Predictor Virtualization: Teaching Old Caches New Tricks

by

Ioana Burcea

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2012 by Ioana Burcea
Abstract

Predictor Virtualization:
Teaching Old Caches New Tricks

Ioana Burcea
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2012

To improve application performance, current processors rely on prediction-based hardware optimizations, such as data prefetching and branch prediction. These hardware optimizations store application metadata in on-chip predictor tables and use the metadata to anticipate and optimize for future application behavior. As application footprints grow, the predictor tables need to scale for predictors to remain effective.

One important challenge in processor design is to decide which hardware optimizations to implement and how much resources to dedicate to a specific optimization. Traditionally, processor architects employ a one-size-fits-all approach when designing predictor-based hardware optimizations: for each optimization, a fixed portion of the on-chip resources is allocated to the predictor storage. This approach often leads to sub-optimal designs where: 1) resources are wasted for applications that do not benefit from a particular predictor or require only small predictor tables, or 2) predictors under-perform for applications that need larger predictor tables that can not be built due to area-latency-power constraints.

This thesis introduces Predictor Virtualization (PV), a framework that uses the traditional processor memory hierarchy to store application metadata used in speculative hardware optimizations. This allows to emulate large, more accurate predictor tables, which, in return, leads to higher application performance. PV exploits the current trend of unprecedentedly large on-chip secondary caches and allocates on demand a small portion of the cache capacity to store
application metadata used in hardware optimizations, adjusting to the application’s need for predictor resources. As a consequence, PV is a *pay-as-you-go* technique that emulates large predictor tables without increasing the dedicated storage overhead.

To demonstrate the benefits of virtualizing hardware predictors, we present virtualized designs for three different hardware optimizations: a state-of-the-art data prefetcher, conventional branch target buffers and an object-pointer prefetcher. While each of these hardware predictors exhibit different characteristics that lead to different virtualized designs, virtualization improves the cost-performance trade-off for all these optimizations.

PV increases the utility of traditional processor caches: in addition to being accelerators for slow off-chip memories, on-chip caches are leveraged for increasing the effectiveness of predictor-based hardware optimizations.
Acknowledgements

My PhD experience entailed a lot more than the technical contributions documented in this thesis. In this acknowledgement, I would like to mention the people that enriched my graduate school experience and contributed - directly or indirectly - to the success of my PhD. There have been so many people I interacted with during my graduate school that I am bound to forget a name or two. I want to start this acknowledgement by apologizing to those that deserve to be in these pages and have been omitted.

It probably goes without saying that this thesis is the result of a fruitful collaboration with my research supervisor and advisor, Prof. Andreas Moshovos. I am thankful that he took me under his supervision, although my background in computer architecture was very limited at the time. Under his guidance, I was able to quickly catch up and find a thesis topic that I enjoyed working on. His initial support extended throughout many years to come and it manifested in a variety of instances. I am particularly grateful for the generous freedom of working on topics that I found interesting and fun. I could fill in pages reminiscing about all the occasions in which Andreas showed his support and guidance. Instead, I will say that, if I were given the chance to choose a PhD supervisor again, I would make the same choice.

My thesis committee members, Prof. Tarek Abdelrahman, Prof. Greg Steffan, Prof. Natalie Enright Jerger and Prof. Dan Sorin, helped improve not only the work presented in this thesis, but also my skills as a researcher.

My research was kick-started by using the Flexus simulation framework, which was developed at the time by a group of CMU researchers lead by Prof. Babak Falsafi. I would like to thank all members of the CMU research group for their generosity in sharing the simulation infrastructure and for their help in using it.

Special thanks go to my close collaborators, Stephen Somogyi and Prof. Babak Falsafi, that shared their implementation of the Spatial Memory Streaming prefetcher and helped with validating my initial research results. Among my collaborators, I would like to thank Livio Soares for modifying the OpenJDK runtime system to provide the information I needed to
conduct the last piece of my PhD research.

I was fortunate to have my years of study and research generously funded by different organizations. The University of Toronto fellowships partially funded my first years in graduate school. The ECE department at the University of Toronto offered a Viola Carless Smith research scholarship to support my research towards the end of my PhD. The Natural Sciences and Engineering Research Council of Canada funded my PhD studies through a three-year Canada Graduate Scholarship. The full support offered by this scholarship gave me the courage to keep on going when I found myself “advisorless” upon the leave of my first PhD advisor. Google awarded me a Canada Anita Borg Memorial Scholarship in 2010. This award not only helped financially, but, more importantly, reinforced my confidence as a woman in a technical field. Google also generously sponsored my participation in the first Google GRAD CS forum and funded my travel to two Grace Hopper Celebration for Women in Computing conferences. Through CRA-W support, I participated in several mentoring and domain specific workshops. These forums, not only offered technical breadth, but helped me find mentors and close friends that I have kept in touch with for years. IBM awarded me an IBM PhD Fellowship in 2010. In addition to the financial support, the fellowship facilitated a summer internship with the research division of IBM at the T.J. Watson lab. The summer I spent at T.J. Watson strongly influenced my decision to join IBM Research upon finishing my PhD.

Although they delayed my graduation, I think of my industry internships as eye-opening with respect to applied research and high-impact projects. Anne Bracy made my six-month Intel internship seem short. I thank her for the technical support, mentoring and the friendship that lasted more than the Californian adventure. During my internship with IBM, I had the opportunity to meet numerous researchers and exceptional individuals that made me realize that I would really enjoy working in such an environment. My mentors at IBM, Viji Srinivasan, Brian Prasky and Alper Buyuktosunoglu, all contributed to a productive and fun summer of research. I would like to thank them for all their support, but, especially for pushing me to present my summer work to the System z processor design team.
Among the IBMers, I would like to extend special thanks to Jaime Moreno and Dilma da Silva. Jaime played an instrumental role in my decision to join IBM Research. I am not sure I would have chosen IBM as my first job out of graduate school, if it were not for his guidance and mentoring. Dilma da Silva encouraged me to apply for an IBM PhD Fellowship when she saw my research presentation during one of the workshops she organized. That encouragement lead to a series of fortunate events that culminated with my joining IBM Research.

I would like to thank my initial PhD advisor, Prof. Michael Voss, for instigating my curiosity on compilers and runtime systems. The course on optimizing compilers that I took with Michael remains the most dear course to me throughout my whole education.

Since she joined the faculty, Prof. Natalie Enright Jerger tirelessly showed her support and offered her advice. I am especially grateful for her care and continuous mentoring. I hope she will keep her passion of mentoring technical women alive and strong through the years to come!

Kelly Chan, Judith Levene, and Darlene Gorzo helped with all the administrative paperwork. Judith and Darlene always welcomed my questions and requests with a smile. Kelly demystified all forms I had to fill in and made paperwork seem easy and straightforward.

All my colleagues at University of Toronto contributed to my learning and to improving my research skills. Thank you! The initial painful days of working with a full-system simulator, launching thousands of simulations and keeping track of the cluster machines were sweetened by Jason Zebchuk’s assistance. I thoroughly appreciate your help conquering the simulator and the cluster machines! After a while, I am sure I will miss even your cynical sense of humor. Patrick Akl, Elham Safi, Myrto Papadopoulou, Maryam Alvandi, Kaveh Aasaraai, Islam Atta, Ian Katsuno, Goran Narancic, Patrick Judd, and the rest of my research colleagues, they all made my days at University of Toronto more fun.

During my graduate school, not only I learned from my colleagues, but I gained true friends that I will keep in touch for years to come. Among them, I would like to mention Myrto Papadopoulou. She has been a tireless source of encouragement, care and support, both technical
and emotional. I consider myself fortunate to have met her at University of Toronto. In the last years of my PhD, my path crossed with that of Danyao Wang, Alma Juarez-Dominguez, Inmar Givoni, and Henry Wong. I truly enjoyed our stimulating conversations and appreciated your cheering during the last stretch of my work. Thank you for your support that came in so many different flavors: from Skype and in-person chats to crossing the US border with me to get my paperwork in order.

My parents had a lot of patience with my graduate school and showed their care and support, although far away. Being a computer scientist is all my mother’s doing. Throughout the years, she instigated and cultivated my love of learning and, in particular, she was the one that made me enroll in the Computer Science and Engineering undergrad in Bucharest, Romania. I would like to thank my father for training my ambition throughout my childhood and always “betting” on me, even when he did not fully understand what he was betting on. I thoroughly appreciate my parents’ faith in me, their unconditional love and the freedom they offered me to do whatever I pleased at any point in my life.

As I mentioned earlier, this thesis documents just a small part of my PhD adventure. Irrespective of all financial, technical and emotional support I have got along the way from others, there were plenty of moments when giving up seemed the only sane option. I would not have succeeded if it were not for my life partner, Livio. Serendipity brought us both to University of Toronto, and I am still bewildered that we met. Livio was the first to hear my research ideas, brainstorm with me on technical details and help me crystallize my own thoughts. He always supported my endeavors, even if it implies sacrifices on his part. When I found myself in doubt, he patiently explained the importance of the research problems I was working on and helped me regain confidence in the value of my research and my skills as a researcher. Countless times, our dinner time transformed into a recap of our research day with heated conversations on predictor virtualization and exception-less system calls. Recently, someone I admire and respect said this: “It is harder to find a great research collaborator than it is to find a great life partner”. How fortunate of me to have found both in the same person! Our PhDs are only the
first project in a life-long collaboration. I am sure the best is yet to come!

I wish I could share this accomplishment with Prof. Irina Athanasiu, who is no longer among us today... Irina was my mentor during my undergrad long before I understood the importance of mentoring. I wish I had listened to her advice more often.

I dedicate this thesis to the loving memory of Maia, who raised me since I was three days old and undoubtedly contributed the most to the person I am today.
## Contents

1 Introduction ................................. 1
   1.1 Thesis and Dissertation Goals .................................. 4
   1.2 Thesis Overview ................................................. 4
       1.2.1 Motivating Trends ......................................... 5
       1.2.2 Predictor Virtualization .................................... 5
       1.2.3 A Virtualized Data Prefetcher ............................... 6
       1.2.4 Virtualized Branch Target Buffers ......................... 7
       1.2.5 Virtualized Object-Pointer Prefetchers for Managed Runtimes 9
   1.3 Thesis Contributions ................................. 10

2 Background .................................. 12
   2.1 The Importance of Speculative Optimizations .................. 13
   2.2 Application Memory Footprints ................................. 16
   2.3 Trends in On-Chip Processor Cache Sizes ...................... 17
   2.4 Virtualization - A General Technique in Computing Systems .......... 18

3 Predictor Virtualization .................. 20
   3.1 Predictor Virtualization Architecture ............................ 20
       3.1.1 PVProxy and PVCache ...................................... 22
       3.1.2 PVTable .................................................. 24
   3.2 Advantages of Virtualizing Hardware Predictors .................. 26
   3.3 Discussion: To Virtualize or Not To Virtualize .................. 28
       3.3.1 Prediction Latency ........................................ 28
       3.3.2 Impact on the Memory Hierarchy ............................. 29
   3.4 Related Work .................................................. 31
       3.4.1 Reconfigurable Caches ..................................... 31
       3.4.2 Unified Microprocessor Core Storage ......................... 32
       3.4.3 Memory Hierarchy Used for Predictor Storage ............... 33
       3.4.4 Follow-up Research ....................................... 35
   3.5 Conclusions ................................................. 35

4 Virtualizing A State-of-the-Art Data Prefetcher .......... 37
   4.1 Background: Spatial Memory Streaming .......................... 38
       4.1.1 Spatial Memory Streaming Implementation .................. 40
   4.2 Virtualizing Spatial Memory Streaming .......................... 42
## Virtualizing Branch Target Buffers

1. **Background: Branch Target Buffers** .............................................. 67
2. **Phantom-BTB: A Virtualized Branch Target Buffer Design** .......... 68
   1. Phantom-BTB Design .......................................................... 70
   2. Implications of BTB Virtualization for the On-Chip Memory Hierarchy ... 75
   3. Phantom-BTB versus Conventional Branch Target Buffers .......... 76
3. **Experimental Analysis** ............................................................. 77
   1. Methodology ................................................................. 78
   2. Benchmarks ................................................................. 79
   3. Experimental Results ..................................................... 80
   4. Hardware Overhead for Phantom-BTB ..................................... 92
4. **Related Work** ........................................................................ 93
   1. Research in Branch Prediction ............................................. 93
   2. EXACT: Explicit Dynamic-Branch Prediction with Active Updates ... 95
5. **Concluding Remarks** ............................................................. 96

## A Virtualized Object Pointer Prefetcher

1. **Software Runtime Assisted Object Pointer Prefetching** .............. 101
   1. Extending the Java Runtime ............................................. 102
   2. Object Metadata Table ..................................................... 103
   3. Pointer Prefetcher ......................................................... 106
2. **Class Oriented Object Metadata Filtering** ................................. 107
3. **Experimental Evaluation** ...................................................... 108
   1. Experimental Methodology .............................................. 109
   2. The potential of software runtime assisted pointer prefetching .... 111
   3. The performance of a practical pointer prefetchers ................ 113
   4. The potential of object metadata filtering ............................ 115
   5. Distribution of useful pointer fields .................................. 116
   6. Class oriented object metadata filtering .............................. 118
   7. Object Pointer Prefetching and State-of-the-Art Prefetching ...... 121
4. **Related Work** ....................................................................... 122
   1. Prefetching Linked Data Structures ................................... 122
   2. DPC: Data Prefetching Champion ....................................... 124
5. **Concluding Remarks** ............................................................ 125
7 Concluding Remarks

7.1 Thesis Summary .................................................. 128
7.2 Future Work ...................................................... 130

Bibliography .......................................................... 134
# List of Tables

4.1 Virtualizing SMS: Baseline system configuration ........................................... 46  
4.2 Virtualizing SMS: Benchmarks ........................................................................ 49  
4.3 Virtualizing SMS: Storage for different predictor configurations .................. 52  

5.1 Baseline system configuration ........................................................................... 78  
5.2 Benchmarks ........................................................................................................ 80  
5.3 BTB storage .......................................................................................................... 82  
5.4 Dedicated storage for different BTB configurations ......................................... 85  

6.1 Input parameters used for running the benchmarks .......................................... 110  
6.2 Percentage of new instructions introduced by the hybrid prefetcher ............ 111  
6.3 Dedicated on-chip storage requirements ............................................................. 115  
6.4 L2 cache statistics ............................................................................................... 120
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Processor Performance - Historical Data and ITRS Projection</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>The Evolution of On-Chip Processor Cache Sizes for different Intel processors</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Conventional Predictor-based Hardware Optimization</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Virtualized Predictor-based Optimization</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>Predictor Virtualization Architecture</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>Predictor Entry Address Mapping</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Spatial Memory Streaming</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Pattern History Table (PHT)</td>
<td>41</td>
</tr>
<tr>
<td>4.3</td>
<td>Virtualizing SMS</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Virtualized PHT: metadata Packing</td>
<td>43</td>
</tr>
<tr>
<td>4.5</td>
<td>Virtualized PHT: PVProxy components</td>
<td>44</td>
</tr>
<tr>
<td>4.6</td>
<td>Virtualized PHT: Memory Request Generation</td>
<td>45</td>
</tr>
<tr>
<td>4.7</td>
<td>Virtualizing SMS: Sampling Methodology</td>
<td>48</td>
</tr>
<tr>
<td>4.8</td>
<td>SMS performance potential</td>
<td>50</td>
</tr>
<tr>
<td>4.9</td>
<td>SMS performance potential - representative behavior</td>
<td>52</td>
</tr>
<tr>
<td>4.10</td>
<td>Performance of the Virtualized SMS Prefetcher</td>
<td>54</td>
</tr>
<tr>
<td>4.11</td>
<td>Percentage Increase in L2 Memory Requests due to Virtualization</td>
<td>55</td>
</tr>
<tr>
<td>4.12</td>
<td>Percentage Increase in Off-Chip Bandwidth due to Virtualization</td>
<td>56</td>
</tr>
<tr>
<td>4.13</td>
<td>Off-Chip Memory Traffic Increase (Application and Predictor Data)</td>
<td>57</td>
</tr>
<tr>
<td>4.14</td>
<td>Off-Chip Bandwidth Increase for Different L2 Cache Sizes</td>
<td>58</td>
</tr>
<tr>
<td>4.15</td>
<td>Performance of the Virtualized Predictor with Higher L2 Latency</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Instruction Execution without and with Branch Prediction</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>Branch Target Buffers: Branch Target Prediction</td>
<td>68</td>
</tr>
<tr>
<td>5.3</td>
<td>Phantom-BTB Concept</td>
<td>69</td>
</tr>
<tr>
<td>5.4</td>
<td>Phantom-BTB: Exploiting program path repetition</td>
<td>70</td>
</tr>
<tr>
<td>5.5</td>
<td>Phantom-BTB Design</td>
<td>71</td>
</tr>
<tr>
<td>5.6</td>
<td>Phantom-BTB: Temporal Groups</td>
<td>73</td>
</tr>
<tr>
<td>5.7</td>
<td>Phantom-BTB: Temporal Group Indexing</td>
<td>74</td>
</tr>
<tr>
<td>5.8</td>
<td>Branch Target Buffers: Accuracy versus Capacity</td>
<td>81</td>
</tr>
<tr>
<td>5.9</td>
<td>Phantom-BTB: Temporal Group Indexing</td>
<td>83</td>
</tr>
<tr>
<td>5.10</td>
<td>Phantom-BTB Accuracy</td>
<td>84</td>
</tr>
<tr>
<td>5.11</td>
<td>Phantom-BTB Performance</td>
<td>85</td>
</tr>
<tr>
<td>5.12</td>
<td>Phantom-BTB: L2 misses before and after virtualization</td>
<td>87</td>
</tr>
</tbody>
</table>
5.13 Phantom-BTB: L2 accesses before and after virtualization ............. 87
5.14 Phantom-BTB Sensitivity to the Virtual Table Size ..................... 89
5.15 Phantom-BTB: Sensitivity to the L2 cache size ......................... 90
5.16 Phantom-BTB performance with larger dedicated BTBs ............... 91
5.17 Phantom-BTB compared to PV-BTB ........................................ 92

6.1 Pointer link ............................................................................. 103
6.2 Software runtime assisted object pointer prefetching .................. 104
6.3 Virtualization is applied to the object pointer metadata ............... 105
6.4 The performance improvement potential for a specialized pointer prefetcher ... 112
6.5 The performance of a large, virtualized pointer prefetcher ............ 113
6.6 The performance improvement of useful pointer links .................. 116
6.7 The distribution of pointer fields for useful prefetches ................. 117
6.8 Performance improvement with pointer link filtering .................. 119
6.9 Off-chip bandwidth before and after virtualization .................... 121
6.10 Prefetching champion run without and with object pointer prefetching .... 122
Chapter 1

Introduction

The computing industry experienced an exponential increase in application performance across the decades [1]. One key component responsible for this performance is the processor. Transistor scaling allowed packing an unprecedentedly high number of fast-switching transistors on chip that enabled designing high performance processors. However, due to thermal and power issues, technology scaling cannot sustain the trend of exponential performance for the upcoming years. On the other hand, given the complexity of current and future systems, software innovation alone cannot supply the increase in performance demanded by emerging applications. We strongly believe that, in the foreseeable future, only disruptive innovation across multiple layers of the computer stack can help continue the trend of application performance. In particular, a stronger collaboration between the hardware and software layer can break some of the current limitations in performance.

The performance of current processors heavily depends on hardware optimizations. As the performance resulting from technology scaling alone reached a plateau, hardware optimizations take a central role in processor performance. One fundamental class of hardware optimizations is speculative optimizations. Modern processors employ a wide variety of speculative optimizations that are predictor-based, including data prefetching, branch prediction and cache replacement policies [2, 3, 4, 5]. At the heart of speculative optimizations is a predic-
tion mechanism that allows processors to anticipate future application behavior and optimize accordingly.

As the application runs, the processor observes the application behavior and collects application metadata that is used in identifying patterns in the execution of the applications. Correlation among the application metadata is used to predict future application patterns and optimize for these patterns.

The application metadata is stored in on-chip lookup tables (called predictor tables hereafter). The amount of metadata used by the optimization is limited by the on-chip storage dedicated to implement the predictor table. On the one hand, the performance of the hardware optimization depends on the capacity of the predictor table. It is well-documented that utilizing more application metadata leads to higher prediction accuracy, and, in consequence, to better application performance [2, 6, 7]. On the other hand, while numerous, on-chip resources are still limited and orchestrating these resources across multiple optimizations is one of the key concerns of processor designers.

Traditionally, processor architects employ a "one-size-fits-all" approach to build hardware predictors, allocating a fixed portion of the on-chip resources to the predictor tables. As a consequence, processor designers have to strike a balance between which hardware optimizations to implement in a processor and the amount of resources to be dedicated to a specific optimization. This trade-off often leads to sub-optimal designs where: 1) resources are wasted for applications that do not benefit from a particular predictor or only require small predictor tables, or 2) predictors under-perform for applications that need larger tables, which cannot be built due to area-latency-power constraints. For multi-core processors, this issue of finding the "one-size-fits-all" design point is exacerbated, since the resources dedicated to the predictor tables multiply by the number of cores on chip.

This dissertation introduces “Predictor Virtualization” (PV), a technique that breaks the limitations of the traditional “one-size-fits-all” design for hardware predictors. PV utilizes the conventional memory hierarchy to emulate large hardware predictors that dynamically adapt to
application demand. On-chip caches have become sufficiently large to accommodate allocating a small percentage of their resources for caching predictor metadata. Thus, instead of dedicating large on-chip predictor tables, PV uses the entire memory hierarchy to store application metadata. The allocation of on-chip cache capacity for metadata is performed on demand, adjusting to the application’s need for predictor resources. As a result, PV is a “pay-as-you-go” technique that allows the application metadata to dynamically share on-chip cache capacity with the application data and instructions.

PV increases the utility of traditional caches: in addition to being accelerators for slow off-chip memories, on-chip caches are leveraged for increasing the effectiveness of predictor-based hardware optimizations. The proposed technique is non-intrusive and requires minimal additions to the memory hierarchy to allow predictor requests to be propagated through the hierarchy starting at the last level on-chip caches.

This dissertation demonstrates that predictor virtualization can greatly improve the cost-performance trade-off of two metadata-based performance enhancing techniques: data prefetching [8] and branch target address prediction [9]. Full-system, cycle-accurate simulations indicate that the virtualized data prefetcher performs within 1% of the original prefetcher, while reducing the dedicated resources from 60KB down to less than one kilobyte. Similarly, a virtualized branch target buffer design with 1K dedicated entries achieves close to the performance of a conventional design with 4K-entries, while using 3.6 times less dedicated storage.

One significant advantage of predictor virtualization is the opportunity to expose the predictor interface to the software layer. The software layer (e.g., the compiler, the runtime system) benefits from a macroscopic view and semantic information that are lost at the hardware level. Since PV allows for large metadata storage without the expensive cost, the software layer can communicate complex metadata to be used in the prediction process. As a case study, this dissertation presents the design of an object pointer prefetcher for managed languages, in which the runtime environment is entirely responsible for generating predictor metadata, while the hardware exploits this metadata to increase application performance with reduced hardware
1.1 Thesis and Dissertation Goals

Our thesis is that processor memory hierarchies have become sufficiently large to allocate portions of their capacity to store application metadata, in addition to application data and instructions. The focus of this dissertation is to demonstrate how memory hierarchies can be leveraged to implement more efficient hardware optimizations. To this end, this thesis introduces “Predictor Virtualization”, a framework that uses the traditional processor memory hierarchy to store application metadata used in speculative hardware optimizations. This allows to emulate large, more accurate predictor tables, which, in return, leads to higher application performance. As a proof of concept, this technique is applied to three different hardware optimizations: a data prefetcher [8], branch target buffers [9] and an object-pointer prefetcher [10]. The next section presents an outline of the dissertation.

