a hash of global and
A.8. O-GEHL

path histories and the branch address. The predictor tables store predictions as signed saturated counters. To compute a prediction, a single counter is read from each predictor table. The final prediction is computed as the sign of the sum $S$ of the $M$ counters. The branch is predicted taken when $S \geq 0$ and not-taken otherwise.

Distinct history lengths are used for computing the index of the distinct tables. The first table is indexed using the branch address only (similar to the 2bc-gskew predictor). The history lengths used for computing the index functions to the other predictor tables form a geometric series.

The predictor update policy is derived from the perceptron predictor update policy. The predictor entries are only updated on mispredictions or when the absolute value of the computed sum $S$ is smaller than a threshold, using saturated arithmetic.
References


Appendix A. References


Appendix A. References


Appendix A. References


Appendix A. References