1.2 Thesis Overview

In Chapter 2, we provide an overview of computing trends that are directly related to the research presented in this thesis. Chapter 3 describes the “Predictor Virtualization” framework. Chapters 4, 5, and 6 discuss how virtualization can be applied to three distinct hardware-optimizations: a data prefetcher, branch target buffers and an object-pointer prefetcher, respectively. Each chapter also outlines the work related to the research presented in that particular chapter. Our concluding remarks and future research directions are presented in Chapter 7. The rest of this section presents a short overview for each thesis chapter.
1.2.1 Motivating Trends

Chapter 2 focuses on several computing trends that directly motivate the research presented in this thesis.

The importance of speculative optimizations is well-documented in the literature, with many examples of techniques that vary in scope and applicability [2, 3, 4, 5, 11, 12, 13]. As processor performance derived from technology scaling reaches a plateau, hardware optimizations become even more important in the quest for improving the flattening application performance. This motivates our focus on breaking the on-chip limitations of predictors used in hardware optimizations.

To better understand how serious these limitations are, we examine the trends in application data and instruction footprints for benchmark suites used to evaluate processor performance. Based on these trends, we project that application data and instruction footprints will continue to grow, straining the resources currently used in hardware optimizations.

To support our thesis that on-chip cache hierarchies have become sufficiently large to store application metadata in addition to data and instructions, we report on trends observed for on-chip processor cache size. In particular, we show how the capacity of on-chip caches has been increasing along the years, with projections that cache sizes will reach 80MB in the near future [14].

In the end of Chapter 2, we briefly discuss virtualization as a general technique used in computing systems, which inspired us in developing the work presented in this thesis.

1.2.2 Predictor Virtualization - A Framework for Predictor-based Hardware Optimizations

Chapter 3 presents Predictor Virtualization (PV), a technique that considerably reduces the dedicated on-chip resources necessary to implement predictor-based optimizations, while preserving their effectiveness. A key observation of this research is that on-chip processor caches
have become sufficiently large to accommodate allocating, on demand, a small percentage of their resources for caching application metadata. Thus, instead of dedicating large on-chip tables, PV stores predictor metadata in a virtual table allocated in the memory hierarchy, at cache line granularity. Predictor metadata is stored starting at the second level cache, to avoid the pollution of the performance-critical L1 caches. To maintain the illusion of a large, dedicated table, application metadata is retrieved on demand from the virtual table and stored in a small dedicated predictor cache that records a few active entries of the predictor. PV can be used to either reduce the dedicated resources for hardware predictors or to emulate large predictor tables that would be otherwise impractical to build.

One important challenge for PV is the higher prediction latency introduced by spilling metadata to the memory hierarchy. Some optimizations, such as data and instruction prefetchers, are tolerant of the higher prediction latency and, for this category of optimizations, virtualization is straightforward. Other optimizations, such as branch predictors, are latency-sensitive and the prediction mechanism needs to be redesigned with virtualization in mind to take advantage of its benefits.

To demonstrate the benefits of virtualization applied to hardware predictors, we present virtualized designs for three different optimizations and compare their performance to the original proposals. The optimizations include a state-of-the-art data prefetcher [2], conventional branch target buffers [15] and a novel object-pointer prefetcher. Our experimental results show that predictor virtualization is a viable design option to address the performance-cost trade-off in implementing hardware optimizations.

### 1.2.3 A Virtualized Data Prefetcher

Aggressive data prefetchers are tolerant to variable latency for predicting memory addresses to be prefetched, and, hence, represent a straightforward case study for predictor virtualization. For our study, we chose a state-of-the-art-prefetcher, *Spatial Memory Streaming (SMS)* [2], that proved to be effective for commercial applications. Commercial workloads are notorious for
their complex memory access pattern that is hard to predict with simple data prefetchers. SMSrecords spatial footprints observed in the application’s memory access stream and uses thesefootprints to predict future memory accesses.

The performance of the prefetcher comes at a high hardware cost. To be effective, theprefetcher needs ample storage to record the application memory footprints. The original studyfound that, to fully achieve the potential of the prefetcher, the storage needed is approximately80KB. To put things into perspective, this storage is higher than the resources dedicated to theL1 data caches in modern Intel processors [16].

In Chapter 4, we present the design of a virtualized SMS prefetcher. In the virtualized de-sign, the table that stores application memory footprints is conceptually moved to main mem-or. The predictor entries required to generate prefetches are retrieved on demand from thememory hierarchy. Regular memory requests are used to retrieve the application metadata,and, as a consequence, predictor entries are stored in the on-chip caches. This allows for lowerlatency than main memory latency in delivering the predictor entries to the optimization engine.

Full-system, cycle-accurate simulations show that a virtualized design maintains the per-formance of the original prefetcher, while requiring less than 1KB of dedicated space. Thesignificant reduction in dedicated resources, coupled with its high performance, makes thevirtualized SMS prefetcher a practical option to consider in processor implementations.

1.2.4 Virtualized Branch Target Buffers

Branch target buffers (BTB) have been utilized for decades in branch prediction, an omni-present processor optimization [15]. Branch prediction allows the processor to speculativelyexecute instructions past branch instructions, increasing instruction processing parallelism and,hence, application performance. To be effective, though, branch prediction accuracy needs tobe high, since the misprediction penalty for each branch consists of several tens of cycles,depending on the depth of the processor pipeline [16].

Until recently, a small BTB (1-2K entries) proved to be sufficient for delivering high accu-
racy predictions. Commercial applications challenge the design of conventional BTBs [17, 18, 19], since they require up to 16K entries to achieve high prediction accuracy [7]. In Chapter 5, we study the benefits of applying virtualization to conventional BTBs.

Processors need to provide a branch prediction every cycle, thus, the branch prediction mechanism is highly sensitive to the prediction latency. For such a predictor, it is not beneficial to wait for the prediction metadata to be retrieved from the on-chip low-level caches because the latency is prohibitively high. To take advantage of the extra storage offered by spilling application metadata to on-chip caches, we re-organize the virtual table such that its contents can be timely prefetched.

The virtualized BTB design relies on repetition and temporal correlation in the stream of BTB branch misses. The virtual table stores groups of temporally correlated branches that miss in the dedicated BTB. To facilitate timely prefetching, each group is associated with a region of code around an earlier branch that also missed in the BTB. Any branch miss within this code region triggers the prefetching of the temporal group. Provided repetition in the branch miss stream exists, some branches that are not present in the dedicated BTB anymore will be found in the temporal group retrieved from the on-chip cache.

With such a mechanism in place, the perceived capacity of the BTB is increased without allocating on-chip resources. Since the prediction accuracy of the virtualized BTB is higher, it appears as if the dedicated BTB can store a larger application footprint. For the benchmarks used in our study, the application performance for a virtualized BTB design with a 1K-entry dedicated BTB reaches the performance obtained by a 4K-entry conventional BTB. More importantly, applying virtualization to larger BTBs further improves the application performance, as long as the application instruction footprint does not fit in the dedicated BTB.

All actions in the virtualized BTB design are triggered by branch misses, which means that, for applications that do not require extra BTB capacity, the virtualization does not lead to any significant memory traffic. In contrast, for those applications that require more BTB resources, the virtualized design starts using the on-chip cache resources, adapting to the behavior of the
application.

We believe that the techniques used in the virtualizing BTBs are general, and they can be applied in virtualizing other latency-sensitive hardware predictors.

1.2.5 Virtualized Object-Pointer Prefetchers for Managed Runtimes

Predictor Virtualization allows to store a large amount of application metadata. This opens up the opportunity for a closer collaboration between the software layer and the underlying hardware. Usually, the software layer (e.g., compiler, runtime environment) benefits from higher-level information, such as data structures, object memory layout, object connectivity. This information is not available at the hardware level and, in general, it is expensive to recreate [6]. We believe that making the software layer responsible for collecting and communicating application metadata to the underlying hardware can lead to higher accuracy predictors. As a proof of concept, we propose a hybrid software/hardware object pointer prefetcher for managed languages.

Managed languages have increased in popularity due to their ease of programming, rich libraries and managed runtimes [20]. All these features usually come at a cost of lower application performance. We believe that a stronger collaboration between the runtime system and the underlying hardware can lead to a better utilization of hardware resources and, hence, higher performance.

In Chapter 6, we propose a software directed pointer prefetcher in which the runtime system is responsible for communicating pointer addresses and the links between objects to the underlying hardware. The hardware layer uses this object metadata to implement specialized pointer prefetchers. We study the potential of such prefetchers and propose a cost-effective virtualized hardware design for an object pointer prefetcher. In addition, we analyze the effect of selectively communicating pointer metadata to the hardware based on class profiling that is readily available at the runtime level, while impractical to extract at the hardware level. To achieve a low-cost hardware implementation, the object metadata used for prefetching is
spilled to the conventional memory hierarchy and retrieved only when needed.

The evaluation of the proposed prefector shows promising performance improvements on a series of pointer intensive benchmarks. For example, when run in conjunction with traditional prefetchers, the pointer prefetching scheme achieves 53% performance improvement for SpecJBB2005 compared to no prefetching, which represents an increase by 23% compared to traditional prefetching alone. In addition, we show that the hybrid prefector complements even high-performance hardware only prefetchers that are not yet implemented in current processors.

1.3 Thesis Contributions

The main contributions of this thesis are listed below:

- This thesis introduces Predictor Virtualization (PV), a framework that takes advantage of the existing processor memory hierarchy to emulate large predictor tables used in hardware optimizations. PV considerably reduces the amount of on-chip resources dedicated to the predictor tables, while preserving or improving the effectiveness of the original optimization. To implement PV in real silicon, only simple additions to the memory hierarchy are required to be able to inject metadata memory requests to last level caches.

- This thesis demonstrates the benefits of virtualization when applied to a state-of-the-art data prefector, the Spatial Memory Streaming (SMS) [2]. The thesis presents a virtualized SMS design and compares its performance with the original proposal. Using a full-system, cycle-accurate simulator to model a quad-core architecture, and a set of eight commercial benchmarks, experimental results show that the virtualized prefector matches the performance of the original scheme within 1% on average, while reducing the dedicated on-chip predictor table from 60 kilobytes to less than one kilobyte per each core.
This thesis presents a virtualized design for conventional branch target buffers (BTB) [15] that employs metadata prefetching to increase the perceived capacity of the BTB. A virtualized 1K-entry BTB improves application performance by up to 12.7% when compared to the equivalent conventional BTB. The hardware overhead of the virtualized design is 8% over the conventional BTB. The application performance obtained with the virtualized design is similar, on average, to the performance obtained with a 4K-entry BTB, while the dedicated storage is 3.6 times smaller. Experimental results show that virtualization remains effective when applied to larger dedicated BTBs, as long as the applications benefit from additional BTB resources.

This thesis proposes a hybrid software/hardware object pointer prefetcher for managed languages. The novelty of the prefetcher is two-fold. First, the runtime system is responsible for collecting and communicating application metadata to the underlying hardware. Second, the hardware employs virtualization to implement a cost-effective object pointer prefetcher. Experimental results show that the pointer prefetcher improves application performance when run in conjunction with traditional prefetching schemes. For example, when run with next line and striding prefetchers, the pointer prefetching scheme achieves 53% performance improvement for SpecJBB2005 compared to no prefetching, which represents an increase by 23% compared to traditional prefetching alone.
Chapter 2

Background

The purpose of this chapter is to provide a brief perspective on how the work described in this thesis fits into the larger domain of computer architecture, with an emphasis on its importance and motivating trends. Subsequent chapters describe the prior work that is closely related to the ideas presented in that chapter.

To better understand the benefits of virtualizing hardware predictors, we first discuss the importance of speculative optimizations in computer architecture and the role of hardware predictors in delivering application performance. We then show evidence of increased application footprints, which motivates the design of larger predictor tables to obtain higher accuracy speculative optimizations. Next, we present existing trends for on-chip processor caches and discuss the continuation of these trends in the foreseeable future. In the end, we briefly note that virtualization is a general technique that has been used in computing since its early inception and motivate the use of virtualization in the design of hardware predictors.
2.1 The Importance of Speculative Optimizations and Hardware Predictors in Modern Processors

Technology scaling allowed an unprecedentedly high number of transistors to be built on the same die. This led to a myriad of micro-architectural techniques that translate this transistor density to application performance. However, in recent years, due to thermal and power constraints, the benefits of technology scaling have stopped, leading to a flattening curve in processor performance. For reference, we show in Figure 2.1 how processor performance varied in the last decades, together with the desired performance curve and the anticipated one [1]. The y axis in the graph is logarithmic, which means that the almost straight performance line represents exponential performance improvement. This high performance enabled a series of discoveries and developments in domains that now depend on computing power. To sustain the same type of advances, it is desirable to maintain the same type of computing performance for the future. In reality, the semiconductor industry has experienced a drastic flattening in processor performance, which is foreseen to continue in the near future. In this context, the importance of micro-architectural optimizations is increased, since only through optimizations some of the lost performance can be recovered.

One of the main techniques used in micro-architectural optimizations is speculative execution. Speculative optimizations have become mainstream techniques that are used to eliminate restrictions imposed by existing dependencies, thus increasing processing concurrency and, in consequence, processor performance.

There are numerous speculative optimizations present in current microprocessors, such as data and instruction prefetching, branch predictors, and memory dependence predictors. The body of research in this domain includes work that, to the best of our knowledge, has yet to be incorporated in current processors, such as value prediction [5] and instruction reuse [4]. These speculative optimizations considerably improve application performance. For example, depending on the application, instruction and data prefetchers can improve performance by
Figure 2.1: Processor performance as measured by a standardized benchmark suite and ITRS projection for future performance [1].

orders of magnitude [2].

In general, the effects of hardware optimizations are additive, for the most part. Thus, even if several optimizations seem to have a low impact (i.e., an improvement in the range of 5-10% speedup), the ability to include several such optimizations in the same processor chip can lead to considerable increase in performance (i.e., in the range of the sum of all the considered optimizations). Thus, it is vital to have the ability to include as many optimizations as possible in the same processor design.

The success of speculative execution depends on two key aspects:

- the ability to accurately predict a certain event
- the ability to recover quickly in case a misspeculation occurs (in case the speculation impacts the architectural state of the processor)

In this thesis, we are concerned with the first aspect, the ability to accurately predict events. In general, the prediction mechanism used in speculative optimizations involves several steps:
first, at runtime, the behavior of the application is monitored, extracting patterns for the events that are under observation; second, these patterns are stored in some type of look-up structures, such as buffers or dedicated cache structures; third, these patterns are used to predict future application behavior and optimize for future execution accordingly. For simplicity, we use the term *hardware predictors* or, in short, *predictors* to refer to the look-up tables used to store application information in speculative optimizations.

The information stored by the predictor refers to the application behavior. Since this information is auxiliary and it is not directly manipulated by the application, we use the term *application metadata*, or, in short, *metadata* to refer to the application behavior data that is collected by the hardware predictor. Since the application does not use the predictor metadata, this metadata is not required for application correctness. The metadata is used only in the prediction mechanism and it can be discarded without affecting application correctness. Discarding metadata can only affect application performance.

Application metadata can take various forms, depending on the speculative optimization using it. For example, branch targets, branch directions and program paths are generally used as metadata in branch prediction. To anticipate future memory accesses, prefetchers monitor application memory access and try to establish correlation between addresses and instructions that access them.

To establish patterns in the application behavior, some type of correlation is necessary for the application metadata. Typical correlations involve instruction addresses, data addresses, and compact representations of history of pertinent events in the system (e.g., the direction of the most recent branches). In general, the correlations used by the predictors influence the amount of metadata storage required for obtaining high prediction accuracy. As correlation over instruction and data addresses is considerably common, we are interested in observing trends for data and instruction footprints. The historical trend in application footprints shows that they increase significantly over the years. In consequence, predictor tables need to scale accordingly to sustain the prediction accuracy necessary for high application performance.
the next section, we present historic trends for application footprint growth.

2.2 Application Memory Footprints

One important characteristic of applications is the amount of data and instructions that they operate on during their execution. This is usually referred to as the application footprint. The increase of computational resources naturally enables more complex applications with larger footprints. In this section we present experimental evidence to support the claim that application data and instruction footprints have been increasing over the years. We expect this trend to continue in the near future.

As application performance became important, the necessity of comparing the performance of computing systems in a standardized way proved essential. For this purpose, the Standard Performance Evaluation Corporation (SPEC) was founded in 1988 with the mission of producing a standardized set of benchmarks for analyzing computing systems performance. In particular, SPEC produces and maintains a set of benchmarks called SPEC CPU that are meant to be used in the analysis of processor performance. These benchmarks are widely employed not only by the research community, but also by industry vendors to measure and compare the performance of their processors.

Since these benchmarks are utilized to analyze processor performance, we find the applications included in the SPEC CPU benchmark suite to be reasonable indicators of application characteristics in general. The SPEC CPU benchmark suite contains two sets of benchmarks: SPEC CPU INT, used for analyzing integer-dominant performance and SPEC CPU FP, which is used for analyzing floating-point performance.

Gove et al. present a detailed comparison between the memory footprints of the SPEC CPU 2000 and 2006 benchmark suites [21]. The conclusion of the study is that the memory footprints increased considerably in the 2006 suite. For example, only 40% of the 2006 floating point benchmarks would fit in the memory requirements of the 2000 benchmark suite.
Similarly, only 60% of the 2006 integer benchmarks would fit in the memory requirements of the 2000 suite. In the 2006 benchmark suite, there are benchmarks that require up to 1GB of memory space, while all benchmarks of the 2000 suite fit in 256KB of memory space. Similar conclusions are reached by Lin et al. in their study on the memory and cache behavior for both SPEC CPU 2000 and 2006 [22].

We believe that the trends experienced in the releases of the SPEC CPU benchmarks are a mirror of the trends of existing applications. In addition, commercial applications are, in general, even more demanding than those found in the SPEC CPU suites. As explained earlier, larger application footprints demand larger capacity predictors for high accuracy speculative optimizations. We next examine the trends in on-chip cache sizing over the last decades and extrapolate from existing data to anticipate future trends.

### 2.3 Trends in On-Chip Processor Cache Sizes

The disparity between processor and memory speed has increased considerably over the years. The processor-memory speed gap reached hundreds of cycles in current processors. On-chip processor caches are highly utilized to create the illusion of a faster off-chip memory.

Figure 2.2 plots on-chip cache capacity for Intel processors over the last decade [14]. In the beginning, the cache size increased slowly. In recent years, growing processor-memory speed gaps and larger application footprints motivated adding more cache on-chip, especially when considering the performance-power trade-off. For reference, one of the latest processors from Intel offers a shared on-chip cache of 24MB. This is not a trend specific to Intel, but observed across processor designers and vendors. Intel experts predict the on-chip cache capacity to continue increasing over the next years, reaching a ball-park of 80MB by 2018 [14].

While the increased on-chip cache capacity improves application performance, it exhibits diminishing returns. This observation encourages novel uses of this massive on-chip storage. The key observation in our research is that on-chip caches can accommodate storing application
Before introducing our novel usage of on-chip caches, we briefly discuss virtualization as a general technique used in different domains related to computing systems.

### 2.4 Virtualization - A General Technique in Computing Systems

Virtualization has been widely used in different contexts of computing systems, such as computer architecture, operating systems, and programming languages and compilers. As a general concept, virtualization decouples the interface exposed by a system or resource from its physical implementation.

Virtual memory [23] is probably the most widespread application of virtualization. Virtual memory creates the illusion that an application has full control over a single, contiguous address space, which is, in general, much larger than the available physical memory. In reality, virtual memory is implemented through virtualization on top of physical memory, which is shared by multiple applications or tasks. Moreover, contiguous regions of virtual memory are

---

Figure 2.2: The evolution of on-chip processor cache sizes for Intel processors (manufacturing process technology shown on the x axis) [14])

metadata to enable the emulation of larger predictor tables used in hardware optimization.
not necessarily contiguous in physical memory. The operating system is responsible for mapping the virtual memory of an application to physical memory and for managing its memory requests. In this context, virtualization allows for a more flexible utilization of the physical resources, while making the job of the programmer easier. The physical memory can be used by several processes at the same time and each process manages its memory references as if it is the only user of the physical memory.

Not only resources can be virtualized, but also entire systems. Resources of one computing system can be virtualized and exposed as multiple computing systems. One advantage of system-level virtualization is that multiple OS instances can coexist simultaneously on the same host hardware. In recent years, system virtualization has been used to run several server applications on the same host machine in complete isolation.

In this thesis, we investigate the application of virtualization to predictors used in speculative hardware optimizations. The necessity of higher performance micro-architectural optimizations, the ever-growing application footprints and the availability of large on-chip caches motivate the virtualization of hardware predictors. Predictor virtualization takes advantage of the on-chip memory resources to emulate large predictor tables for increased accuracy and performance. As with other instances of its application, virtualization allows for a more flexible utilization of resources, while preserving the interface of the hardware predictor.

The next chapter details the key ideas behind predictor virtualization. Subsequent chapters present different designs for virtualized predictors with an experimental analysis of their performance and impact on the memory hierarchy.
Chapter 3

Predictor Virtualization

Hardware predictors are an essential component in speculative optimizations. As application footprints grow, the predictor storage needs to scale for optimizations to remain effective. In this chapter, we describe a novel hardware technique called predictor virtualization (PV) that addresses the need for larger predictor tables in hardware optimizations.

The key feature of predictor virtualization is that it enables hardware optimizations to benefit from large predictor tables without using precious on-chip resources. Instead of storing all predictor metadata in a fixed, dedicated on-chip table, PV stores the predictor table in the memory hierarchy. To maintain the illusion of a large, dedicated table, PV delivers predictor entries on-demand to a small cache that is tightly-coupled to the processor. In this chapter, we describe the predictor virtualization framework, and discuss its advantages, challenges and limitations. To better understand the contributions of this technique, we also present work that is related in motivation and spirit to the predictor virtualization approach.

3.1 Predictor Virtualization Architecture

Conceptually, most hardware optimizations can be split into an optimization engine that contains the logic of the optimization and a predictor table that stores the application metadata used by the optimization engine. For example, in the case of prefetchers, the optimization engine
discovers patterns in the application memory access stream and stores these patterns in some type of look-up table (i.e., the predictor table). Subsequently, based on the stored patterns, the optimization engine predicts application memory accesses and fetches the data from the memory hierarchy before the application needs it.

![Figure 3.1: Conventional prediction-based hardware optimization with a well-defined interface between the optimization engine and the predictor storage.](image)

In a traditional hardware optimization design, the predictor table is implemented as dedicated storage on chip. This dedicated storage is accessed by the optimization engine through a well-defined interface, as shown in Figure 3.1. The well-defined interface decouples the functionality of the predictor from its implementation. Thus, the predictor table can be “virtualized”, as shown in Figure 3.2, by allowing predictor information to be stored in the memory hierarchy.

In the virtualized architecture shown in Figure 3.3, the predictor table, instead of being dedicated on chip, is allocated in physical memory (PVTable). The optimization engine and the interface with the predictor table remain unchanged. To maintain the illusion of a dedicated table, PV delivers predictor entries on-demand to a small cache (PVCache in the figure) that is tightly-coupled to the optimization engine. If the optimization engine requests metadata that is already present in the PVCache, the request is serviced similarly to the non-virtualized design. On a PVCache miss, the metadata is retrieved from wherever it sits in the memory hierarchy.
The main difference between the virtualized and the traditional design is that the requests from the optimization engine are serviced with metadata that can reside anywhere in the memory hierarchy, not only in the dedicated table.

Compared to a traditional design, predictor virtualization introduces three new components: the PVProxy, the PVCache and the PVTable. The next sections describe these components in more detail.

### 3.1.1 PVProxy and PVCache

The interface between the optimization engine and the original predictor table is preserved in the virtualized design. The requests from the optimization engine are intercepted by the PVProxy. The PVProxy provides the optimization engine with the same two basic operations as in the original design: 1) store an entry in the prediction table, and 2) retrieve an entry for prediction.

The PVProxy uses a small, dedicated structure (PVCache) that stores predictor metadata that have been recently created or used. Whenever the PVCache is not sufficient to store the application metadata produced by the optimization engine, predictor entries are spilled to the
Figure 3.3: The architecture of the virtualized predictor with three new components: PVProxy, PVCache, and PVTable

conventional processor cache hierarchy and, optionally, backed up in physical memory space in the PVTable.

When the optimization engine requests a predictor entry, the PVProxy checks whether the requested entry is present in the PVCache. On a hit in the PVCache, the request from the optimization engine is serviced as in the non-virtualized design. On a miss, the PVProxy initiates a request to retrieve the predictor entry from the memory hierarchy.

The PVProxy contains a set of registers that keep track of the outstanding predictor requests (i.e., predictor requests that miss in the dedicated PVCache). These registers are similar to the miss-handling status registers (MSHRs) used in conventional lockup-free caches [24].

When the reply for an outstanding request arrives, all predictor entries brought from the memory hierarchy are installed in the PVCache. The request from the optimization engine is completed as usual. In addition, the MSHR allocated for the metadata memory request is freed.

The spilling of the predictor metadata can be performed starting at any level in the memory hierarchy. Usually, first level processor caches (L1 caches) are small and performance-critical. Thus, to avoid pollution and contention at the first level caches, the predictor metadata is allocated in the memory hierarchy starting at the second level cache (L2 cache), as shown in
Figure 3.3.

The memory requests generated by the PVProxy do not differ from the requests generated by the L1 caches and, thus, the memory hierarchy is oblivious to the predictor metadata it transfers and stores. The only modification required to the memory hierarchy is to allow the PVProxy to communicate with the L2 cache. The interface between the PVProxy and the on-chip caches is similar in nature to the one used by prefetchers or hardware page walkers that fill in TLB entries. These type of components are omnipresent in current processors, hence, we believe that the addition of the PVProxy does not lead to a penalty on the cache access latency.

3.1.2 PVTable

The PVTable contains the predictor metadata and is stored in physical memory space. There are multiple alternative design options for the predictor table. One option is to reserve a portion of the physical address space to be used by the PVTable. A second option is to rely on operating system (OS) support to manage a set of either virtual or physical addresses to map the contents of the PVTable to physical memory.

The research in this thesis adopts the first option: a small chunk of the physical memory addressing space is reserved for allocating and accessing the PVTable. This option is non-intrusive, as it does not require either operating system support, nor changes to the memory hierarchy. The start address for the PVTable is fixed and the physical memory space dedicated to the predictor is reserved, without declaring the corresponding physical addresses to the OS. Thus, the OS is oblivious to the physical space dedicated to the predictor. For the type of processors targeted in this study, on-chip processor caches starting at secondary levels are indexed and tagged using physical addresses. Thus, the PVProxy uses the corresponding physical addresses to allocate or retrieve portions of the PVTable from the memory hierarchy.

For predictor metadata transfer and storage, it is desirable that several predictor entries be packed in a memory block equal to the size of a cache block for two reasons. First, the footprint of the predictor metadata in the caches is reduced. Second, multiple predictor entries can be
brought in with one memory request. Consequently, the latency of fetching a predictor entry from memory can be amortized over several predictions, provided locality in the predictor access stream exists. In addition, packing predictor entries in one cache block lowers the additional bandwidth needed for predictor metadata traffic.

Figure 3.4 shows how predictor entries are mapped to memory addresses to install/retrieve portions of the $PVTable$ in/from the memory hierarchy. Several predictor entries are packed in a cache block, and, in consequence, the $PVTable$ is allocated in the memory hierarchy at cache line granularity. One predictor entry belongs to a certain index in the $PVTable$. In the figure, a $PVTable$ entry is shown to fit in a memory block equal in size to the processor cache block size. The memory address corresponding to a particular predictor entry is computed by adding the $PVTable$ index multiplied by the cache block size to the start address of the $PVTable$. Assuming a 64-byte cache size, the multiplication is equivalent to padding the index with six zeros, as shown in the figure.

As a design option, the main memory back-end storage can be completely eliminated from the virtualized architecture. This option requires the on-chip caches to be aware of the predictor metadata they store and transfer. This can be easily implemented either by storing an extra tag bit to indicate whether the cache line stores application data or metadata, or by filtering out the memory addresses corresponding to the application metadata. Upon eviction from the on-chip cache, modified cache lines that belong to the predictor are propagated only on chip, but not off-chip. In this approach, the reserved physical memory address range is still required.
for addressing purposes, but it does not need to have corresponding space on the physical device. This design option avoids any increase in off-chip memory bandwidth at the expense of potential lower prediction accuracy (i.e., some predictor metadata may be lost). Since the prediction is an advisory mechanism used in the hardware optimization, this approach affects only the effectiveness of the optimization, not the correctness of the application.

3.2 Advantages of Virtualizing Hardware Predictors

Predictor virtualization provides several advantages over traditional techniques of building predictor-based hardware optimizations. In particular:

- PV introduces a flexible and adaptive way of building on-chip predictor metadata tables, with resources being allocated in the memory hierarchy only when the application requires them. Since the chip real estate is a precious resource, this flexibility may lead to more hardware optimizations implemented on chip, which can result in increased application performance.

- In virtualized predictors, only the entries that are produced and accessed are stored. This is an advantage over large dedicated predictor tables that are either sparse or non-uniformly utilized in a conventional predictor-based optimization.

- If the \(PVCache\) hit-rate is high, PV is preferable over a large dedicated predictor because it reduces predictor lookup latency for the common case, as smaller buffers are usually faster to access.

- The size of the \(PVTable\) can be configured at runtime, based on the observed behavior of the application, as long as it does not exceed the space reserved in the physical memory. Assuming OS support, storing the \(PVTable\) in the virtual memory alleviates this restriction.
• With a feedback mechanism in place, optimizations can be toggled off when not effective, without wasting resources for unnecessary application metadata. The predictor metadata allocated in the memory hierarchy, if not utilized, will be silently replaced by application data, as a natural consequence of cache replacement policies. In a traditional design, when the optimization is turned off, the dedicated space occupied by the large, on-chip predictor is completely wasted.

• Conventional hardware predictor designs usually provide per-core predictor tables. These predictor tables are shared across multiple applications that context-switch on the same processor. With no operating system support, the same phenomenon happens for virtualized predictors, i.e., the virtualized predictor tables that are stored in physical memory will be shared among applications running on the same processor. However, with simple operating system support, PV can associate the predictor table with a thread, a process, multiple processes, a core or a set of cores. Since predictor metadata is accessed based on the start address of the in-memory table, it is sufficient to provide a proper start address to enable shared or independent tables. This feature can avoid inter-process predictor table pollution in multi-programmed environments or can promote metadata sharing across cores for multi-threaded applications.

• PV allows applications to influence the prediction process, provided the predictor table is mapped in the process’s virtual memory space. The process can modify predictor entries by simply writing to the appropriate memory locations. The $PVCache$ needs to be memory-coherent for guaranteed delivery of these updates.

• Allocating the predictor table in memory also offers the possibility of making the application metadata semi-persistent, with future invocations of the application benefiting from previously collected metadata.

The sum of these advantages has the potential of changing the way we reason about and build predictor-based hardware optimizations.
3.3 Discussion: To Virtualize or Not To Virtualize

The previous section described several advantages that PV brings to the design of prediction-based hardware optimizations. In this section, we discuss the challenges that PV raises in the context of hardware optimizations and processor design. The discussion is focused on two main aspects: challenges introduced by variable prediction latency and the impact of virtualization on the memory hierarchy.

3.3.1 Prediction Latency

The most important challenge introduced by PV is non-uniform prediction latency. For virtualization to be effective, the optimization engine needs to tolerate this non-uniform latency. The predictor latency depends on where the metadata required by prediction resides in the memory hierarchy (e.g., the dedicated PVCache, lower on-chip caches or main memory). Ideally, all requests would be filled in by the PVCache. However, in practice, some requests are filled in by the L2 cache or lower levels of the memory hierarchy. In general, the closer to the optimization engine the application metadata resides, the more effective PV can become.

Virtualized predictors can exploit locality just as conventional caches do. Conventional caches work due to repetition in application behavior. Since the metadata used in predictors is usually a compact representation of the application behavior, repetition in program behavior leads to repetition in the metadata access stream. This repetition leads to reuse of predictor entries. Assuming traditional cache replacement policies, timely reuse of predictor metadata makes it hot in the caches and, thus, metadata remains stored on-chip, closer to the processor.

The design of the virtualized predictor is essential for exploiting locality. Locality can be enforced by deciding what information is stored in each entry, how the virtual table is indexed, how entries are clustered and brought together into the PVCache. Any a priori knowledge about predictor entry correlation can be exploited by packing entries in the virtualized table such that a single access brings in several correlated entries. Thus, one miss in the PVCache
can be amortized over several consecutive predictions that hit in the dedicated table. Such an example is presented in Chapter 5.

Some hardware predictors, such as the one discussed in Chapter 4, are inherently tolerant to longer access latencies for two reasons: either a single prediction entry generates several speculative actions, and/or the optimization is applied to a much longer latency operation (e.g., main memory accesses). For these type of predictors, virtualization is straightforward: the otherwise dedicated table is conceptually “moved” to main memory, becoming the virtual PVTable. Predictor metadata is brought into the dedicated PVCache, which acts as a buffer for storing metadata before being sent to the optimization engine.

Several hardware optimizations, such as branch prediction, are sensitive to increased prediction latency. In this case, the predictor needs to be designed with virtualization in mind to be able to take advantage of its benefits. For these predictors, exploiting temporal and spatial locality is key to maximizing performance. Predictor entry correlation can also be exploited by anticipating future predictor accesses and prefetching predictor metadata. Chapter 5 discusses the design of a virtualized branch target buffer that employs metadata prefetching.

While virtualization can improve the accuracy of a large class of predictors, virtualization will be less effective for predictors that are intolerant to variable latency and for which locality in the access stream is hard to enforce.

### 3.3.2 Impact on the Memory Hierarchy

PV can install application metadata in the memory hierarchy using regular memory requests, since metadata allocation is performed at cache-line granularity.

The main modification to the memory hierarchy required by PV is the ability to inject memory requests to the lower-level caches, similar to the way higher-level caches communicate with lower-level ones in the memory hierarchy. Current processors already have different components that are not an intrinsic part of the memory hierarchy, but that require the same ability of injecting memory requests. For example, data/instruction prefetchers and page-table
walkers in hardware managed TLBs need to inject requests to the memory hierarchy. For this reason, we believe that the adoption of PV in real hardware is feasible and does not require intrusive modifications to the memory hierarchy.

The memory requests injected by PV compete on the cache access points and cache capacity with the rest of the memory requests in the system. In the use-cases presented in this thesis, we choose to leave the underlying arbitration and cache management policies unchanged for several reasons. First, fewer modifications of the memory hierarchy makes the design more appealing for real-hardware implementation. Second, our experimental results show that for a small number of cores on chip, the interference of PV with the cache hierarchy is limited. However, with an increased number of cores on chip or with an increased number of virtualized predictors per core, we expect some type of prioritization for application demand requests and application data to be required to limit the impact of the memory hierarchy traffic introduced by virtualization.

In general, the footprint of application metadata is smaller than the footprint of application data and instructions. This allows the application metadata to survive in the on-chip caches and cause little cache pollution. To maximize the benefit of application metadata, several optimizations could be composed such that same metadata is used for generating predictions for multiple speculative optimizations. For example, storing branch information as metadata can be used for branch prediction, instruction prefetch or instruction-TLB prefetch.

In the case of chip-multiprocessors, on-chip caches are kept coherent. Unlike application data and instructions, the application metadata used in hardware predictors is informative in nature and it is not required for application correctness. The metadata is normally used for speculative optimizations, thus, keeping the application metadata coherent is not necessary. Depending on the underlying coherence protocol, to avoid extra latency due to coherence messages, coherence can be bypassed for the PV metadata. In this case, the caches need to be aware of the presence of PV data, which can be achieved either by tagging metadata entries in the cache or using metadata address filtering.
3.4 Related Work

The key motivating trends behind the predictor virtualization framework are the following:

- Modern processors are used for a variety of application domains with different characteristics and requirements for hardware resources in general, and hardware predictors in particular.

- The instruction and data footprints of modern applications have increased, and, as a consequence, hardware predictors need to scale such that hardware optimizations remain effective.

- The capacity of on-chip processor caches have been increasing, a trend mainly motivated by large instruction footprints and slow off-chip memories. This trend is projected to continue for the foreseeable future.

These trends have been noted and addressed in several research projects. In this section, we briefly discuss the research ideas that are the closest in spirit and motivation to the predictor virtualization approach.

3.4.1 Reconfigurable Caches

Ranganathan et al. [25] note that not all applications benefit equally from the large on-chip caches present in modern processors. The authors advocate increasing the flexibility of processor caches, such that the on-chip SRAM resources used to implement processor caches can be allocated for different purposes depending on the application’s needs running on the processor. Specifically, the authors introduce a reconfigurable cache design that allows conventional caches to be divided dynamically in different partitions, with one or more partitions being allocated to different processor activities other than conventional caching. As a use case for the reconfigurable cache design, the authors propose using one of the cache partitions as a lookup table for dynamic instruction reuse [4] applied to media processing applications.
While reconfigurable caches add some flexibility to traditional designs for processor caches, there are several shortcomings for the proposed technique, such as: 1) The size and the number of the partitions are fixed at design time, limiting the flexibility of SRAM resource usage; 2) The partitioning is performed at coarse granularity; 3) Maintaining data consistency upon repartitioning is non-trivial; 4) For a realistic implementation, software support is required for re-partitioning decisions. In addition, intrusive modifications of traditional components inside the processor are harder to adopt in real hardware.

In PV, the on-chip caches are transparently used for storing predictor information. Depending on the implementation, the memory hierarchy can be oblivious to the fact that it stores application metadata in addition to the application data and instructions. The allocation of cache capacity for storing predictor metadata is performed on demand, at fine granularity (i.e., cache block), potentially leading to a better utilization of the SRAM resources.

### 3.4.2 Unified Microprocessor Core Storage

Meixner and Sorin [26] propose changes to the processor design such that several processor structures that require SRAM for their implementation use the same unified, dedicated on-chip storage. In the proposed processor design, several components, such as the data/instruction L1 caches, the data/instruction TLBs and the branch predictors, are implemented as two-level tables, in which the second-level table is dynamically allocated in a storage structure unified across all components. The second-level tables correspond to the virtual tables used in PV and they are dynamically allocated into the unified core storage (UCS) following a fixed allocation policy. The UCS is fixed in size at design time and it occupies dedicated on-chip area.

UCS and PV are both motivated by the observation that different applications impose different requirements for processor components to achieve good performance. However, there are two fundamental differences between these research proposals.

First, PV does not dedicate storage for the virtual tables. Instead, PV takes advantage of the existing underutilized space in the memory hierarchy to store predictor metadata. On
the one hand, this approach considerably reduces the dedicated on-chip resources, while, on
the other hand, it needs to overcome the higher access latency of predictor metadata. One
key consequence of PV is the ability to break the limitations imposed by dedicated on-chip
structures.

Second, PV proposes to virtualize only components that store application metadata used by
the processor to implement speculative optimizations. The metadata is not needed for applica-
tion correctness and allows for more flexible policies for storing metadata.

While motivated by similar trends and observations, we believe PV offers a more flexible
approach for building hardware predictors.

### 3.4.3 Memory Hierarchy Used for Predictor Storage

Using the memory hierarchy to store information other than application instructions or data is
not a new concept. Techniques such as LimitLESS directories [27], Virtual Context Architec-
ture [28] and Virtual Transactional Memory [29], spill machine state to the memory hierarchy.
One fundamental difference between these schemes and PV is the nature of the information
spilled to the memory hierarchy. In all the techniques mentioned above, the information spilled
to the memory hierarchy is required for application correctness. In PV the information stored
in the memory hierarchy is advisory in nature and it can be silently discarded without affecting
application correctness.

The need for large predictors to achieve higher accuracy is recognized by other researchers,
specially in the context of data prefetchers. For example, Wenisch et al. [12] reserve space in
physical memory to store miss address streams observed on one processor and use them to
anticipate memory accesses on a different processor in a distributed shared memory system.
Wenisch et al. show that miss streams exhibit temporal address correlation for shared memory
addresses (i.e., groups of shared addresses tend to be accessed together and in similar order) and
temporal reuse (i.e., recent miss address streams tend to recur). Based on these observations,
Wenisch et al. propose Temporal Data Streaming (TDS), a data prefetcher for shared memory.
The TDS engine records sequences of miss addresses observed on a local processor and replays the same sequence on a remote processor when a triggering coherence miss is filled in the local processor. The shared-memory architecture studied assumes an in-memory directory that is augmented with a log of coherent miss streams that is also stored in reserved memory. To achieve its full potential, TDS requires approximate 1.5 MB of miss-address log storage per node.

Lai et al. [11] introduce dead-block correlating prefetchers (DBCP) for application data. The core idea of the DBCP approach is predicting when data blocks become dead in the cache (i.e., last accessed before eviction) and which cache blocks usually replace them. When a data block is predicted dead, a prefetch for the predicted replacement is issued and installed in the place of the dead block. For high accuracy and coverage, the dead block predictor requires large storage structures, in the order of tens of megabytes, which are impractical to store on-chip. Ferdman et al. [13] propose LT-cords, an optimization for the DBCP predictor, in which the prefetching metadata is stored in main memory and streamed in a dedicated structure on-chip. More details about LT-cords can be found in Section 4.4.1. LT-cords makes no attempt to minimize the dedicated on-chip resources and relies on large dedicated on-chip predictors (e.g., 200KB). PV shares some of the goals of LT-cords in that it seeks to enable the implementation of large predictors that are impractical to build due to expensive storage requirements. While LT-cords is an example of how main memory can be used to store metadata, PV is a general framework for emulating large predictors with limited on-chip storage.

The AMD Opteron processor uses the L2 ECC bits to store branch history information upon eviction of an instruction cache line to the L2 cache [30]. This approach can be viewed as a form of predictor virtualization. However, the AMD Opteron technique is not transparent to the memory hierarchy. In contrast with the technique used in the AMD Opteron processor, PV uses the data array of the on-chip caches to store predictor information, which allows for more general and flexible application metadata storage.
3.4.4 Follow-up Research

After its initial proposal [8], the predictor virtualization technique has been used to reduce dedicated resources for implementing hardware optimizations in different research projects. Ferdman et al. [31] uses PV to alleviate the on-chip space requirements for an instruction prefetcher, while preserving the performance improvement of the prefetcher. More details about this work are presented in Section 4.4.2. Al-Otoom et al. [32] uses PV to study the space-performance trade-offs for a branch direction predictor. This work is described in more detail in Section 5.4.2.

3.5 Conclusions

Processing systems have reached a stage where they can not sustain sheer performance by technology scaling alone. Thus, to maximize performance, innovative ideas are required to better utilize the available resources. Predictor virtualization is a technique meant to overcome the limitations of the traditional one-size-fits-all approach for designing predictor-based hardware optimizations. Applying PV to existing predictors reduces the cost of the dedicated on-chip resources, while maintaining the effectiveness of the predictor. More importantly, PV breaks the limitations of conventional designs and allows the implementation of otherwise impractical predictors.

PV exploits the current trend towards large on-chip secondary caches that can sustain considerable amounts of bandwidth. We believe that PV is a timely proposal, taking advantage of the current technological inflection point where the wealth of resources allocated to on-chip caches experience diminishing returns. PV increases the utility of conventional processor caches: in addition to being accelerators for slow off-chip memories, on-chip caches become leverage for effective predictor-based hardware optimizations. This is a novel use of available on-chip transistors that goes beyond traditional caching.

Implementing predictor virtualization techniques in real silicon is tractable. PV does not
require intrusive design modifications. The only important addition to existing designs is the ability of the PV components to communicate with the memory hierarchy. Arbitration is already supported between the L1 data and instruction caches in traditional designs. Adding a new component at the back side of the L1 caches should be a straightforward extension, similar to the addition of an L2 prefetcher that injects requests to the L2 cache.

Due to its advantages and non-intrusive design, we believe that predictor virtualization represents a valid design option for hardware optimizations. PV has the potential of influencing the design and the type of hardware optimizations built in commercial processors.
Chapter 4

Virtualizing A State-of-the-Art Data Prefetcher

Processor speed has increased considerably during the last decades, at higher pace than physical memory speed has. This lead to a serious performance problem in modern systems today, usually referred to as “the memory wall”. With slow memories, the processor spends a lot of the time idle waiting for data or instructions to arrive from memory. Building large on-chip caches is one way of addressing the memory wall. However, as explained earlier, increasing the size of the caches results in diminishing returns. After a certain point, adding more capacity does not improve the cache hit rate, due to limited temporal and spatial locality in the application memory access stream.

The memory wall problem motivated a large body of work dedicated to data and instruction prefetching. Prefetchers aim at learning the memory accessing behavior of applications and anticipating future memory accesses. Thus, prefetchers request the data to be brought from memory in the on-chip caches before the processor demands it.

Despite considerable effort to understand, design and tune data prefetchers [2, 6, 13, 33, 34, 35, 36, 37, 38], only the simplest proposals are implemented in real hardware [39].

Adopting data prefetchers in processors is difficult due to several factors. First, the intrusive
nature of some prefetchers calls for significant changes in the processor design. Second, the more effective techniques often require a considerable amount of dedicated on-chip resources that are too expensive to provide. The research presented in this chapter addresses the latter concern.

The demand for precious on-chip storage can be alleviated by virtualizing the predictor table used in prefetchers. As a proof of concept, we choose to virtualize a state-of-the-art data prefetcher, Spatial Memory Streaming (SMS) [2], that was shown to improve the performance of applications with complex memory access patterns. Using several commercial applications and a combination of trace and execution driven experiments, we demonstrate that the performance of the prefetcher can be preserved through virtualization, while the dedicated resources are reduced by a factor of 69.

This chapter is organized as follows. The SMS data prefetcher [2] is briefly described in Section 4.1. Section 4.2 discusses a virtualized design for SMS that follows the architecture presented earlier in Chapter 3. The experimental analysis for the virtualized SMS prefetcher is detailed in Section 4.3. Section 4.4 outlines the related work and Section 4.5 discusses our conclusions.

### 4.1 Background: Spatial Memory Streaming

Spatial memory streaming (SMS) [2] is a data prefetcher that exhibits high performance for commercial applications, which have notoriously complex memory access patterns that are hard to predict using simple techniques.

SMS infers at runtime spatially-correlated memory access patterns over large regions of memory and uses these patterns to predict future memory accesses. Due to its high accuracy, SMS “streams” the predicted data blocks directly into the level one cache as fast as available bandwidth and resources allow. This section briefly explains how SMS works.

Applications exhibit spatial relationships among the cache blocks they access. At runtime,
SMS records the memory accesses of the application, discovers the spatial relationships among them, and uses these spatial patterns to predict the cache blocks accessed by the application in the future. Once the blocks are predicted, they are brought from main memory directly into the first level data cache.

SMS splits the memory accessed by the application into contiguous regions called spatial regions, as shown in Figure 4.1. A spatial region contains a fixed number of memory blocks. The size of each block is equal to the size of a cache line. At runtime, SMS learns spatial patterns within spatial regions based on the application’s requests for memory. A spatial pattern records which blocks from a region are accessed together within a certain period of time called generation. In other words, a spatial pattern represents the footprint of memory accesses within a spatial region observed during a particular generation.

![Figure 4.1: Spatial memory streaming (SMS)](image)

The first access in a spatial region is called a triggering access and determines the start of a new generation. A spatial region generation ends when any block within the region that has been accessed during the current generation is removed from the cache by replacement or invalidation. The next access to the same spatial region will start a new generation for that region (i.e., the next access represents a new triggering access). To be able to predict future accesses, SMS correlates spatial patterns with the triggering instruction address and the offset within the memory region that the triggering instruction touches. At the beginning of a new
generation, the previous spatial footprint is used to prefetch blocks from the corresponding memory region.

4.1.1 Spatial Memory Streaming Implementation

Spatial patterns are represented as bit-vectors: each block within a region maps to a bit within the bit-vector; if the block is touched within a generation, the bit is set, while all the untouched blocks map to zero bits. One bit-vector records the application access footprint within a memory region during a generation.

To detect spatial patterns, SMS uses two main hardware structures: the active generation table (AGT), and the pattern history table (PHT). The PHT stores the spatial patterns discovered by the AGT and it consumes the bulk of the physical resources needed by SMS.

The AGT records the active spatial patterns that are generated as the application accesses memory. A pattern is active as long as all region blocks that have been accessed since the triggering access are still present in the cache. For each spatial pattern, the AGT stores the program counter (PC) of the triggering memory instruction and the offset within the spatial region of the block that the triggering instruction accesses. At the end of a generation (i.e., the eviction or the invalidation of any block accessed during the generation), the spatial pattern from the AGT is transferred to the PHT and the corresponding AGT entry is freed.

Although the AGT is conceptually a single table, in practice, it is implemented as two tables: the accumulation table and the filter table. The filter table records the spatial region tag, the PC and the block offset for all triggering accesses. Once a spatial region observes an access different than its triggering one, the corresponding entry from the filter table is transferred to the accumulation table where the spatial pattern is being generated. The goal of the filter table is to reduce the pressure on the accumulation table by filtering out all spatial regions that have only one access during a generation (i.e., the triggering access). The accumulation table stores all the active regions that contains at least one block set in their pattern.

Once a spatial pattern is completed, it is transferred to the PHT. Patterns stored in the PHT
are used to predict memory accesses and generate the corresponding data prefetches. Figure 4.2 depicts the structure of the PHT and the prediction process. The PHT table is indexed by a combination of the triggering instruction’s PC and the block offset of the triggering access.

The PHT is accessed at the start of each spatial generation and it predicts the memory blocks accessed within the current generation. If the PHT contains a pattern corresponding to the triggering access, the spatial pattern is used to anticipate the memory blocks that will be accessed during the generation. The region base address and the spatial pattern are used to generate the stream of prefetch addresses. As the experimental evaluation for this prefetcher demonstrated [2], the prefetcher is sufficiently accurate to allow for direct installment of data blocks into the L1 data cache.

Figure 4.3 shows a diagram with the main components within SMS: the detector (AGT) and the predictor (PHT) tables. In terms of dedicated resources, in practice, the detector requires less than one kilobyte of space, while, to achieve its full potential, the predictor table requires a lot more resources, in the range of 60KB. Thus, the virtualization is applied only to the predictor table since it is by far the most resource-hungry among the tables used by the SMS prefetcher. The rest of the prefetcher remains identical to the original design. The next section presents the details of the virtualization of the predictor table.
4.2 Virtualizing Spatial Memory Streaming

To provide accurate predictions, the PHT needs to store a high number of spatial patterns. The original SMS study [2] shows that, for the prefetcher to reach its full potential, the PHT table needs to have 16K entries. The study uses a 16-way set-associative PHT (i.e., 1K sets). Considering spatial regions of 32 blocks and using 16 bits from the PC for indexing the table, the space required for the PHT is 86KB. The data array contains 16K entries with 4 bytes per entry (i.e., one bit for each of the 32 blocks in the spatial region) for a total of 64KB. The index for the PHT is obtained by concatenating 16 bits from the PC with 5 bits that represent the offset within the 32-block spatial region. Thus, the index is 21 bits long. Ten out of the 21 bits are used to index the PHT table, while the rest of 11 bits are stored as tags in the PHT. Thus, the tag array portion of the PHT is 22KB. The total storage for the PHT in the original design is 86KB. Virtualization can preserve the performance improvements of the data prefetcher with less than a kilobyte worth of dedicated on-chip space for the PHT.

The virtualized SMS is a straightforward implementation of the architecture discussed in Chapter 3. In the virtualized design, the PHT is stored in main memory as the PVTable is in the general framework. The PVProxy buffers the current entries of the PHT, interfaces with the AGT and also ensures proper delivery of the PHT predictions to the prefetching engine. The next sections describe the implementation of the two main components of the virtualized
prefetcher: the \textit{PVTable} and \textit{PVProxy}.

\subsection{PVTable}

In the proposed implementation, all entries within one PHT set (both tags and data) are packed into a contiguous block of memory equal in size to the L1 cache block (64 bytes), such that an entire predictor table set is retrieved from the lower cache hierarchy levels with only one memory request. The associativity of the PHT table is adjusted such that one set fits in a cache block. Adjusting the associativity of the PHT does not reduce the prefetching performance potential, as the results in Section 4.3.3 show.

Figure 4.4 depicts how predictor entries within a set are packed into a 64 byte contiguous memory block. The PHT table chosen for virtualization contains 1K sets, each set with 11 ways. The index used for the table is composed by concatenating 16 bits from the triggering access PC with five bits representing the block offset of the access within a spatial region of 32 blocks. Since the table contains 1K sets, 10 bits from the 21-bit table index are used to retrieve the set from the PHT. The remaining 11 bits are stored as a tag in the table. A spatial pattern needs 32 bits to record the footprint for the 32 blocks in a spatial region. Thus, there are 43 bits to be stored per predictor entry. After packing 11 entries of 43 bits each in a 64 byte block, there are some trailing unused bits. These bits could be used for LRU access information, or could be reserved for further optimizations. These bits are unused in the current implementation.

\begin{center}
\begin{figure}
\begin{tabular}{cccc}
0 & 11 & 43 & 54 \\
\hline
tag & pattern & tag & pattern \\
\hline
\end{tabular}
\end{figure}
\end{center}

Figure 4.4: metadata packing: one entire PHT set is packed in one L2 cache block
4.2.2 PVProxy

The PVProxy acts as a mediator between the in-memory PVTable and the rest of the components of the prefetcher. All requests that would normally go to the PHT table in the original design are intercepted by the PVProxy. The PVProxy contains a dedicated cache (PVCache) in which a few PVTable sets are stored at a time (eight sets in the configuration studied in this work). The table used for the PVCache is fully associative and each entry has an identifier tag that indicates the PVTable set that is stored in that entry. Upon receiving a request, the PVProxy checks its PVCache. On a hit, the request is satisfied as in the non-virtualized design. On a miss, the PVProxy sends a memory request to the L2 cache for the corresponding PHT set. To handle memory requests, the PVProxy contains several MSHRs. In addition, to track pattern updates, each PVCache entry is tagged with a dirty bit which is set to one once a pattern is modified. To ensure proper delivery to the cache for the dirty blocks, the PVProxy also contains several write-back buffers. The components within the PVProxy are depicted in Figure 4.5.

Figure 4.5: PVProxy components

Figure 4.6 shows the generation of the address for the memory request used to retrieve the application metadata. The PVProxy receives an index that uniquely identifies the entry in the PHT. This index is used to determine the PHT set to be fetched from the memory hierarchy. To obtain the memory address, the set index is padded with six zeros (each set is stored in a 64-byte contiguous memory block) and added to the start address of the PVTable. A memory
request with the generated address is sent to the cache hierarchy to retrieve the corresponding set from the virtualized PHT.

![Diagram showing address generation for memory requests used to retrieve entries from the virtual PHT](image)

Figure 4.6: Address generation for memory requests used to retrieve entries from the virtual PHT

Once the data is brought in the *PVCache*, the request is completed as in the original design. If the predictor set contains a pattern corresponding to the request that triggered the retrieval of the PHT set, the pattern is used to generate a stream of prefetches similarly to the non-virtualized design.

### 4.3 Experimental Analysis

This section quantitatively demonstrates that SMS benefits from predictor virtualization. Section 4.3.1 details the simulation methodology. The benchmarks used in this study are briefly described in Section 4.3.2.

Section 4.3.3 starts by analyzing how the prefetching potential is influenced by the predictor table size, concluding that large PHTs outperform their small counterparts by a great margin. This result motivates the virtualization of a large PHT. The rest of the results presented in Section 4.3.3 show that the virtualized SMS design offers similar performance compared to the original design, while considerably reducing the dedicated on-chip resources. The impact of virtualization on the memory hierarchy is also analyzed.
4.3.1 Methodology

In our experiments, we used the Flexus simulation infrastructure [40]. Flexus is built on top of Simics, a commercially-available full-system simulation tool, which provides the ability to functionally run unmodified operating systems and commercial applications [41]. Flexus leverages the micro-architectural interface provided by Simics to enable both functional and cycle-accurate timing simulation of uniprocessor and multiprocessor architectures. Flexus provides a modular simulation infrastructure: a simulator is composed of several interconnected modules. In general, each module corresponds to a single hardware structure (e.g., a branch predictor or a cache). Flexus is an open-source simulation framework, which enables the modification of existing modules, the integration of newly developed components in existing simulators or the development of new simulators using both existing and new modules.

The system simulated in our experiments is a four-core chip multiprocessor with a shared, unified L2 cache, based on the Piranha cache design [19]. Table 4.1 shows the processor configuration used in most experiments. We explicitly mention in the text whenever one of these parameters is modified or varied.

<table>
<thead>
<tr>
<th>Branch Predictor</th>
<th>Fetch Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>8K GShare, 16K bi-modal, 16K selector 2 branches per cycle</td>
<td>Up to 8 instructions per cycle 64-entry fetch buffer</td>
</tr>
<tr>
<td>ISA &amp; Pipeline</td>
<td>Scheduler</td>
</tr>
<tr>
<td>UltraSPARC III ISA, 4GHz 8 stage pipeline, out-of-order execution</td>
<td>256-entry/64-entry LSQ</td>
</tr>
<tr>
<td>Main Memory</td>
<td>Issue/Decode/Commit</td>
</tr>
<tr>
<td>3 GB, 400 cycles</td>
<td>8 instructions/cycle</td>
</tr>
<tr>
<td>L1D/L1I</td>
<td>UL2</td>
</tr>
<tr>
<td>64KB 4-way set-associative 64B blocks, LRU replacement 2 cycle latency</td>
<td>8MB, 16-way set-associative 8 banks, 64B blocks, LRU replacement 6/12 cycle tag/data latency</td>
</tr>
</tbody>
</table>

Table 4.1: Baseline system configuration

All cores employ a next-line instruction prefetcher. The baseline configuration does not contain any form of data prefetching. This allows us to isolate the performance improvements
obtained by the original SMS prefetcher and its virtualized design.

Each core has its own SMS prefetcher that inserts prefetched data directly in the corresponding L1 cache. The SMS prefetcher was simulated using an AGT with 64 entries in the accumulation table and 32 entries in the filter table. Spatial regions contain 32 memory blocks. These parameters are the tuned values from the original SMS study [2]. We verified that these values are still the best choice for the chip-multiprocessor architecture we simulate. The geometry of the predictor table (PHT) is varied throughout the experiments as specified in the text. For the virtualized configurations, each core has its own PVProxy and PVTable. There is no sharing between the predictor information collected on one core versus the other to eliminate any potential benefit that virtualization could show on top of the original design. Studying the benefits of virtualization due to information sharing across cores is deferred for future work.

The behavior of the virtualized SMS design is analyzed through a combination of functional and cycle-accurate simulations. When measuring events for which detailed processor simulation is not necessary (e.g., number of cache misses, number of cache requests), we use functional simulation of two billion cycles from a steady point of each benchmark. The first billion cycles is used as a warm-up period and the measurements presented are for the second billion cycles. Two workloads, TPC-H queries 2 and 17, complete in less than two billion cycles; thus, for these queries, we take measurements for 209 million and 259 million cycles, respectively, after the warm-up period of one billion cycles.

The timing performance results (e.g., speedup) use the Flexus cycle-accurate, full-system simulator with the SMARTS sampling methodology [42]. This sampling methodology allows to cover a longer portion of the benchmark in a reasonable simulation time. Figure 4.7 depicts the main concept behind the SMARTS sampling methodology. Detailed timing simulations are performed for several samples across the execution of the benchmark. Each sample starts with a checkpoint of the benchmark that contains on top of the architectural state also micro-architectural state for major processor components (e.g., cache hierarchies, branch predictors, prefetch tables). These micro-architectural components are warmed-up using functional simu-
lotion throughout the execution of the benchmark.

![Diagram showing the sampling methodology](image)

Figure 4.7: Sampling methodology

Each sample measurement consists of 50K cycles of detailed timing simulation, which are preceded by 100K cycles of detailed warmup. The 100K cycles warmup is performed using timing simulation and it is meant to warm up smaller processor structures, such as queues and buffers. The reported results show error bars for a 95% confidence interval. We used matched-pair sampling [43] to measure performance variation. Performance is measured as the aggregate number of user instructions committed each cycle (i.e., committed user instructions summed over the four processor cores divided by the number of total elapsed cycles). This aggregate measures total application throughput [12].

### 4.3.2 Benchmarks

Table 4.2 describes the workloads used in the experimental analysis. The workloads fall in one of the three categories: online transaction processing (OLTP), decision support systems (DSS) and web servers, as follows:

- The TPC-C v.3.0 online transaction processing workload running on both IBM DB2 v8 ESE and Oracle 10g Enterprise Database Server
• Four queries from the TPC-H DSS workload running on IBM DB2 v8 ESE (Queries 1, 2, 16 and 17)

• The SPECweb99 benchmark running with Apache HTTP Server v2.0 and Zeus Web Server v4.3, respectively. The web servers were driven with separate simulated client systems; client activity is not included in the reported results.

<table>
<thead>
<tr>
<th></th>
<th>Online Transaction Processing (TPC-C)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Oracle</td>
</tr>
<tr>
<td></td>
<td>100 warehouses (10GB), 16 clients, 1.4 GB SGA</td>
</tr>
<tr>
<td></td>
<td>DB2</td>
</tr>
<tr>
<td></td>
<td>100 warehouses (10GB), 64 clients, 450 MB buffer pool</td>
</tr>
<tr>
<td></td>
<td><strong>Decision Support (TPC-H on DB2)</strong></td>
</tr>
<tr>
<td>Query 1</td>
<td>Scan-dominated, 450 MB buffer pool</td>
</tr>
<tr>
<td>Query 2</td>
<td>Join-dominated, 450 MB buffer pool</td>
</tr>
<tr>
<td>Query 16</td>
<td>Join-dominated, 450 MB buffer pool</td>
</tr>
<tr>
<td>Query 17</td>
<td>Balanced scan-join, 450 MB buffer pool</td>
</tr>
<tr>
<td></td>
<td><strong>Web Server</strong></td>
</tr>
<tr>
<td>Apache</td>
<td>16K connections, FastCGI, worker threading model</td>
</tr>
<tr>
<td>Zeus</td>
<td>16K connections, FastCGI</td>
</tr>
</tbody>
</table>

Table 4.2: Benchmarks used in the analytical evaluation

4.3.3 Experimental Results

This section presents the experimental results that quantitatively analyze the virtualized prefetcher design compared to the non-virtualized counterpart. First, we discuss how the performance potential of the prefetcher varies with predictor table size. Second, we demonstrate that SMS can benefit from virtualization by reducing the dedicated on-chip resources considerably, while preserving the performance improvement. Third, the interactions of the virtualization with the underlying memory systems is analyzed in terms of increased memory traffic and sensitivity to L2 cache size and latency.
Prefetching Potential versus Predictor Table Size

This section analyzes the variation in SMS performance potential as a function of predictor table size. The results show that, to exploit the full potential of the prefetcher, the predictor table has to be sufficiently large to store a considerable number of spatial patterns. This result motivates applying predictor virtualization to the PHT.

Figure 4.8 shows the performance potential of the prefetcher for different predictor table configurations. The performance potential is plotted as the percentage of L1 read misses that are eliminated by the prefetcher (covered misses in the figure). The misses that are not eliminated by SMS are presented in the graph as uncovered. The overpredictions represent the prefetched data blocks that are evicted or invalidated before they are used by the processor.

![Figure 4.8: SMS performance potential as a function of the PHT size.](image)

The first bar in the graph corresponds to unbounded predictor tables. In this case, the predictor table stores all the spatial patterns discovered throughout the execution of the application. The second configuration (1K-16a) is the configuration that was found to perform the best in the original SMS study for a 16-node, shared-memory architecture [2]. In this configuration, the predictor is organized as a 16-way set-associative table with 1K sets for a total of 16K entries.

Packing an entire predictor table set in a contiguous memory block equal in size to the cache
block allows retrieving an entry from the table with only one memory request, thus reducing the pressure on the memory hierarchy. Reducing the table associativity from 16 to 11 ways (1K-11a) does not decrease the performance potential noticeably, while it conveniently allows to pack an entire predictor table set in a cache block (for a 64-byte cache size).

Figure 4.8 also shows results for smaller 11-way set-associative tables down to eight sets (8-11a) to visualize the impact on prefetching potential when limiting the storage available to the predictor table.

The behavior of the prefetcher varies across applications when the predictor size is decreased. For example, for TPCC running on Oracle, coverage drops from 44% with 1K sets down to less than 4% with eight sets. Other workloads, such as some of the TPC-H queries are less sensitive to the predictor size. For example, for Query 1, the coverage decreases from 73% with the infinite tables to 62% with a 16-set table. The speedup results presented later in this section demonstrate that even this relatively small decrease in coverage can lead to important performance loss.

Figure 4.9 presents several intermediate table sizes (i.e., from 1K entries to eight entries) for three representative workloads. While the curve describing the decrease in covered misses is different for each benchmark, the results show that reducing predictor sizes leads to a significant drop in performance potential for most workloads. Thus, naively reducing the size of the predictor table is not a viable solution for decreasing the on-chip resources dedicated to the predictor.

Table 4.3 lists the storage requirements for each configuration. In its original proposal, the predictor needs more than 80 kilobytes of on-chip space to achieve its potential (1K-16a configuration). Each core uses its predictor, hence the space dedicated on chip is multiplied by the numbers of cores on chip. While the small predictors require approximately one kilobyte of space, their performance potential is much lower for most workloads. Predictor virtualization aims at achieving the best area versus performance potential trade-off.

We choose to virtualize the table with 1K sets and associativity of 11, since its performance
CHAPTER 4. VIRTUALIZING A STATE-OF-THE-ART DATA PREFETCHER

Figure 4.9: SMS performance potential - representative behavior

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Tags</th>
<th>Patterns</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K-16a</td>
<td>22KB</td>
<td>64KB</td>
<td>86KB</td>
</tr>
<tr>
<td>1K-11a</td>
<td>15.125KB</td>
<td>44KB</td>
<td>59.125KB</td>
</tr>
<tr>
<td>16-11a</td>
<td>374B</td>
<td>880B</td>
<td>1.225KB</td>
</tr>
<tr>
<td>8-11a</td>
<td>198B</td>
<td>440B</td>
<td>0.623KB</td>
</tr>
</tbody>
</table>

Table 4.3: Storage for different predictor table configurations

is close to that of the infinite table. The reduced number of ways allows for packing an entire PHT set within a memory block equal in size to the cache line used in our configuration (i.e., 64 bytes). The difference in coverage between the 1K-11a configuration and an unbounded table is at most 3% across all applications. For the virtualized configuration, each PVTable has 1K sets, and each set is mapped to a 64 byte block, for a total of 64KB main memory storage per core. In our experiments, we present results for a four-core chip multi-processor (CMP), which corresponds to a total of 256KB of reserved physical memory space for predictor virtualization.

Performance of the Virtualized Predictor

In the previous section, we analyzed the potential of the original SMS prefetcher while varying the size of the predictor table. Across the applications studied, reducing the size of the predictor
table lowers the number of misses covered by the prefetcher. In this section, we quantify the application performance loss when limiting the storage used by the predictor table. In addition, we demonstrate that the performance loss can be recovered by using a virtualized SMS design. In other words, we show that the virtualized SMS design emulates the performance of the original design, while drastically reducing the hardware storage.

Figure 4.10 shows the percentage speedup\(^1\) for four configurations. The first three bars represent the percentage speedup increase of the non-virtualized prefetcher with different predictor table sizes: 1K sets, 16 sets and eight sets, respectively; all tables are 11-way set-associative. The last bar in the graph shows the speedup for the virtualized predictor with just eight dedicated on-chip sets. The baseline uses no prefetching.

Overall, the virtualized prefetcher achieves similar performance to the non-virtualized one. On average, the original prefetcher improves performance by 19%, while the virtualized improves performance by 18%. In the worst case, for TPCC running on Oracle, the improvement of the non-virtualized prefetcher is 6.7%, while the virtualized prefetcher achieves 4.2% performance improvement. For the same benchmark, the small, non-virtualized prefetchers improve performance only by 2% over no prefetching.

On average, the smaller dedicated prefetchers achieve only half of the performance of the 1K-set prefetcher. For Apache, the small predictor tables are entirely inefficient, leading to no speedup compared to the baseline, while the virtualized predictor achieves similar performance to the non-virtualized case.

For Apache, some of the sample measurements show higher performance with virtualization compared to the original prefetcher. The virtualized prefetcher incurs a higher latency to figure out what to prefetch, which leads to installing the prefetched lines in the L1 cache with a delay compared to the original prefetcher. In some corner cases, this delay slightly reduces the cache pollution produced by the prefetcher. Some cache lines that would have been replaced

\(^1\)Percentage speedup is computed as the percentage increase in instructions per cycle (IPC) between the prefetching scheme and the baseline with no prefetching. IPC is measured using cycle-accurate simulations as explained in Section 4.3.1.
by the prefetched blocks survive longer in the cache in the virtualized design. This corner case happens only rarely and we do not expect higher performance for virtualized prefetchers.

The performance results of the virtualized design make it a viable option for implementing prefetchers with large predictor state. One way of interpreting these results is that, if at design time, the on-chip storage available for implementing a data prefetcher is restricted to one kilobyte of dedicated space, this budget can be used to implement a traditional SMS design or a virtualized design. For some benchmarks, the traditional design obtains reasonable performance as it is the case for TPCH Query 1 in our experiments, while for other applications, such as Apache, no performance improvement is achieved. In the virtualized design, the one kilobyte of dedicated space is sufficient to achieve most of the potential of the SMS prefetcher.

![Figure 4.10: Performance of the virtualized SMS prefetcher](image)

**Impact of the Predictor Virtualization on the Memory System**

The performance of the virtualized prefetcher depends on two factors: 1) the interaction with the rest of the memory system, and 2) the timeliness of the prefetcher after virtualization. In this section, we study the impact of virtualization on the number of the L2 requests and on the off-chip memory traffic. The timeliness of the prefetcher is reflected in the performance results presented in the previous section.

Figure 4.11 reports the increase in L2 memory requests due to virtualization as a function of
the number of PVCache sets. For an eight set PVCache, the increase in memory requests varies between 25% and 44%, with an average of 33%. Increasing to 16 the number of dedicated sets for the PVCache leads to similar results. This implies that before an entry gets evicted from the PVCache it is either used only once or exhibits very short term temporal locality. Only for Query 16, increasing the number of dedicated sets to 32 leads to a decrease in the additional L2 traffic of more than 5%.

![Figure 4.11: Percentage increase in L2 memory requests due to virtualization](image)

Predictor virtualization trades off dedicated on-chip space for additional memory traffic. While the increase in L2 memory requests is significant, the timing results presented in the previous section demonstrate that it has a minimal impact on performance. In this set of experiments, no prioritization was performed for virtualization requests and regular requests. Such prioritization may be necessary for CMPs with higher number of cores on chip to limit the interference of the requests for application metadata on the demand requests for application data and instructions.

Figure 4.12 shows the impact of virtualization on the off-chip bandwidth expressed as the percentage increase in the number of L2 cache misses and write-backs. The results are presented for virtualized predictors with eight and 16 sets in the dedicated PVCache, respectively.
compared to the non-virtualized SMS prefetcher. The results for 32 dedicated sets in the \textit{PV-Cache} are similar.

For five out of the eight simulated workloads, the increase in number of misses is less than 1\%. For the rest, the increase in L2 cache misses is less than 3\%. The percentage increase in the number of L2 cache misses does not change when the number of dedicated entries is increased to 16 or even 32 entries. The L2 write-backs experience minimal increases as well, with a maximum of 3.2\% increase for Zeus. The increase in off-chip bandwidth due to virtualization is 3.3\% on average, with a maximum of 6.5\% for Zeus.

![Figure 4.12: Percentage increase in off-chip bandwidth due to virtualization](image)

Overall, there is little benefit from increasing the number of dedicated on-chip resources from eight sets to 16 or even 32, therefore, we decided to use only eight sets for the \textit{PVCache} in our final design.

Figure 4.13 further splits the increase of L2 misses and write-backs into application data and predictor metadata for a configuration with eight dedicated entries in the \textit{PVCache}.

These results demonstrate two important points. First, the predictor entries that are naturally cached in the L2 do not lead to cache pollution. L2 application data misses increase by
less than 2.5% for all studied benchmarks, with an overall average increase of 1%. The L2 cache space is better used to store predictor information rather than exclusively storing application data. Second, most of the memory requests generated by the PVProxy are satisfied by the L2 cache. Across all applications, more than 98% of the PVProxy requests are filled in by the L2 cache. The predictor entries are sufficiently hot to be kept in the L2 cache. This considerably reduces the latency of the virtualized predictor compared to the case where its entries would be delivered from main memory most of the time.

![Graph](image.png)

Figure 4.13: Percentage increase in off-chip memory traffic split between application and predictor data (PV-8 configuration)

The low off-chip traffic for predictor information also indicates that not using the main memory as a back-up storage will lead to limited loss in performance. Depending on the application access pattern, the spatial footprints can be recreated and reinstalled in the on-chip cache hierarchy.

**Sensitivity to the L2 Cache Size and Latency**

In this section we analyze the sensitivity of the virtualized prefetcher to the L2 cache capacity and latency.

Predictor virtualization is motivated by diminishing returns in application performance im-
improvement with increased cache sizes. In this context, we expect that the space occupied by the PV data becomes less of a concern for larger caches. In this section, we analyze the original prefetcher and the virtualized design in functional simulation, while varying the L2 cache size from 512KB to 2MB per core, for a total of 2MB to 8MB. Figure 4.14 shows the increase in off-chip bandwidth split between L2 misses and L2 write backs. As expected, PV interferes less with the application as the L2 cache capacity increases. The interference is minimal in the case of an L2 cache size of 2MB per core, for a total of 8MB.

The size of the virtual table can be varied such that the interference on the cache capacity is minimal. If cache pollution becomes a concern, well-known solutions can be employed, such as inserting the metadata in a lower position of the LRU stack or allocating the virtual table in an address space such that the metadata is limited to a portion of the on-chip cache (technique known as page coloring [44]). The analysis of such techniques is deferred for future work.

![Figure 4.14: Off-chip bandwidth increase for different L2 cache sizes split into L2 misses and write backs](image)

We also verify that PV remains effective even with a longer L2 latency. For this experiment, we measured performance with the L2 tag and data latencies increased to eight and 16 cycles respectively (previous runs assumed latencies of six and 12 cycles), for a total of 24 cycles latency. The first bar in Figure 4.15 represents the speedup obtained by the original data
prefetcher, while the second bar shows the speedup of the virtualized design. On average, the difference is less than 1.5%. These results are obtained with detailed timing simulation.

![Performance of the virtualized predictor with increased L2 latency](image)

**Figure 4.15: Performance of the virtualized predictor with increased L2 latency**

**PVProxy Space Requirements**

The components of the PVProxy are all dedicated on chip and occupy the following resources:

- the **PVCache** - 8 sets, each set contains 11 entries of 32 bits of data and 11 bits of tags, for a total of 473 bytes

- PHT indexes that uniquely identify the sets within the **PVTable** that are currently stored in the **PVCache**; each index has 10 bits (to index a table of 1K entries), for a total of 10 bytes

- dirty bits that record whether a set in the **PVCache** needs to be stored to the caches or not; one bit for each of the 8 sets, for a total of one byte

- 16 MSHRs that hold the PHT index (10 bits) and the start address of the region to be prefetched (26 bits; the address is cache aligned) - 72 bytes.

- a set of 4 write-back buffers, each 64 bytes long for a total of 256 bytes
• a 16-entry pattern buffer that stores the patterns while the corresponding sets are brought from the lower cache, for a total of 64 bytes

The total space required by the PVProxy adds up to 876 bytes per core. The corresponding non-virtualized prefetcher requires 59.125KB of on-chip storage per core. The virtualized design reduces the hardware cost for the predictor table by a factor of more than 69, while preserving most of the performance.

4.4 Related Work

High off-chip memory latencies have been one of the major bottlenecks in achieving higher performance in modern processors. As a consequence, a lot of research has been dedicated to designing, evaluating and tuning data and instruction prefetchers [2, 6, 11, 12, 13, 31, 33, 34, 35, 36, 37, 38]. In this section we discuss only the few approaches that focus on reducing the on-chip dedicated resources for implementing instruction and data prefetchers.

4.4.1 LT-cords: Last-Touch Correlated Data Streaming

Lai et al. [11] introduced a dead-block predictor that identifies the last access to a cache block before the cache block is evicted. Dead blocks occupy precious space in the cache without any benefit since dead block are evicted before they get a chance of being re-accessed. A dead-block predictor, when coupled with a prediction for which block will replace the dead-block, can be used as an effective data prefetcher. Based on this observation, the authors proposed dead-block correlating prefetchers (DBCP). In DBCP, when a block is foreseen dead, it is invalidated and a prefetch for its predicted replacement is sent to the lower memory hierarchy. The prefetched block replaces the dead-block in the cache.

The prediction metadata used in DBCP is based on traces of instructions that touch memory blocks. Thus, the metadata scales with the number of blocks and instructions touched by the application. DBCP obtains high accuracy only when its predictor is large, on the order of
several of megabytes in the original study, which is an impractical size for dedicated on-chip resources. While the metadata can be stored in off-chip main memory, the prefetcher suffers from less-timely predictions and shows lower performance improvements.

Ferdman et al. [13] show that, for more complex applications, the dead-block predictor needs in the order of tens of megabyte of storage to achieve its full potential. In addition, the authors show that the metadata used in the DBCP prediction exhibits temporal locality, which can be exploited to implement a practical DBCP. Thus, the authors propose LT-cords [13], an optimization for the DBCP predictor, in which the prefetching metadata is stored in main memory and streamed in a dedicated structure on-chip.

In the proposed design, LT-cords still requires large dedicated on-chip buffers (e.g., approximately 200KB for the applications studied). LT-cords shares some of the goals of predictor virtualization in that it seeks to enable the implementation of large predictors that are impractical to build due to expensive storage requirements. While LT-cords is an example of how main memory can be used to store metadata, PV is a general framework for emulating large predictors with limited on-chip storage. It may be possible to further apply virtualization to the dedicated buffers in LT-cords to either reduce the dedicated space or increase the tolerance to memory access reordering.

### 4.4.2 TIFS: Temporal Instruction Fetch Streaming

Database workloads are notorious for their large instruction footprints with complex access patterns which lead to high instruction cache misses. Ferdman et al. [31] show that instruction misses are temporally correlated: applications exhibit long sequences of instruction misses that tend to repeat in the same order. Based on this observation, the authors propose temporal instruction fetch streaming (TIFS), an instruction prefetcher that records and replays recurring L1 instruction miss sequences.

One of the main components in the hardware implementation of TIFS is the Instruction Miss Log (IML), a structure that records the misses observed by the L1 instruction cache.
The IML is used to anticipate future instruction misses and fetch them in advance, before the demand access. Provided sufficient repetition exists in the stream of instruction misses, using the IML as a predictor, the prefetcher is able to create the illusion of a larger instruction cache.

The authors show that, in order to achieve its full potential, the IML needs to store around 8K instruction block addresses per core. For a system with four cores, this leads to a storage requirement of 156KB. The authors study the performance of the instruction prefetcher with a dedicated IML and with a virtualized IML that follows the architecture described in Chapter 3. For the virtualized design, the IML is stored in the L2 cache. For all the benchmarks studied with one exception the virtualized design performs similarly to the non-virtualized prefetcher. For one of the benchmarks, the performance of the virtualized design is lower due to a slight increase in L2 bank contention.

### 4.5 Concluding Remarks

In this chapter, we explored applying virtualization to a state-of-the-art data prefetcher, Spatial Memory Streaming. SMS was proved to improve the performance of commercial applications which are well-known to exhibit complex memory access patterns. To achieve its full potential, SMS requires large storage structures to record memory access footprints. SMS is an excellent candidate for virtualization. First, the prefetcher is tolerant to a higher prediction latency since it is optimizing for long-latency memory accesses. Second, SMS is an aggressive prefetcher that uses each entry in its predictor table to generate multiple predictions for future memory accesses (i.e., one spatial footprint can generate prefetches for multiple blocks within the corresponding memory region).

In the virtualized design, a small buffer stores few sets from the main predictor table used in SMS. The rest of the predictor table is allocated on demand in the on-chip cache hierarchy. Using several commercial applications and full-system, cycle-accurate simulation experiments, we showed that the performance of the prefetcher can be preserved through virtualization,
while the dedicated resources to implement the main predictor table are scaled down from more than 60KB to less than one kilobyte of storage. All this is obtained without any modifications to the underlying cache hierarchy, except for the ability to insert virtualization requests in the cache hierarchy, starting at the L2 cache level.

Virtualization comes with a cost: increased activity in the on-chip caches. The number of L2 requests observed increases by 33% on average. Although this increase is non-trivial, for the four-core CMP system we used in our experiments, there was no need for special arbitration for on-chip cache requests. For systems with a higher number of cores on chip, cache requests resulting from demand memory accesses can be prioritized over requests from the virtualization engine to avoid potentially negative effects of congestion on application performance.

The footprint of the application metadata allocated in the on-chip caches and main memory is small compared to the application footprint. Thus, there is only a marginal increase in cache pollution and off-chip bandwidth as a result of virtualization.

The results presented in this study, reinforced by independent studies for virtualizing instruction prefetchers [31], lead us to believe that virtualization is a “plug-and-play” technique for data and instruction prefetchers which employ large storage structures for recording memory access patterns. A straightforward application of virtualization leads to similar performance with reduced on-chip resource requirements.
Chapter 5

Virtualizing Branch Target Buffers

Branches are special instructions that allow programs to divert from the sequential flow of execution to a different location in the program. Branch prediction is a speculative optimization omnipresent in current processors. Branch prediction increases the concurrency of instruction processing within the processor pipeline by anticipating the diversion of the instruction flow introduced by a branch. Without branch prediction, the processor would have to first fetch the branch, decide whether it is taken and if it is, calculate its target, and redirect the instruction fetch to the new address. All these operations would stall instruction fetching until the address of the next instruction is known, which would lead to bubbles in the processor pipeline and, in consequence, to lower application performance. Branch prediction increases application performance by allowing the processor to fetch ahead in the instruction stream while previous instructions are still being processed, without waiting for branch resolution, as shown in Figure 5.1.

Branch prediction involves the prediction of two orthogonal components of the branch: 1) the direction of the branch and 2) the target of the branch. The direction of the branch refers to the outcome of the branching condition, i.e., whether the branch results in a jump to a new location (taken branch) or if the branch is non-taken and the stream of instructions follows the sequential flow. For taken branches, the target of the branch specifies what instruction has to be
CHAPTER 5. VIRTUALIZING BRANCH TARGET BUFFERS

Figure 5.1: Instruction execution without (a) and with (b) branch prediction

executed next. There has been a lot of research on both direction and target branch prediction. The focus of this chapter is the virtualization of target branch prediction.

The main structure used in branch target prediction is a Branch Target Buffer (BTB) [15]. The BTB acts as a last-value predictor by recording the last target seen by a particular branch. In its most straightforward implementation, the BTB is a table that associates the address of a branch with its target address after the branch is taken for the first time. Upon subsequently encountering the same branch, the processor predicts its target address by looking it up in the BTB and retrieving the recorded target. If the target address is the same as the previous time the branch was encountered, which is the case for most branches \(^1\), the BTB prediction is correct.

To improve application performance, branch prediction needs to be accurate and timely. On one hand, ideally, the BTB would be large enough to capture the working set of taken branches to provide high prediction accuracy. On the other hand, a larger BTB has a higher access latency, which can lead to longer prediction latencies, offsetting the benefits of higher accuracy in terms of application performance. In addition, a larger BTB implies a higher hardware cost. Depending on the application mix, it is challenging to find a "one-size-fits-all" design point for BTBs that strikes a balance among BTB accuracy, latency, and dedicated resources.

\(^1\)Some branches, such as return from procedure calls or indirect branches may have several possible targets. For this type of branches, the BTB is not as efficient. Usually, processors include special structures to predict indirect branches and returns.
For example, commercial applications, such as databases and web servers, exhibit large instruction footprints and challenge conventional BTB designs [7, 17, 18, 19, 45]. Hierarchical BTBs [46], pipelining, and overriding predictors [47] can partially compensate for the longer access latency to larger BTBs. However, the challenge of balancing the area cost of a larger BTB remains.

In this chapter, we describe how predictor virtualization can be applied to latency-critical predictors such as the BTBs. The goal of this research is to design a BTB that achieves much of the accuracy of larger conventional BTBs without incurring their latency and area penalties. This chapter presents Phantom-BTB, a novel virtualized hierarchical BTB design [9].

As with other virtualized predictors, Phantom-BTB taps into transiently under-utilized cache capacity for storing branch information to accommodate application demands for a larger BTB. Phantom-BTB complements a reasonably tuned, first-level, conventional BTB with a second-level, large virtual table. While the first-level BTB uses dedicated storage and operates like a conventional BTB would, there is no dedicated storage for the second-level table (hence the name “virtual table”). The virtual table entries are transparently allocated and stored on demand, at cache line granularity, in the on-chip memory hierarchy starting at the L2 cache (to avoid contention at the performance-critical L1 caches). The virtual table entries compete for cache capacity with application data and instructions.

The key challenge for Phantom-BTB compared to hierarchical predictor designs [46] is compensating for the much higher latency for accessing entries within the virtual table. For this purpose, Phantom-BTB deviates from conventional hierarchical BTB designs where the first level operates as a cache for a second level, larger predictor table. Instead, the virtual table is organized such that it facilitates timely prefetching of BTB entries into the first-level BTB. Phantom-BTB relies on repetition and temporal correlation in the BTB miss stream to store and prefetch temporally correlated branches from the virtual table, creating the illusion of a much larger dedicated BTB.

Before presenting the virtualized BTB design, we briefly discuss conventional BTBs in
Section 5.1. We further motivate and present the Phantom-BTB design in Section 5.2. The experimental evaluation of the virtualized BTB design is discussed in Section 5.3. Section 5.4 overviews the various pieces of work that address challenges in the design and implementation of branch predictors. Section 5.5 concludes by summarizing our findings.

5.1 Background: Branch Target Buffers

As explained earlier, branch target prediction is usually performed by using a branch target buffer (BTB) that records previously seen branch target addresses. A BTB is implemented as a table that stores pairs of branch instruction address and the corresponding target address, as shown in Figure 5.2. The BTB is accessed using instruction fetch addresses (i.e., PCs). Each entry in the BTB contains a remaining tag of the branch PC, its target address and optionally control bits (e.g., whether the branch is a special type of branch such as returns). When the current PC matches an entry within the BTB (i.e., a BTB hit), provided the branch is predicted taken by the branch direction predictor, the corresponding target address found in the BTB is used as the next instruction to be fetched and processed by the processor pipeline. In addition, the new PC is used to index in the BTB to predict whether the next instruction is a branch and whether the control flow will follow sequential execution or not.

Branch target prediction is a speculative action. Like any speculative action within the processor, there must be a mechanism that verifies whether the speculation was correct and that safely recovers the program state in case of misspeculation. Thus, the branch instruction will eventually be executed in the pipeline and the results of the execution will be compared with the speculative prediction. For the BTB, if the real branch target is different than the one stored in the BTB, the BTB is updated with the new branch target. In addition, the processor recovers from the branch misprediction and the instruction fetch is redirected to the correct instruction.

Branch target prediction is vital for branch prediction. Even if the branch direction predictor
decides that a particular branch is taken, if the branch target information is not present in the BTB, a next instruction address cannot be determined before branch execution. In this case, the processor can choose to stall the front-end or to continue fetching instructions on the sequential path. Once the branch is executed in the pipeline, the fetch unit can be redirected to the correct address if the branch proved to be indeed taken. Thus, ideally, BTBs would be sufficiently large to record all taken branches within programs. Commercial applications usually exhibit large branch footprints that are not captured by current BTBs. In the next section, we discuss how virtualization can be applied to conventional BTBs to create the illusion of a larger, higher accuracy dedicated BTB.

5.2 Phantom-BTB: A Virtualized Branch Target Buffer Design

In this section, we present Phantom-BTB, a virtualized BTB design that augments a dedicated, conventional, first-level BTB with a larger virtual table, as shown in Figure 5.3. The virtual
table does not have fixed dedicated on-chip storage. Instead, its entries are allocated on demand, at cache line granularity, in the second level cache, competing for cache capacity with application data and instructions. The purpose of the virtual table is to capture branch information that can be fetched into the dedicated BTB to improve its perceived accuracy. Compared to the general PV architecture presented in Chapter 3, the dedicated BTB corresponds to the `PVCache`, while the virtual table corresponds to the `PVTable`.

![Diagram](image)

Figure 5.3: Phantom-BTB augments a dedicated BTB with a virtual table that is allocated on demand in the on-chip caches.

Application performance is highly sensitive to branch prediction latency. Stalling the processor front-end while branch information is retrieved from the virtual table is not a viable option since this can lead to performance loss. Accordingly, the main challenge for Phantom-BTB is compensating for the high access latency of the virtual table. To address this challenge, Phantom-BTB employs branch metadata prefetching. For this purpose, Phantom-BTB exploits the temporal correlation in the BTB miss stream. Temporal correlation in the instruction stream has been used extensively to implement data and instruction prefetching [11, 13, 31]. It has also been shown that branches exhibit temporal correlation. For example, correlating branch direction predictors [3] rely on temporal correlation to improve the accuracy of branch direction prediction. While branch direction predictors rely on temporal correlation in the branch access stream, this work finds that temporal correlation persists even in the restricted BTB miss stream.
As applications execute, they tend to follow similar paths in the instruction stream [31]. The key idea behind Phantom-BTB is to record in the virtual table targets for branches that miss on a program path and reuse these targets on path repetition, as shown in Figure 5.4. Upon re-encountering a similar path in the execution of the program, the branch information stored in the virtual table is retrieved into a prefetch buffer. Provided sufficient overlap across program paths exists, some of the branches retrieved from the virtual table will be repeated in the branch stream without being present in the dedicated BTB. Thus, the perceived capacity of the dedicated BTB is increased by augmenting it with the extra branch information stored in the virtual table. The performance improvement of Phantom-BTB highly depends on the timely prefetching of branch information from the virtual table. While perfect repetition in the BTB miss sequences is desirable, it is not necessary for Phantom-BTB’s success. Similar to any prefetching technique, as long as a prefetch trigger brings in useful BTB entries, Phantom-BTB ends up increasing the perceived capacity of the BTB, hence improving application performance.

![Figure 5.4: Phantom-BTB: Exploiting program path repetition](image)

Next, we present in detail the design of the Phantom-BTB, compare it to conventional BTBs, and discuss the implications of the virtualized design on the cache hierarchy.

### 5.2.1 Phantom-BTB Design

Phantom-BTB adds three components to a dedicated conventional BTB:
• a virtual table that stores branch target information

• a temporal group generator that records temporally correlated branches and installs them in the virtual table

• a prefetch engine responsible for prefetching branch information from the virtual table

Figure 5.5: Phantom-BTB design

Figure 5.5 shows the main components in the Phantom-BTB design. Upon branch resolution, branch information is used to update the corresponding entry in the dedicated BTB. Branches that are not found in the dedicated BTB are also sent to the temporal group generator. Once several branch targets are collected, the corresponding temporal group is installed in the virtual table. Branches that miss in the dedicated BTB are used as prefetch triggers that bring the metadata stored in the virtual table in a separate prefetch buffer. The prefetch engine is responsible for orchestrating the prefetching of the branch metadata from the virtual table. Next, we explain in more detail the role and the functionality of each of the new components introduced in the Phantom-BTB design.

Virtual Table

The main component that Phantom-BTB adds to a conventional BTB is a virtual table. This virtual table collects branch target information as the application runs. The table is used to
emulate a large BTB by prefetching its contents into the dedicated BTB. To ensure timely delivery of branch target information, the virtual table’s organization is decoupled from the dedicated BTB.

The metadata table does not have fixed dedicated physical storage. Instead, it lives in the L2 cache, allocated on demand at cache line granularity. A contiguous region in physical address space is reserved for the purpose of indexing in the virtual table. Thus, the table has associated reserved physical address space (i.e., physical addresses that are not exposed to the operating system) for the sole purpose of installing its contents in the on-chip memory hierarchy in a non-intrusive way. Since this address space is used only for indexing purposes, it does not need to correspond to physical space in main memory. We choose not to propagate metadata off-chip, since off-chip latency is prohibitively high for timely BTB prefetches. To avoid the propagation of predictor metadata requires simple extensions to the memory hierarchy, as detailed in Section 5.2.2.

A fundamental challenge of the Phantom-BTB design is the organization of the branch metadata in the virtual table such that it can be prefetched in a timely fashion, increasing the perceived hit rate of the dedicated BTB. Building on previous knowledge on branch and path correlation [3, 48] the virtual table stores groups of temporally correlated branches, which we refer to as temporal groups.

A straightforward option for generating temporal groups is to aggregate target information for branches that are accessed closely in time. This option would result in high information redundancy across groups, which reduces the effective capacity of the virtual table. Instead, Phantom-BTB uses the dedicated BTB itself as a filter, and generates temporal groups including only the branches that miss in the dedicated BTB, as shown in Figure 5.6. Several consecutive branches that miss in the dedicated BTB are packed together with their target information in a memory block equal in size to the L2 cache block. This memory block is sent and installed into the L2 cache, similarly to dirty-evict lines from the L1 data cache. Later on, this memory block can be retrieved from the L2, and its information can be used to augment
Conceptually, the virtual table is tag-less and direct mapped. Each temporal group is associated with a trigger that is used for memory address calculation. The triggering memory address and the geometry of the virtual table dictate where each temporal group is stored. In our design, we choose a region of code surrounding the last branch in the previous temporal group to trigger the prefetching of the temporal group. The same address is also used for installing the entry in the L2 cache. Thus, indexing of the virtual table is performed using the least significant portion of a branch address shifted right, such that each temporal group is associated with a small region of code. Any branch within this region of code that misses in the dedicated BTB triggers the prefetch of the temporal group. Using a branch previous to the group to index into the virtual table allows to tolerate the latency of the L2 cache when temporal groups are retrieved from the on-chip cache.

**Temporal Group Generation**

Upon branch resolution, the out-of-order core sends feedback to the BTB with updated branch target information. The temporal group generator (TGG) collects consecutive branches that missed in the dedicated BTB together with their target information and packs them into a
contiguous memory block. This memory block is installed in the L2 cache. The number of branch entries in the TGG is dictated by the branch target metadata representation in the virtual table.

Each branch entry stores the full branch address, a two-bit branch type field and 30 bits from the target. This allows to form temporal groups of six branches, assuming 48-bit virtual addresses and 64-byte L2 cache lines. Hence, the TGG is a simple table that contains only six entries. Once the TGG’s entries fill up, a temporal group is formed, and the corresponding memory request and data is sent to the L2 cache. The address of the last branch in the group is used to create the virtual table index for the next temporal group. This index generates the physical address of the memory request that installs the temporal group in the L2 cache, as depicted in Figure 5.7

![Figure 5.7: Phantom-BTB: Temporal group indexing](image)

**Prefetch Engine**

Branches that miss in the dedicated BTB trigger branch metadata prefetching. The address of the branch is used to generate a memory request that is sent to the L2 cache to retrieve the temporal group associated with the branch. The prefetch engine contains several miss status handling registers (MSHRs) to keep track of in-progress prefetches to avoid generating requests
for the same temporal group. Upon receiving the L2 cache reply, the temporal group metadata is unpacked and the resulting BTB entries are installed in a FIFO prefetch buffer.

The prefetch buffer is accessed in parallel with the dedicated BTB. Upon a BTB miss and a prefetch buffer hit, the branch found in the prefetch buffer is used for the target prediction. The branch information is also installed into the dedicated BTB and removed from the prefetch buffer. The prefetch buffer serves two roles. First, it reduces pollution of the dedicated BTB by allowing the insertion of only those entries that are requested by the processor. This is important as the prefetched temporal groups only partially overlap with the current execution path. Second, it avoids the need for multi-entry updates to the dedicated BTB. Entries are written into the dedicated BTB one-by-one as requested by the processor using the existing BTB ports.

5.2.2 Implications of BTB Virtualization for the On-Chip Memory Hierarchy

The L2 cache is metadata oblivious: it receives memory requests from the Phantom-BTB and stores predictor metadata the same way it stores application data and instructions. Simple hardware extensions are required to orchestrate the interaction between the L1 caches and the Phantom-BTB with the L2 cache, and to avoid propagating metadata off-chip.

Redirecting the replies from the L2 cache to the Phantom-BTB can be performed by filtering the physical addresses associated with predictor metadata, or by tagging memory requests with their originator. Similar information is required to route memory replies to instruction and data caches respectively.

In our Phantom-BTB design, evictions of cache lines that contain metadata (i.e., cache lines with addresses corresponding to the virtual table) are dropped, without being forwarded to the off-chip memory hierarchy. In addition, upon a miss in the L2 cache for a virtual table address request, a reply that contains no data is sent back to the Phantom-BTB and the request is not propagated off-chip.
Finally, coherence is not necessary for predictor metadata, and thus, memory requests associated with the virtual table can bypass the coherence mechanism. Bypassing the coherence mechanism can eliminate a couple of cycles from the latency of prefetching branch metadata, which could be important since branch prediction is latency sensitive.

The changes of the on-chip caches necessary for supporting the virtualization of BTBs are minimal and we believe they are straightforward to implement in real hardware.

### 5.2.3 Phantom-BTB versus Conventional Branch Target Buffers

There are several differences between a traditional second-level BTB and the virtual table in Phantom-BTB. First, the virtual table implicitly contains information on the temporal correlation of BTB entries. This information is encoded in the way groups are formed. Second, the virtual table contains redundant information across groups (i.e., one BTB entry can be part of multiple temporal groups). Third, while the maximum size of the virtual table is fixed for indexing purposes, its entries are allocated on demand, at cache line granularity in the L2 cache. Depending on the application access patterns and demand, some virtual table indexes may not be used at all, while the temporal groups for others may be evicted from the L2 cache. Accordingly, the virtual table size is an upper bound on the L2 space it occupies at any given point. For all these reasons, a capacity comparison between the virtual table and a conventional BTB is not meaningful.

Although motivated by applications with large instruction footprints, Phantom-BTB naturally adapts to small branch working sets as well, since all its activity is generated by BTB misses. If the dedicated BTB is sufficiently large to accommodate most branches from the working set of the application, temporal groups are generated only for compulsory misses and during program phase changes. Once the dedicated BTB warms up, it rarely incurs misses, and only few temporal groups are sent to and prefetched from the cache hierarchy. Assuming traditional cache replacement policies, application demand for cache capacity naturally evicts infrequently accessed predictor metadata. Section 5.3.3 demonstrates experimentally
that Phantom-BTB dynamically adjusts its request for L2 resources according to the application’s demands for additional BTB resources.

The previous observation justifies a Phantom-BTB design even when it is possible to build a large and fast conventional BTB, which is not possible with today’s technology. A conventional, large BTB dedicates resources for only one purpose. These resources are wasted for applications that do not need the large BTB. A virtualized BTB uses a practically-sized dedicated BTB and relies on the on-chip memory hierarchy to expand the perceived BTB capacity on demand, as the applications need it. The disadvantage of the static resource allocation in traditional BTB designs is more pronounced in homogeneous chip-multiprocessors where the cost of each BTB multiplies by the number of cores on chip. Phantom-BTB reduces the amount of storage per core and, thus, the area savings multiplies by the number of cores. Even when implementing large BTBs is possible, Phantom-BTB can improve performance, as long as the application benefits from additional BTB capacity, as it is shown in Section 5.3.3.

The Phantom-BTB design could lead to increased pressure on the on-chip cache capacity and bandwidth. The experimental evaluation of Phantom-BTB presented in Section 5.3.3 quantifies the impact on the on-chip cache hierarchy and shows that a large virtual table can reside in reasonably-sized on-chip caches, having negligible interference with application data.

5.3 Experimental Analysis

In this section, first, we describe the experimental methodology and the benchmarks used in evaluating the Phantom-BTB design. Second, we present a series of experiments that motivate, evaluate and compare the Phantom-BTB design with conventional BTB organizations. In particular, we analyze the performance of Phantom-BTB and the impact of virtualization on the memory hierarchy. As the size of dedicated BTBs may increase in the future, we also verify that virtualization remains beneficial when used in conjunction with large dedicated BTBs.
5.3.1 Methodology

The dynamically-scheduled, super-scalar uniprocessor configuration described in Table 5.1 was simulated using Flexus [40], an open-source, full-system simulator based on Simics [41]. The simulated processor has an eight-stage pipeline. A deeper pipeline would be more sensitive to improvements in branch prediction accuracy.

<table>
<thead>
<tr>
<th>Branch Predictor</th>
<th>Fetch Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>8K GShare, 16K bi-modal</td>
<td>up to 4 instructions per cycle</td>
</tr>
<tr>
<td>16K selector, 1 branch per cycle</td>
<td>64-entry fetch buffer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ISA &amp; Pipeline</th>
<th>Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>UltraSPARC III ISA, 4GHz</td>
<td>256-entry</td>
</tr>
<tr>
<td>8 stage pipeline</td>
<td>64-entry LSQ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main Memory</th>
<th>Issue/Decode/Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 GB, 400 cycles</td>
<td>4 instructions per cycle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L1D/L1I</th>
<th>UL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>64KB 2-way set-associative</td>
<td>4MB, 16-way set-associative</td>
</tr>
<tr>
<td>64-byte blocks, LRU replacement</td>
<td>64-byte blocks, LRU replacement</td>
</tr>
<tr>
<td>2 cycle latency</td>
<td>12 cycle latency</td>
</tr>
</tbody>
</table>

Table 5.1: Baseline system configuration

The branch direction predictor remains unchanged for all experiments to isolate the impact of branch target prediction on application performance. The geometry of the conventional BTB designs is varied throughout the experiments, as specified in the text. Unless otherwise noted, the dedicated first-level conventional BTB included in the Phantom-BTB design contains 1K entries, and the virtual table consists of 4K temporal groups. A temporal group is designed to fit within a single L2 cache line and it contains metadata for six branches. Sizes are measured in number of branch entries for the dedicated BTBs and in number of temporal groups for the virtual table. The region of code that triggers prefetching of a temporal group spans 32 instructions. This value was empirically found to perform the best for the benchmarks we considered in our experiments.

The dedicated BTBs are indexed with the lower bits of the branch address. Each BTB entry contains the remaining branch tag, a 2-bit branch type field, and the 30 low-order bits of the target. This allows the BTB to capture all branch targets within a 4GB of contiguous
virtual space. For the applications studied, this covers all branches. The target of the branch is obtained by concatenating the upper bits of the branch with the lower bits of the target. All Phantom-BTB designs use a 64-entry prefetch buffer.

The experimental results presented in Section 5.3.3 are obtained through a combination of functional and cycle-accurate timing simulation. Specifically, the results that measure the BTB accuracy and the impact on the cache hierarchy are obtained using functional simulation of two billion cycles during a steady state of the workload. The first one billion cycles are used as warm-up and results are presented for the second billion cycles. All speedup measurements use the Flexus cycle-accurate timing simulator [40]. Each workload was run for 500 million cycles starting from a steady state of the application. To account for non-determinism in multi-threaded workloads, small perturbations are artificially introduced in the main memory latency [49]. For these experiments, on each L2-cache miss, the memory latency is increased by a uniformly distributed pseudo-random number of up to ten cycles. The average memory latency is the same for all runs. However, the timing perturbations lead to slightly different execution paths and different runtime results. In all speedup graphs, runtime error bars are shown for 95% confidence intervals. For each bar, ten different experiments are run. The variations between the different runs are marginal. In general, the performance of the first 10-20 million cycles in the execution of the benchmark are a good indicator of the performance obtained for the larger simulation interval.

5.3.2 Benchmarks

Table 5.2 describes the commercial workloads used in our analysis:

- The TPC-C v.3.0 online transaction processing workload running on IBM DB2 v8 ESE
- Two queries from the TPC-H decision support workload running on IBM DB2 v8 ESE
  (Query 2 and Query 17)
• The SPECweb99 benchmark running with Apache HTTP Server v2.0 and Zeus Web Server v4.3, respectively.

The web server workloads were driven with separately simulated client systems. The results present the server activity.

<table>
<thead>
<tr>
<th>Online Transaction Processing (TPC-C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decision Support (TPC-H on DB2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
</tr>
<tr>
<td>Q17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Web Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
</tr>
<tr>
<td>Zeus</td>
</tr>
</tbody>
</table>

Table 5.2: Benchmarks

5.3.3 Experimental Results

This section discusses a series of experiments that motivate and evaluate the Phantom-BTB design. First, we discuss how the accuracy of branch prediction varies with the size of BTBs. This experiment demonstrates that commercial applications benefit from large BTBs, which motivates the virtualization of BTBs. Next, we compare the performance improvements of Phantom-BTB against that of several conventional BTB organizations. Since predictor virtualization trades-off dedicated resources for cache capacity, we show that the impact of Phantom-BTB on the on-chip memory hierarchy is minimal.

We also examine the sensitivity of the Phantom-BTB design to the virtual table size and the L2 capacity. To verify that Phantom-BTB is a viable option even when larger dedicated BTBs are possible to implement, we show that Phantom-BTB improves application performance even when applied in conjunction with larger conventional BTBs, as long as the application benefits from additional BTB capacity. Finally, we compare the Phantom-BTB design with an alternative virtualized design where the virtual table organization mirrors the organization of the
dedicated BTB. This experiment shows the importance of metadata prefetching for latency-sensitive predictors.

**Branch Target Buffer Accuracy versus Capacity**

To motivate the Phantom-BTB design, Figure 5.8 presents the number of mispredicted branches per 1K instructions (MPKI) for conventional branch target buffers as a function of BTB capacity expressed in number of entries. The size of the BTB is varied from 1K to 32K entries. All BTB configurations are four-way set-associative. Further increasing the associativity shows marginal improvements. A branch is considered mispredicted when either its direction or its target is mispredicted. Increasing the number of BTB entries affects only the accuracy of branch target prediction, since separate hardware structures are used for predicting branch directions. Branch direction prediction is orthogonal to target prediction and the branch direction predictors remain unchanged for all experiments presented in this chapter.

![Figure 5.8: The accuracy of conventional BTBs as a function of the number of BTB entries](image)

These experimental results show that the workloads considered have a high number of static branches and, as a consequence, they all benefit from larger BTBs, albeit to different extents. On average, increasing the number of BTB entries from 1K to 16K results in 4.5 times fewer mispredicted branches. Further increasing the BTB size to 32K entries leads to marginal improvements. These results corroborate previous findings about instruction footprints of commercial workloads [7, 17, 18]. In particular, Hilgendorf et al. show four different
traces of commercial applications that benefit from up to 16K-entry BTBs [7].

To better understand the trade-off between space requirements and accuracy, Table 5.3 shows the size in kilobytes for each BTB configuration. Although the accuracy of a 16K-entry BTB is far higher than that of smaller BTBs, its storage overhead is considerable. To put things into perspective, the size of a 16K-entry BTB is close to twice the L1 data cache size of current modern processors.

<table>
<thead>
<tr>
<th>BTB Entries</th>
<th>1K</th>
<th>2K</th>
<th>4K</th>
<th>8K</th>
<th>16K</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTB Size</td>
<td>8.75KB</td>
<td>17.25KB</td>
<td>34KB</td>
<td>67KB</td>
<td>132KB</td>
</tr>
</tbody>
</table>

Table 5.3: BTB storage

The results presented in this section expose the trade-off between prediction accuracy and storage overhead. The Phantom-BTB design addresses this trade-off by approximating the accuracy of a large BTB without the area and latency penalty.

**Phantom-BTB Accuracy under Different Temporal Group Indexing Schemes**

The temporal groups stored in the virtual table are associated with a region of instructions around a branch that previously missed in the dedicated BTB. This region is determined by shifting the virtual address of the branch to the right by \( n \) bits. The number of shifting bits dictates the region of code around the branch that will trigger the prefetch of a particular temporal group.

Figure 5.9 shows the branch MPKI measured with Phantom-BTB when using different temporal group indexing schemes for the virtual table. In this experiment, the virtual table contains 4K temporal groups. In the graph, each bar corresponds to a different value for the number of bits used in the address shift (e.g., S0 corresponds to no shifting, while S5 corresponds to shifting by five bits). With no shifting, a temporal group is correlated with a specific branch. Shifting allows the correlation with a set of branches within a contiguous region of instructions. The latter is preferable since application control flow exhibits local variations.
The results show that TPCC and the web server workloads are more sensitive to the indexing scheme than the TPCH workloads. Overall, no shifting or too much shifting can be detrimental since the correlation between misses and temporal groups is either too strict or too loose, respectively. When no shifting is performed, fewer groups are prefetched, while a high index shifting results in useless prefetches. Five-bit shifting (S5) performs the best for the benchmarks studied. For the rest of the experiments, we use S5 as our indexing scheme. Thus, the temporal group indexing is performed by shifting five bits of the virtual address (i.e., instructions are split into 32-instruction groups for indexing purposes) to maximize the instruction region that triggers prefetches while minimizing useless prefetches.

**Phantom-BTB Accuracy**

Figure 5.10 compares the branch MPKI achieved with Phantom-BTB to that of conventional BTBs with 1K, 2K and 4K entries, respectively. The accuracy of the Phantom-BTB with a 1K-entry dedicated table is within 3% of a conventional 4K-entry BTB on average, while in terms of dedicated storage, the Phantom-BTB adds only 8% storage overhead to a conventional 1K-entry BTB, as explained in Section 5.3.4. The Phantom-BTB design requires 3.6 times less dedicated storage than a conventional 4K-entry BTB. The next section shows that the accuracy of the Phantom-BTB translates into application performance improvements.
Figure 5.10: The accuracy of Phantom-BTB compared to conventional BTB configurations

Phantom-BTB Performance

This section compares the performance of Phantom-BTB against several conventional BTB configurations. Figure 5.11 shows the percentage of IPC increase relative to a baseline 1K-entry BTB for the following configurations:

- **1K BTB-64VB**: 1K-entry BTB with a 64 entry victim buffer

- **1.25K BTB**: 1.25K-entry BTB

- **4K BTB 1-cycle**: 4K-entry, single-cycle access latency BTB

- **1K Phantom**: Phantom-BTB with a 1K-entry dedicated BTB and a 4K temporal group virtual table

- **16K Ideal**: 16K-entry BTB with an optimistic single-cycle access latency.

The last configuration, while not practical, shows the performance potential of a large conventional BTB ignoring latency effects. The Phantom-BTB configuration (1K Phantom) applied to a 1K-entry dedicated BTB uses a 64-entry prefetch buffer, and a 4K-group virtual
table. The dedicated tables in all configurations are four-way set associative, except the 1.25K BTB which is five-way set-associative.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>1K</th>
<th>1K-64VB</th>
<th>1.25K</th>
<th>4K</th>
<th>16K</th>
<th>1K Phantom-BTB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>8.75KB</td>
<td>9.36KB</td>
<td>9.72KB</td>
<td>34KB</td>
<td>132KB</td>
<td>9.45KB</td>
</tr>
</tbody>
</table>

Table 5.4: Dedicated storage for different BTB configurations

Phantom-BTB requires additional storage compared to the baseline. For this reason, we included in the performance evaluation two BTB configurations that use the additional storage in traditional ways: a victim buffer (1K BTB-64VB) and a table with an extra entry in each set (1.25K BTB). Both these configurations use additional resources comparable to those introduced by Phantom-BTB. Table 5.4 shows the amount of dedicated storage for each configuration.

The results in Figure 5.11 show that Phantom-BTB makes good use of the hardware overhead it introduces on top of the conventional 1K-entry BTB. The 1K-Phantom achieves an average speedup of 6.9% and up to 12.7% for TPCC, the most BTB demanding of our applications. The conventional approaches where the extra hardware is used for a victim buffer or for adding one more way to each set in the dedicated BTB show little improvement (less than 1.5% on average).
On average, the 1K-Phantom design performs within 1% of a 4K-entry single-cycle conventional BTB, while using 3.6 times less dedicated storage. For TPCC, which is more sensitive to BTB accuracy, the 1K-Phantom outperforms the 4K-entry BTB.

Improving on the timeliness of the prefetcher and reducing the redundancy across temporal groups is key for approaching the performance potential of the 16K-Ideal BTB. On average, 55% of the prefetched entries are already present in the dedicated BTB when they arrive from the virtual table.

**Impact of BTB Virtualization on the On-Chip Memory Hierarchy**

The virtualized BTB design competes with the application for L2 capacity and bandwidth. This section quantifies the impact of virtualization on the on-chip memory hierarchy by comparing the L2 misses and accesses per 1K instructions before and after virtualization.

As Figure 5.12 shows, Phantom-BTB does not cause pollution in the L2 cache. The number of misses per 1K instructions that the application suffers due to BTB virtualization is negligible. The relative increase in the number of misses observed by the applications with Phantom-BTB is at most 2.4%, and 1.2% on average. This result is expected, since the footprint of the application metadata, in this case, branch metadata, that is stored in the on-chip caches is, in general, much smaller than the application data and instructions.

Figure 5.13 shows the number of L2 requests per 1K instructions with a 1K conventional BTB and 1K Phantom-BTB. As anticipated, Phantom-BTB increases the pressure on accessing the L2 cache, since it stores and retrieves temporal groups from the virtual table allocated in the on-chip caches. However, even in the worst case, for the TPCC workload, the number of accesses per 1K instructions is low, reaching only 90. For the rest of the workloads, the L2 traffic with Phantom-BTB is lower than for TPCC without virtualization. This implies that, if the on-chip cache hierarchy can support demanding applications such as the TPCC workload, there is sufficient bandwidth remaining for additional traffic introduced by virtualization for less demanding applications.
The relative increase in the number of L2 requests with Phantom-BTB depends on the accesses to the virtual table. The results in Figure 5.13 indicate that Phantom-BTB dynamically adapts to application demands. As the figure shows, the L2 traffic varies according to the need of the application for additional BTB capacity. For example, TPCH Query 17, which exhibits the smallest improvement from larger BTBs, incurs the lowest increase in L2 traffic, while TPCC, the most BTB demanding application among the considered workloads, incurs a much larger increase in L2 traffic.
For all experiments presented in this chapter, the queues accessing the L2 cache are oblivious to the difference between on-demand requests and metadata requests. We found that, for the benchmarks considered and the uniprocessor design evaluated in our experiments, no request prioritization was necessary. However, for large number of cores on chip, as the pressure on the L2 cache increases, increasing the priority of application demand request over metadata requests may be needed to avoid increasing the latency of memory demand accesses.

**Sensitivity to the Virtual Table Size**

Allocating the virtual table in the on-chip caches and prefetching the branch metadata allows increasing the perceived capacity of the BTB without dedicating precious on-chip resources. The performance of the Phantom-BTB depends on the capacity of the virtual table. In this section, we study the sensitivity of the Phantom-BTB performance to the size of the virtual table.

Figure 5.14 shows the performance improvement for Phantom-BTB compared to a baseline 1K-entry conventional BTB when varying the size of the virtual table (i.e., 1K, 2K, or 4K temporal groups). We restrict the size of the virtual table to 4K temporal groups to avoid potential pollution of the on-chip caches. All Phantom-BTB configurations use a 1K-entry first-level dedicated BTB. While all applications experience a higher speedup with the 4K-group virtual table, the difference is more visible for TPCC. Even with a smaller virtual table, performance improves for all applications. This result suggests that the virtual table size can be tuned to obtain performance even with smaller caches, as long as there is excess capacity that can be used. The next section shows experimental results that confirm this observation.

**Phantom-BTB with Smaller L2 Caches**

The Phantom-BTB design is motivated by increased capacity on-chip caches that show diminishing returns. With smaller caches, Phantom-BTB may cause pollution when competing with the application for cache capacity. In this section we report the performance of Phantom-BTB
Figure 5.14: Phantom-BTB performance improvement with different virtual table sizes when run on processor configurations with smaller L2 cache size.

Figure 5.15 shows the application speedup obtained by a 1K-entry Phantom-BTB design compared to the one obtained by a 1K-entry conventional BTB with two different L2 cache configurations: 1MB and 2MB (all previous results used a 4MB L2 cache). The virtual table in Phantom-BTB is configured to store up to 4K groups for the 2MB run, while for the 1MB configuration, the table is reduced to 1K temporal groups. In both configurations, the improvement obtained by higher branch prediction accuracy overcomes the negative effects of cache pollution, except for TPCC that incurs a slowdown of 2% with the 1MB L2 cache. With smaller caches, these workloads become more sensitive to memory performance and the improvements in branch prediction accuracy become less important.

Phantom-BTB could be disabled using a feedback mechanism that indicates whether the metadata prefetching is beneficial. When prefetched branches are not utilized, the virtualization can be turned-off and the metadata will be replaced from the on-chip cache by the application data and instructions.

**Phantom-BTB Applied to Larger Dedicated BTBs**

All results so far analyzed the Phantom-BTB design when applied in conjunctions with a dedicated 1K-entry BTB. This section demonstrates that Phantom-BTB can improve performance
over larger dedicated BTBs, as long as the application benefits from additional BTB capacity. 

Figure 5.16 shows the performance improvement for Phantom-BTB with 1K, 2K, and 4K-entry first-level BTBs compared to using the corresponding conventional BTBs alone. The baseline in each run is different (i.e., 1K, 2K and 4K-entry BTB respectively). The speedups vary according to the application’s need for a larger BTB. For example, virtualizing a 4K-entry dedicated BTB improves performance for TPCC by 7.1%. In contrast, for TPCH Query 17, which does not benefit from a larger than 4K-entry BTB, Phantom-BTB does not show any change in performance. This supports the observation that Phantom-BTB dynamically adapts to the application demands for additional BTB capacity.

**Phantom-BTB and the Organization of the Virtual Table**

For all the results presented so far, the organization of the virtual table was decoupled from the one of the dedicated BTB to allow for timely prefetching of branch metadata. In this section, we investigate the performance of a virtualized design in which the virtual table has the same organization as the dedicated table, and only the size is different between the two (i.e., the virtual table is a much larger version of the dedicated table). In this design, the dedicated BTB acts as a cache for a much larger, second-level, virtual BTB stored in the L2 cache. This
Figure 5.16: Phantom-BTB performance improvement over dedicated BTBs of different sizes

The experiment was performed to demonstrate the need of metadata prefetching for such a latency sensitive prediction.

Figure 5.17 shows results for a virtualized BTB using a virtual table that stores a large second level BTB (PV-BTB) and the Phantom-BTB design. The baseline is a conventional 1K-entry BTB. Both Phantom-BTB and PV-BTB use a virtual table designed to occupy at most 4K blocks in the L2 cache. In PV-BTB, each entry in the virtual table stores a set from the dedicated BTB. The dedicated BTB in both schemes is a conventional 1K-entry BTB.

For the PV-BTB design, when a branch is not found in the dedicated BTB, a request for the cache line that may contain the particular branch in the virtual table is sent to the cache hierarchy. If the memory request is not received in time, the processor’s front end is stalled until the memory request returns. Note that sending a memory request to the cache hierarchy does not guarantee that the branch information is found. The branch may not be in the virtual table or the virtual table entry may have been evicted from the on-chip caches.

As expected, branch target prediction is not tolerant of the L2 latency and, in consequence, the PV-BTB incurs slowdowns for all benchmarks except TPCC. TPCC is a BTB-hungry application for which the extra latency to predict the correct path of execution offsets the branch misprediction penalty. TPCC is notorious for branches that depend on load instructions that miss in the on-chip cache. Thus, for these branches, branch resolution takes hundreds of cycles.
(i.e., main memory latency), and, as a result, waiting for the branch information to be returned from the on-chip cache proves beneficial.

![Figure 5.17: Phantom-BTB compared to PV-BTB](image)

The PV-BTB design proves ineffective because the branch stream exhibits insufficient temporal and spatial locality. Temporal locality would amortize the L2 latency over multiple accesses to the same entry, while spatial locality would do so over accesses to other entries of the same set. Because of the large instruction footprints, consecutive instances of the same branch appear far apart in the branch stream. Due to BTB set indexing, spatially correlated branches end up in different BTB sets. Spatial correlation within a BTB set could be enforced by using a higher portion of the instruction address to access the BTB. This effectively compresses the instruction stream, forcing neighboring branches to fall onto the same set. We have experimented with increasing spatial locality through different indexing schemes, however, the results look similar to the PV-BTB design.

### 5.3.4 Hardware Overhead for Phantom-BTB

The breakdown of the additional hardware introduced by a Phantom-BTB design with a 4K-group virtual table and a 64-entry prefetch buffer is as follows:

- Virtual table start address: 40 bits
CHAPTER 5. VIRTUALIZING BRANCH TARGET BUFFERS

- Temporal group generator: 12 bit index for the virtual table, six branch entries of 78 bits each

- Prefetch engine: 16 MSHRs of 12 bits each

- Prefetch buffer: 64 entries of 78 bits each.

For dedicated storage computations, we assume 48-bit virtual addresses and 46-bit physical addresses, similarly to the Intel Nehalem processor [16]. The total hardware overhead introduced by the Phantom-BTB design is 0.7 KB. This represents an 8% storage overhead on top of a 1K-entry conventional BTB.

5.4 Related Work

In this section we briefly describe the research done in branch prediction which is closely related to our work. In the end, we report on a related project that applies virtualization to branch direction prediction.

5.4.1 Research in Branch Prediction

Phantom-BTB is motivated by the need for larger BTBs in commercial workloads. Several previous studies have characterized commercial workloads, finding that they exhibit large instruction footprints with high number of static branches [17, 18, 19, 45]. In particular, Annavaram et al. invalidate the common belief that high branch misprediction in commercial application is due to the complex branching patterns. Instead, the authors show that the high branch misprediction rate in commercial applications comes from aliasing due to high static branch count [18]. Once the tables used in branch direction prediction are large, the misprediction decreases significantly, becoming lower than that of common benchmarks such as gcc. Hilgendorf et al. characterize four traces of commercial applications and show that they benefit from large BTBs of up to 16K entries [7]. Our results show similar trends.
The concept of a branch target buffer has been studied extensively. Sussenguth describes one of the first BTB designs [15]. A detailed overview of BTB design alternatives is given by Lee and Smith [50] and by Perleberg and Smith [46], including hierarchical BTB designs. The Intel Nehalem processor incorporates a two-level BTB design; the details of this design are unknown [16], as it is with most performance-critical components in current processors.

Hierarchical BTBs aim at offering much of the accuracy of larger BTBs. Phantom-BTB shares this goal and, in addition, it avoids the area cost of larger BTBs and the potentially higher access latency. Reducing the area cost of BTBs is possible by encoding its information [51, 52, 53]. In some cases, these savings come at the expense of increased latency as the BTB is split into several structures. These techniques are orthogonal to Phantom-BTB as they can be applied to both the dedicated BTB and the virtual table, reducing the cost of the dedicated predictor and the cache pressure of the virtual table.

The BTBs studied in this work make no effort to improve accuracy for indirect branches, except for returns where a return address stack was used [54]. Improving prediction accuracy for indirect branches is possible by using several approaches such as pattern or history based prediction [51, 55, 56, 57]. Using any of these techniques would increase the performance potential of Phantom-BTB as they would reduce indirect branch misses. Moreover, it may be possible to apply virtualization to these mechanisms reducing their cost. This investigation is left for future work.

Several studies proposed ways of utilizing larger and slower branch predictors. Seznec et al. describe an overriding predictor where a larger, slower, but more accurate branch direction predictor is used to correct the predictions of a smaller, faster, and less accurate one [47]. Jimenez et al. explain the trade-off among performance, accuracy and latency in branch direction prediction and propose several techniques to compensate for larger and slower branch direction predictors, including using a small cache for fast access to recently accessed entries [58]. Their work relies on dedicated storage for the various tables. Phantom-BTB exploits the existing memory hierarchy and relies on prefetching to compensate for the longer latencies. Jimenez
proposes a pipelined branch direction predictor that uses prefetching to fetch correlated pattern history entries into a small and fast first-level predictor [59]. Similarly, Phantom-BTB uses prefetching of branch target information of temporally correlated branches instead of branch direction entries. However, the second-level table (i.e., the virtual table) in Phantom-BTB does not use dedicated storage and it has higher access latency, since the table resides in the second level cache. As a result, the trade-offs and solutions are different.

Phantom-BTB indexes temporal groups of branches with an earlier branch that missed in the dedicated BTB to compensate for the long access latency to the virtual table. This technique is similar to the proposal by Yeh and Patt that indexes branch target addresses using an earlier instruction that dominates the branch [60]. In Yeh and Patt’s mechanism, this allows the front-end to run ahead of the rest of the pipeline for wide-superscalar designs.

5.4.2 EXACT: Explicit Dynamic-Branch Prediction with Active Updates

Al-Otoom et al. [32] characterize the different causes behind branch misprediction in state-of-the-art branch direction predictors. The authors show that a significant percentage of mispredicted branches come from branches that depend on load instructions (called producer loads). A significant fraction of branch mispredictions stem from two dynamic instances of the same static branch that map to the same predictor entry, but for which the memory addresses or values of the producer loads are different. Motivated by this analysis, the authors propose the EXACT predictor, a novel predictor that specializes in providing direction outcomes for branches that depend on load addresses and values.

Store operations can modify memory values that are later on loaded and fed into branch outcomes. Such memory modifications can lead to branch mispredictions. To address this issue, one component of the EXACT predictor is responsible for translating store operations into “active updates” to the predictor information. The active update unit has two characteristics that make it an excellent candidate for virtualization. First, this unit requires a lot of storage resources to provide accurate updates. Second, these updates are tolerant to delays of hundreds
of cycles since, usually, the distances between stores and encounters with branches that they update are long.

The authors apply predictor virtualization to the active update unit and study the trade-off between accuracy and dedicated storage. For benchmarks with high demand on the active unit resources, virtualization eliminates 200-300KB of dedicated space for similar predictor accuracy.

5.5 Concluding Remarks

Applications with large instruction footprints, such as commercial workloads, expose the trade-off among accuracy, latency and hardware cost in BTB design. To address this trade-off, the research described in this chapter introduced Phantom-BTB, a novel virtualized BTB design. Phantom-BTB exploits the current trend towards larger on-chip caches and leverages it to expand the effective capacity of the BTB.

Full-system simulation of commercial workloads demonstrated that Phantom-BTB improves application performance over a 1K-entry BTB by 6.9% on average, and up to 12.7%, while the storage overhead is only 8% of a conventional 1K-entry BTB. Phantom-BTB performs within 1% of a conventional 4K-entry, single-cycle access BTB, yet using 3.6 times less dedicated storage. Experimental results also showed that Phantom-BTB remains effective even when applied on top of larger dedicated BTBs, especially for applications with high BTB demands.

Phantom-BTB is a “pay-as-you-go” design that dynamically adapts to the application demand for BTB capacity, without incurring a latency penalty or imposing a fixed allocation of resources. To achieve this, Phantom-BTB employs several key techniques. First, Phantom-BTB augments a dedicated BTB with a larger, virtual table. This virtual table collects predictor metadata at runtime and is allocated in the on-chip caches, at cache line granularity. Second, the organization of the virtual table is decoupled from that of the first-level predictor and only
the active entries of the virtual table are allocated in the second-level cache. In general, this
decoupling allows metadata to be restructured to improve prediction accuracy or latency of the
first-level predictor. It also opens up the possibility of using a sparsely populated virtual table,
a property that offers further flexibility in the allocation and grouping of predictor metadata.
In Phantom-BTB, metadata is organized to facilitate the prefetching of branch entries into the
dedicated BTB, increasing its perceived capacity. Third, metadata collection and retrieval is
triggered on misses incurred at the first level predictor. This allows the dynamic allocation of
cache resources according to application’s demand for predictor capacity.

The techniques employed in the virtualization of conventional BTBs are general and they
can serve as starting point in the virtualization of other latency-sensitive processor components,
such as translation look-aside tables.
Chapter 6

A Virtualized Object Pointer Prefetcher for Managed Runtimes

The last computing decade has witnessed a widespread adoption of managed programming languages [20]. The popularity of these languages stems from their ease of programming, rich libraries and managed runtimes. However, the high level abstraction of these languages comes at the cost of lower performance compared to non-dynamic languages [61].

We believe that runtime systems can benefit from a closer collaboration with the underlying hardware to provide a more efficient utilization of resources, and, hence, higher performance. In general, runtime systems observe rich information about the application characteristics and behavior, such as hot methods, hot fields, object layout, pointer location and mutation. The underlying hardware is unaware of any software-level constructs and inferring this type of information without help from the software level can be expensive. In this work, we propose communicating software level details to the hardware through a thin interface to help the processor make informed decisions. This collaboration helps achieve rich and accurate runtime information with fast and low overhead hardware prediction.

One barrier for this type of collaborative optimizations could be the amount of information communicated by the software that needs to be stored and manipulated at the hardware level.
Traditionally, processors limit the amount of resources available to each optimization, which may result in overall ineffective optimizations. However, through predictor virtualization, we can utilize the capacity of the memory hierarchy (including processor caches) to store the metadata communicated by the software level.

In this chapter, we focus on one important source of performance degradation in runtime systems, namely main memory behavior [62]. Hardware data prefetchers have been shown to have limited effectiveness for managed runtime systems [62, 63, 64]. The difficulty of predicting memory accesses of managed applications relates to the dynamic memory layout and the heavy use of pointers in runtime systems. Reconstructing application memory layout information in hardware [6, 34] is only partially successful and fundamentally more expensive than leveraging the information that is readily available in the software runtime. A different approach is to delegate the responsibility of prefetching completely to software, by inserting prefetch instructions into application code [62, 65]. Unfortunately, a software only prefetching strategy also poses several unique challenges, which we discuss in more detail in Section 6.4.

As a proof of concept, we present the design and evaluation of an object pointer prefetcher for Java applications, in which the runtime is responsible for exposing application pointers to hardware, while the hardware uses this information to generate pointer prefetches, anticipating future memory accesses.

In our system, we extend the runtime system to communicate to the underlying hardware which memory addresses are pointers and the connections between them. The code modifications required are simple since the information communicated to the hardware is readily available in the Java runtime system. With pointer information in place, the hardware generates targeted data prefetches that follow the pointer links as the application runs.

In general, complex applications exhibit large pointer footprints. Storing all object pointer metadata in on-chip storage is impractical for real-world applications. Thus, to allow for a cost-effective implementation, instead of dedicating on-chip resources to store pointer metadata, we virtualize the pointer table [8]. In the virtualized design, the metadata table is stored in physical
memory and its entries are retrieved as needed, taking advantage of the on-chip processor cache capacity.

As expected, not all pointers are created equal. Depending on the application, several pointer links are traversed more frequently and exhibit higher performance potential for prefetching. To reduce the footprint of the metadata in the memory hierarchy and increase the effectiveness of the prefetcher, we implement a runtime feedback mechanism that focuses prefetching on delinquent pointer links. The pointer analysis used in the feedback mechanism is straightforward to implement in software, since it uses information already available at the runtime level. On the other hand, this information is hard or expensive to infer at the hardware level.

In summary, the research presented in this chapter makes the following contributions:

- We modify a widely deployed Java Virtual Machine (OpenJDK 7) to communicate object pointer information to the processor.
- We propose a new hardware prefetcher based on the JVM-supplied object pointer information, and apply the predictor virtualization technique to limit the amount of dedicated storage necessary to store object metadata.
- We analyze the distribution of useful object pointers for prefetching, and subsequently propose ways for the runtime to selectively narrow the pointer information communicated to the hardware.
- We analyze the application performance of the new prefetching scheme and show that, with limited hardware overhead (i.e., less than 1KB of dedicated storage), software runtime assisted pointer prefetching can achieve an improvement of 53% for SpecJBB 2005 when coupled with traditional prefetchers such as next-line and stride. This improvement represents an increase of 23% over the traditional prefetchers alone. Our evaluation also shows that the new proposal complements a state-of-the-art pattern-based data prefetcher [66], improving overall application performance.
Next, we present the design and implementation of the object pointer prefetcher. The experimental evaluation of the prefetching scheme is discussed in Section 6.3. Section 6.4 outlines the related work. We conclude in Section 6.5.

6.1 Software Runtime Assisted Object Pointer Prefetching

Managed programming languages, such as Java and C#, have become popular due to their ease of programming, rich libraries and automatic garbage collection. The heavy use of pointers in such languages leads to lower application performance, since pointers usually imply irregular memory accesses that are hard to predict. As a result, traditional hardware data prefetchers exhibit limited effectiveness for such applications [62, 63, 64, 65].

The difficulty in predicting access patterns of managed applications is not a fundamental property of such applications, but stems from the fact that processors are oblivious to language level constructs such as the memory layout of applications. This research proposes exposing to the hardware specific memory layout details, such as pointer locations and references, that are available at the software runtime layer. We show that a closer collaboration between the runtime and the underlying hardware can result in an effective object pointer prefetcher. Communicating object pointer information is cheap to implement in managed runtimes, since the object memory layout and the relationship between objects is already tracked to support efficient automatic garbage collection.

At a high level, our proposed software assisted pointer prefetcher consists of the following components:

1. Simple extensions to the Java runtime to communicate pointer information to the processor, including pointer creations and pointer changes due to object relocations during garbage collection.

2. An Object Metadata Table, implemented in hardware, that stores the object pointer information supplied by the software runtime. To minimize the amount of dedicated storage,
we apply virtualization to the table, effectively spilling its content to the on-chip and off-chip memory hierarchy.

3. An L2 cache hardware prefetcher that generates prefetch requests based on application memory accesses, as well as the pointer information retrieved from the Object Metadata Table.

In the remainder of this section, we describe the design and implementation of the software runtime assisted pointer prefetching, detailing each of the three components listed above. In the next section, we describe an optional pointer filtering strategy implemented in software that reduces the amount of metadata exposed to the hardware, by focusing on object pointers that are more likely to generate effective prefetch requests.

### 6.1.1 Extending the Java Runtime

To be able to perform automatic garbage collection, runtime systems keep track of all relationships between objects. To expose these relationships to the hardware, we annotate both the just-in-time compiler, as well as the garbage collector, in all contexts where links between objects are created or updated. There are two main points where links between objects are modified: 1) in the compiled code of the application, whenever an object field is attributed to another object and 2) at garbage collection time, when objects are moved in memory. We call the link created between two objects a *pointer link*, as shown in Figure 6.1. A pointer link creates a link from *Object A* to *Object B*. We refer to the field address within *Object A* that gets updated as *Slot*. We refer to the address that is written in the *Slot* as *Pointer Address* (i.e., the start address of *Object B*), as shown in the figure. Upon the creation or modification of a pointer link, the pair of addresses (*Slot, Pointer Address*) comprising the pointer link are communicated to the underlying hardware.

In our implementation, pointer links are communicated to the underlying hardware through a new instruction that contains two memory addresses (*Slot, Pointer Address*) to indicate that at
the address *Slot* a pointer to address *Pointer Address* is created/updated. The execution of the newly introduced instruction is simple: once the instruction gets decoded, the two addresses can be sent to the prefetcher and the instruction does not need to be advanced to any remaining stages in the processor pipeline. Due to the limited processing of such instructions, we expect the overhead of introducing such instructions to be limited. For the benchmarks used in this study, the newly introduced instructions account for less than 1% of the application instructions. In addition, such an instruction is advisory in nature and, if necessary, it can be discarded without any penalty on the correctness of the application.

For the Java runtime, we modified the OpenJDK 7 source code to provide the object metadata as needed. The modifications are straightforward, as all information needed is readily available in the just-in-time compiler and the garbage collector. All modifications amount to 290 lines of code across 12 files.

### 6.1.2 Object Metadata Table

The hardware prefetcher stores information provided by the runtime in an *Object Metadata Table*, as shown in Figure 6.2. The purpose of the object metadata table is to record the memory addresses of application objects, as well as their connectivity.

Conceptually, each entry in the object metadata table contains pointer information for a given object. An entry in this table can be seen as a compressed description of an object, containing all the addresses of the objects it has access to. In practice, to optimize for space and to simplify lookups, we assume that all pointers in a cache-line belong to the same object.
Thus, each entry in the table corresponds to a memory block equal in size to the L2 cache line. When the pointer link instruction is executed, the cache-line address of the Slot is used to index into the table. One of the pointers of the indexed entry is then updated with the Pointer value, overwriting the oldest pointer value in the entry (effectively implementing a FIFO replacement policy), if no entry is free.

**Virtualizing the Object Metadata Table**

To achieve the full benefits of our proposed pointer prefetcher, given that complex applications have large pointer footprints, the hardware storage for the object pointer metadata needs to be large. Dedicating on-chip storage for such a large structure is not practical. As discussed earlier, predictor virtualization mitigates the requirement of dedicated storage by emulating the large table using the conventional memory hierarchy to store application metadata.

For a cost-effective hardware implementation, we apply virtualization to the object metadata table. In the virtualized design, the table is conceptually moved to main memory, as shown in Figure 6.3. A small dedicated on-chip metadata cache holds the active entries of the table. The contents from the large, in-memory table are retrieved as needed by the prefetcher, using regular memory requests injected starting at the L2 cache level. As a consequence, the content
Figure 6.3: Virtualization is applied to the object pointer metadata of the table is partially stored in the L2 cache.

In the virtualized approach, several entries from the object metadata table are packed together and stored in one cache block in the L2 cache. Whenever metadata is required for prefetching, the dedicated cache is checked first. If the metadata is found in the dedicated cache, it is immediately used for prefetching. Otherwise, on a miss to the dedicated cache, a request is sent to the memory hierarchy to retrieve the necessary metadata. Upon the retrieval of the metadata from the memory hierarchy, the entries received are unpacked and installed in the dedicated cache. Several entries are retrieved with one memory request and all metadata entries are installed in the dedicated cache.

To track the outstanding metadata requests, a series of request status registers are used. These registers hold information about the L2 pending requests, similarly to the MSHRs (miss-status handling registers) used in regular caches. Upon a miss to the dedicated cache, the status registers are checked for a match before forwarding the request to the L2 cache. This avoids sending multiple L2 requests for the same data.

When a pointer link is modified, to be able to propagate the modification to the L2 cache, we need the contents of the rest of the corresponding L2 cache line. To store pending modifications, we include a few write buffers in our design. Each write buffer is equal in size to the L2 cache block. Experimental analysis shows that, for the benchmarks studied, 16 status registers and four write buffers are sufficient to track outstanding read/write metadata requests without dropping any updates.

A direct consequence of virtualization is the higher latency of accessing the object meta-
data. This affects the timeliness of the prefetches and, for this reason, a virtualized prefetcher generally shows lower performance than the equivalent scheme where the entire metadata table is implemented on-chip. However, the trade-off between the performance and the hardware cost is by far beneficial in this case. In Section 6.3, experimental results show that the performance of the virtualized design matches that of the non-virtualized one, while greatly reducing the on-chip dedicated storage used by the metadata table.

### 6.1.3 Pointer Prefetcher

The object pointer prefetcher injects prefetches in the second level of the on-chip cache. We chose the second level cache for two reasons. First, since first level caches are generally small, they are highly sensitive to the pollution introduced by prefetching. Second, application performance suffers mostly due to accesses that experience the full latency to main memory.

The object pointer prefetcher consults the information in the object metadata table to predict subsequent memory accesses of the application. Due to the object-oriented nature of the Java programming language, memory is traversed according to the paths described by object pointers. As such, it is possible to anticipate the objects to be accessed next based on the pointers of currently accessed objects.

Upon an application demand access to the L2 cache, the object metadata table is accessed to determine whether the address of the demand access is present in the table. If the demand address has a corresponding entry in the object metadata table, it means that the cache block being accessed contains pointers to other memory locations. These pointers are prefetched proactively such that, if accessed, the latency to retrieve the data from memory is partially or fully amortized. One demand access can generate multiple prefetches if the object metadata table records multiple pointer links for the cache block being access. When prefetches requests are filled from the memory hierarchy, the prefetched data are installed directly in the L2 cache, without any intermediate buffering.

In general, L2 caches are physically indexed and tagged, which means that they are ac-
cessed with physical memory addresses. Since the object metadata table contains virtual addresses, a virtual-to-physical translation is needed for the prefetcher. Accessing the same TLB used for translating application’s demand requests would increase the contention on this precious structure. Thus, we assume a lazily-updated shadow TLB that is used for the prefetcher alone. Since prefetches are advisory in nature and are not required for the correctness of the application, lazy updates to the shadow TLB are sufficient. The hardware overhead imposed by the shadow TLB is not included in the storage analysis. We expect the same shadow TLB to be used in different optimizations.

6.2 Class Oriented Object Metadata Filtering

Not all object pointers are created equal. Some pointer fields are traversed more often, and, as a consequence, they generate more L2 cache misses and exhibit higher potential for prefetching. One way to increase the benefit of the runtime assisted pointer prefetcher is to focus prefetching on the pointer links with higher prefetch potential. In this section, we describe one technique we experimented with, implemented in software, to filter out pointer links that are less likely to yield beneficial prefetching opportunities.

With this goal in mind, we implemented a profile analysis in the runtime system that tracks which pointer links generate the most L2 cache misses. In this analysis, we focused on understanding the behavior and classifying pointers with respect to the class each object instantiates. There are several advantages to filtering at the class level. First, class level information is easily accessible within the Java runtime, and can be used to decide which pointer links to communicate to the hardware. Second, as we show in the experimental analysis in Section 6.3.5, the distribution of delinquent pointers is biased to a small set of class fields, making it a good criteria to discard a large portion of pointer links without sacrificing prefetching opportunities. Third, the runtime overhead resulting from filtering at class-field granularity is expected to be low, since a simple general check can decide to include, or not, the new pointer link instruction.
This would not be the case for filtering specific objects or specific pointer links.

For all the classes of the application, we attempt to identify the specific class fields that contribute to the largest portion of the delinquent pointers. To do so, we extend the software runtime in the following way. Each pointer link of an object corresponds to a particular field of the class that the object instantiates. Whenever an L2 cache miss occurs, we map the miss address to a pointer link, and subsequently map the delinquent link to its corresponding class field. We increment a counter for the corresponding class field on each delinquent pointer miss. After a profiling phase, we can rank the class fields according to counter values.

When communicating the pointer links to the underlying hardware, the software runtime can choose to filter out the object metadata and communicate only the pointer links resulting from the top-most class fields in the ranking. These pointer links are predicted to be delinquent, and, in consequence, their prefetching potential is expected to be higher.

Filtering the object metadata has several consequences. First, there is less contention for storage capacity in the object metadata table. By filtering the most important pointers, we eliminate the potential overwriting of useful pointers by non-useful ones that can occur due to aliasing. Thus, filtering out non-useful pointers can lead to higher application performance. Second, less updates to the table means less accesses to the L2 cache. Third, a smaller object metadata footprint means less pollution in the L2 cache due to the virtualization of the metadata table. In Section 6.3 we show how filtering improves performance for certain benchmarks and reduces the pressure on the L2 cache.

### 6.3 Experimental Evaluation

In this section we evaluate the performance of our proposed software runtime pointer prefetcher. First, we describe the methodology used in our study, our benchmarks and the simulation infrastructure. Second, we analyze the potential of performance improvement for software assisted pointer prefetchers. Third, we present the evaluation of both dedicated and virtualized
pointer prefetchers. Motivated by our analysis on distribution of useful pointers, we propose a class profiling technique that focuses the prefetching to a limited number of class pointer fields. We compare this approach with an idealized filtering and study the performance improvement and the impact on the memory hierarchy. In our evaluation, we also show that the new proposal complements a state-of-the-art pattern-based data prefetcher [66, 67].

6.3.1 Experimental Methodology

In our experiments, we used our own simulation infrastructure based on PIN, a dynamic binary instrumentation framework [68]. For each benchmark, we collect a trace with all the information necessary to run our simulations. We use this trace to drive an in-order processor model in which each instruction, except for memory instructions, takes one cycle to execute. The latency of memory instructions is determined by simulating a memory hierarchy with a 32KB first level data cache and a 4MB second level cache. The latency of the first level is one cycle, the latency of the second level is 12 cycles and the latency of the main memory is 400 cycles. The cache block size used in all caches is of 64 bytes.

We used two microbenchmarks from the Java implementation of the Olden benchmark suite: em3d and mst. For em3d, we present results for two different set of inputs, since they exhibit different behavior. We chose these benchmarks from the JOlden suite since they exhibit sufficient memory pressure with complex access patterns that can not be entirely covered by stride prefetchers. We excluded health benchmark from our study based on the observation made by Craig Zilles [69].

In addition, we include results for a benchmark that was recently used to analyze the performance and the features of different programming languages [61]. The benchmark is an implementation of an algorithm that analyzes code to recognize reducible and irreducible loops, described by Havlak [70]. The benchmark builds a large control flow graph containing 4-deep

---

1The full benchmark sources are available as open-source, hosted at http://code.google.com/p/ in the project multi-language-bench.
Benchmark | Input Parameters
---|---
em3d-A | 10,000 nodes, degree 4
em3d-B | 20,000 nodes, degree 5
mst | 1,024 nodes
loop | default (76000 loops, 50 iterations)
specjbb | warehouse_population = 2,000
| jbb_transactions=100,000

Table 6.1: Input parameters used for running the benchmarks

loop nests with a total of 76000 loops. It performs loop recognition on this graph for 50 times. We used the optimized Java implementation for this benchmark. In the text and in the graphs, we refer to this benchmark as loop. The last benchmark used in our study is SpecJBB2005 modified to execute a fixed number of transactions, as it is used in several Java studies under the name of pseudo-SpecJBB. The main input parameters for the benchmarks used in the evaluation are specified in Table 6.1.

All traces were collected using our modified OpenJDK 7 JVM, using the x86 32-bit backend. To avoid the interference of the compilation activity with our measurements of application time, we forced OpenJDK to just-in-time compile all code with the optimizing compiler on first encounter of the code. We verified by running the benchmarks on a real machine that the Java runtime produces efficient code when run with this option (within 10-15% of the code produced when the just-in-time compiler uses feedback directed optimizations).

For all benchmarks, several iterations of the benchmark are run before any timing measurement is performed to ensure that all code is compiled and the Java runtime reaches a steady state. For all benchmarks, we choose to collect time measurements for the fifth iteration of the benchmark. Based on our experiments on a real machine, the performance stabilizes by the fifth iteration. For all benchmarks except SpecJBB, we report performance numbers for a complete run of one benchmark iteration. For SpecJBB, we report performance numbers for running the first four billion instructions in the fifth benchmark iteration.

In all experiments, we enable two additional orthogonal, simple hardware prefetchers that have been widely deployed: stride and next-line prefetchers. For the stride prefetcher we use a
table with 128 entries to identify striding patterns in the memory access stream [71]. Next-line prefetching is triggered on misses at the second level cache. All prefetches are directly inserted in the L2 cache, without intermediate buffering.

In all experiments, instruction fetch is idealized (i.e., instruction fetch time is not modeled). To verify the impact of inserting new instruction for our hybrid prefetcher, we count the number of new instructions for each benchmark iteration. Across all benchmark studied, the new instructions account for less than 1% of the application instructions. The exact numbers are depicted in Table 6.2.

### 6.3.2 The potential of software runtime assisted pointer prefetching

In this section, we aim to quantify the potential of software runtime assisted object pointer prefetching to improve application performance. For this set of experiments, the Java runtime communicates all object pointer links upon their creation or modification. We also assume that the Object Metadata Table is unbounded in size. While this is an impractical assumption, our goal is to quantify the potential of the proposed prefetching scheme (we discuss a realistic implementation in the next section).

Whenever an L2 cache block is accessed, for all slots (i.e., in our 32-bit system with 64-byte cache lines, there are 16 blocks of 4 bytes each), the unbounded table is searched for any Slot address. If such a Slot exists, a prefetch for the cache block corresponding to the address Pointer Address is generated, provided this address is not already present in the L2 cache. In a variation of this prefetching scheme, we generate prefetches for up to two cache lines: the one containing the address Pointer Address and the subsequent cache line in memory.

Figure 6.4 shows the percentage performance improvement relative to a baseline with no prefetching for three different prefetching techniques: 1) NL-Stride: next-line and stride

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>em3d-A</th>
<th>em3d-B</th>
<th>mst</th>
<th>loop</th>
<th>specjbb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage New Instructions</td>
<td>0.74%</td>
<td>0.70%</td>
<td>0.72%</td>
<td>0.91%</td>
<td>0.19%</td>
</tr>
</tbody>
</table>

Table 6.2: Percentage of new instructions introduced by the hybrid prefetcher

1) NL-Stride: next-line and stride
prefetching running together, 2) \textit{NL-Stride-Unbound}: the unbound pointer prefetching technique when run in conjunction with the next-line and stride prefetching, and 3) \textit{NL-Stride-Unbound2L}: the same as the previous scheme, but with up to two cache lines prefetched as explained above.

As expected, traditional prefetching schemes improve the performance of memory intensive applications. These schemes work for regular access patterns and also for object pointers that happen to be aligned in memory at constant strides. However, a runtime assisted pointer prefetcher adds considerably to the performance improvement. As is shown in the graph, the potential performance for such a scheme is significant across all applications, with a 70\% performance improvement for SpecJBB. Across all applications, prefetching the subsequent line for an object pointer is beneficial, therefore, for the rest of the experiments presented in the chapter, prefetching is always performed for two cache lines.

While these results are promising, they are obtained with an unrealistic hardware implementation that assumes unbounded resources to store the object metadata. In the next section, we study the performance obtained under realistic conditions, limiting the resources available for storing the metadata.
6.3.3 The performance of a practical pointer prefetchers

In this section, we show that even a large and fast dedicated table is not sufficient to capture the full potential of the prefetcher. To address this issue, we virtualize the object metadata table as explained in Section 6.1.2 and compare its performance with the dedicated approach.

For the first set of experiments, the object metadata table is limited to a fixed size. Each entry in the table corresponds to a 64-byte memory block and can store up to four pointer addresses (i.e., up to four pointer addresses that exist in the 64-byte memory block). The pointer information is stored using a FIFO replacement policy. The table is two-way set associative, which performs slightly better than a direct mapped table for the real benchmarks (the microbenchmarks are insensitive to this parameter).

To break the limitations of the on-chip resources, we virtualize the object metadata table. Virtualization allows for increased storage capacity without the cost of dedicated on-chip resources. The drawback of virtualization is the increased pressure on the memory hierarchy and an increase in the prefetch latency.

In Figure 6.5 we present the performance improvement over a baseline with no prefetching for the following schemes: 1) NL-Stride: next-line and stride prefetching, 2) NL-Stride-Small-Dedicated: next-line and stride prefetching run in conjunction with a pointer prefetcher with a dedicated table with 256K sets, 3) NL-Stride-NoFiltering-Dedicated: same as the previous scheme, but, in this case, the table has 2M sets (i.e., \(2^{20}\) sets), 4) NL-Stride-NoFiltering-Virtual: the table in the previous scheme is virtualized, and 5) NL-Stride-Unbound2L: the
unbound version of the pointer prefetcher. The *NL-Stride* and *NL-Stride-Unbound2L* are the same as in the previous graph and are included in this section for comparison purposes.

For the microbenchmarks that have smaller pointer footprints, the small table still provides considerable performance improvement on top of the traditional prefetchers. However, on the benchmarks with larger pointer footprints, the smaller table is insufficient to achieve the true performance potential. On a different note, although smaller in size, the dedicated table occupies considerable on-chip resources, which may not be available in a real-world processor design. Virtualization represents a viable solution for such hardware optimizations. To reduce the dedicated resources, we apply virtualization for a large pointer metadata table. For reference, in Figure 6.5, we present the results for a large dedicated table and the corresponding virtualized design (*NL-Stride-NoFiltering-Dedicated* and *NL-Stride-NoFiltering-Virtual*). Due to the large sizes of the tables in the dedicated schemes, their access time should be multiple cycles. However, in our experiments, we optimistically assume a one-cycle latency for these tables which emphasizes the penalty of virtualization.

Increasing the size of the table, while not fully achieving all the performance possible, allows pointer prefetching to come close to the performance of the unbounded table. However, the size of the table is prohibitively high to be implemented on-chip. Applying virtualization loses some of the performance since some of the cache space now is allocated to store object metadata and the timeliness of the prefetches is affected. However, the trade-off between performance and dedicated space is highly beneficial in this case.

For one of the microbenchmarks, *em3d-B*, virtualization shows visibly less performance than the dedicated table. In this case, the object metadata causes considerable pollution on the L2 cache. For *em3d-B*, virtualizing a smaller table can achieve higher performance. Tuning the size of the virtual table depending on the application is possible and deferred for future work.

To understand the benefits of virtualization, we discuss the storage required for each of the schemes presented. For all schemes, the tables are tagged with the remaining bits after indexing is performed. Each entry in the table has sufficient space to store metadata for up to
four pointers. Each pointer metadata represents a memory address aligned to a memory block of 128 bytes (i.e., two 64-byte cache lines). Table 6.3 shows the amount of storage required for each of the pointer prefetching schemes. The storage for the virtualized approach includes the resources needed for the status registers and write buffers. The trade-off between dedicated space and performance makes the virtualized pointer prefetcher a promising candidate.

As discussed in Section 6.2, not all pointer links contribute equally to the performance improvement. The results presented so far did not make any attempt at filtering the object metadata stored for prefetching. In the next sections, we experiment with filtering of object metadata to achieve higher performance.

### 6.3.4 The potential of object metadata filtering

In the schemes discussed so far, the runtime communicates all object pointers to the processor. However, not all prefetches generated by our prefetcher are useful. In fact, since the capacity of the table is smaller than the pointer footprint of the applications, it is possible that useful pointer metadata is overwritten, due to aliasing, by metadata that does not contribute to performance. In this section, we aim to quantify how much additional performance object metadata filtering could provide, using hindsight information.

To identify which pointer links to filter out, we ran the unbound scheme and mark which pointers are prefetched and used by the application before being replaced and removed from the L2 cache. We call these pointers *useful pointers*. To understand the potential of metadata filtering, we run the virtualized prefetcher described in the previous section, but only useful pointers are communicated to the hardware and stored. Note that the useful pointer footprint

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Dedicated storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL-Stride-Small-Dedicated</td>
<td>6.75 MB</td>
</tr>
<tr>
<td>NL-Stride-NoFiltering-Dedicated</td>
<td>52 MB</td>
</tr>
<tr>
<td>NL-Stride-NoFiltering-Virtual</td>
<td>879.5 bytes</td>
</tr>
</tbody>
</table>

Table 6.3: Dedicated on-chip storage requirements for the specialized object-pointer prefetchers
is still larger than the capacity of the virtual table, and, in consequence, some entries can be overwritten.

Figure 6.6 shows the performance improvement of filtering useful pointers (i.e., the bar labeled NL-Stride-Useful-Virtual) compared to schemes presented previously. For the em3d microbenchmarks, the capacity of the virtual table is sufficient to store most of the application pointers and, as a consequence, filtering ends up reducing the potential in performance. However, for the rest of the benchmarks, that have higher pointer footprints, filtering object metadata visibly increases the performance potential. For example, for the loop benchmark, the performance improvement increases from 51% to 68%. For specjbb, the filtering brings the performance within 6% of the idealized unbound prefetching scheme.

While the useful-pointer filtering presented in this section shows performance improvement, the filtering is performed with information not available at runtime. In the next section we analyze the distribution of useful pointers that inspired the practical filtering scheme, based on class fields, presented in Section 6.2.

### 6.3.5 Distribution of useful pointer fields

Due to the access patterns of data structures in programs, certain object pointers are more frequently traversed than others. For example, in the mst benchmark, the pointer fields that link the buckets within a hash-table are traversed more often than the object pointers that refer to
the data stored in each element in the hash-table. In this section, we experiment with filtering object metadata based on class object pointers. To better understand the potential of such a filtering technique, we analyze the distribution of useful pointer fields. We are interested in studying the percentage of unique class fields that contribute to most of the useful prefetches. To collect these statistics, we mark the pointers that are useful using the unbound scheme. Once all pointer links are collected, we identify the unique class fields that generate at least 100 useful pointer prefetches.

Figure 6.7 shows the cumulative distribution of unique class fields that contribute to the useful prefetches. These results show that a fraction of the pointer fields contribute to most of the useful prefetches. The most notable example is mst for which only two pointers contribute to more than 90% of the useful pointers. Even for the more complex benchmarks loop and specjbb, less than a quarter of class fields contribute to more than 80% of the useful prefetches.
For the microbenchmark *em3d*, runs *A* and *B* look similar, with most of the pointers contributing to the useful prefetches, and we therefore omit the graph for *em3d-B*. Since the pointer footprints are small for these microbenchmarks, the virtual table can accommodate all object metadata. For the rest of the chapter, we exclude these microbenchmarks from our analysis, to focus on the benchmarks with more interesting pointer footprints.

These statistics indicate that filtering the object metadata to focus prefetching on a subset of the pointer fields can potentially achieve higher performance. Reducing the amount of metadata to be stored in the table will reduce aliasing, increasing the efficiency of the table. In addition, the pressure of the virtual table on the cache capacity may decrease, since it may store less information. In the next section we experiment with object metadata filtering to better understand its benefits.

### 6.3.6 Class oriented object metadata filtering

In this section we experiment with applying class oriented object metadata filtering to our pointer prefetcher. Using the statistics collected in the previous section, we focus prefetching on the most beneficial class pointer fields by communicating to the hardware only the pointer links that correspond to the class fields with higher contributions to the useful prefetches. While this type of filtering is not practical, we include it as a comparison for our runtime filtering scheme.

One straightforward way of filtering object metadata is focusing on the delinquent pointer links. In this filtering, we note which pointer field traversals lead to misses in the L2 cache. As misses are observed, they are attributed to the corresponding class fields that lead to that memory access. This type of runtime profiling is similar to the one presented by Adl-Tabatabai et al. [62]. Based on this profiling, we rank the class pointer fields and focus prefetching only for the pointer links corresponding from the top-most class fields.

Figure 6.8 shows the performance results for three virtual prefetching schemes: no filtering, filtering based on runtime delinquent pointer fields and filtering based on useful prefetches.
CHAPTER 6. A VIRTUALIZED OBJECT POINTER PREFETCHER

Figure 6.8: Performance improvement with pointer link filtering

using hindsight information (see Section 6.3.4). Filtering shows the most performance benefits for loop, with an increase in performance of 6% and 8% for the two filtering techniques over no filtering.

Filtering object metadata also results in less pressure on the virtual table. This has implications on the amount of metadata stored and retrieved from the memory hierarchy. To better understand the impact of filtering on the memory hierarchy, we next analyze the memory traffic caused by the virtual prefetching scheme.

Table 6.4 shows several statistics that characterizes the traffic to the L2 cache as a percentage difference between filtering and no filtering. For all metrics presented, there is a slight decrease for the scheme that employs filtering. The decrease is more pronounced for writebacks, both application data and object metadata writebacks. With filtering, the footprint of the metadata in the cache is smaller, which leads to less cache replacements, and, as a consequence, less application writebacks. In addition, filtering implies less updates to the virtual table, which means less object metadata writebacks.

The last row in the table captures the maximum cache capacity used to store metadata. The table entry shows the maximum number of cache lines that store metadata that are present at any time in the L2 cache. For the more involved benchmarks, the metadata reaches almost a third of the cache capacity. This is a reinforcement of the fact that locality is poor for these
Table 6.4: L2 cache statistics differences for the filtering scheme when compared to no filtering applications and the cache storage is better utilized to store object metadata that helps with prefetching than to use the same space for traditional application data caching.

An important benefit of delegating metadata collection to the software runtime layer, as opposed to implementing it fully in hardware, is the added flexibility at a low cost. The simple profiling scheme presented in this section exemplifies this benefit. With no changes to our proposed hardware, or the interface, the software runtime is able to leverage various sources of information about its memory layout and specific application access pattern, to provide more accurate or relevant information to the hardware pointer prefetcher.

One concern with virtualization in place is the additional off-chip bandwidth used by the application metadata traffic. To better understand the bandwidth overhead, Figure 6.9 plots off-chip bandwidth measurements for a scheme without virtualization (*NL-Stride*) and a virtualized prefetcher that employs filtering (*NL-Stride-Filtering-Virtual*). The bandwidth is measured in accesses per 1K instructions and it is split in several components: misses, writebacks, prefetches, metadata misses and metadata writebacks. As can be seen from the figure, the bandwidth is dominated by the prefetch traffic, while the traffic due to virtualization represents a small portion of the total traffic. Even with the additional traffic, the bandwidth requirements are low, with a maximum of 26 accesses per 1K instructions.
6.3.7 Object Pointer Prefetching and State-of-the-Art Prefetching

So far, we have used and compared the object pointer prefetcher with next-line and stride prefetchers. This was motivated by the fact that these two prefetchers are already deployed in virtually all current processors. In this section, we strive to understand how the object pointer prefetcher performs when run in conjunction with a state-of-the-art prefetcher.

Due to the performance penalties of long memory latencies, data prefetching is a well-researched topic with a plethora of proposals. For the purpose of this section, we chose to use the winner from the first data prefetching championship [66]. In the text, we refer to this scheme as DPC (data prefetching champion). The details of the DPC prefetching scheme are described in Section 6.4.2.

We ported the implementation of the DPC to our simulation infrastructure and performed experiments running the DPC by itself and in conjunction with our object pointer prefetcher. For the experiments included in this section, we used two different configurations for the DPC: the original proposal (i.e., 4KB of dedicated space) and a reduced-storage design with only 1KB of dedicated storage. Figure 6.10 shows the results for the DPC when run without and with the object pointer prefetcher. The results vary across the benchmarks used in this study. For em3d-B and loop, there is a negative interference between the DPC and the object pointer prefetcher, leading to a slight slowdown when the two prefetchers are run together. However, for em3d-A, mst and spec-jbb, the object pointer prefetcher adds to the application performance
obtained with the DPC in both configurations used. In particular, a DPC-1KB with the object pointer prefetching on top performs better than the DPC-4KB with a larger dedicated storage. This shows that using precise object pointer information in prefetching exposes a performance potential that is difficult to be covered by pattern-based prefetchers alone.

The results in this section indicate that the object pointer prefetcher has the ability to perform well when run with other type of prefetchers. We believe that co-designing the prefetchers together could lead to even higher performance.

6.4 Related Work

6.4.1 Prefetching Linked Data Structures

Prefetching linked data structures is a well-researched topic with various proposals [6, 34, 62, 64, 72, 73, 74, 75, 76, 77, 78]. In this section, we outline the schemes that are most similar to our proposed prefetcher.

Content-directed data prefetching uses a simple heuristic to determine which slots from each cache line look like virtual addresses and generates prefetches for these addresses [34]. One major issue with this approach is the timeliness of the prefetches. To be able to identify which slots contain pointers, the data needs to be brought from the main memory before it
is inspected. This way, the distance between the prefetch and the actual demand request is reduced to the point that most prefetches are ineffective. To address this issue, the pointer cache technique stores the identified pointers in a dedicated on-chip structure [6]. This technique requires large dedicated resources to be effective and relies on heuristics to determine pointer links. In our approach, the pointer links are communicated by the runtime system and no heuristic is needed. In addition, virtualizing the table that stores the object metadata eliminates the requirements of large, dedicated resources. Ebrahimi et al. use compiler hints to improve the precision of the content-directed data prefetching [72]. The compiler hints rely on a static object layout, which is generally not the case for managed languages. Ebrahimi et al. also propose heuristics to ensure a better collaboration between the traditional prefetchers and the pointer prefetcher. Their scheme is orthogonal to our approach and can be used in conjunction for a better utilization of the prefetch bandwidth and cache capacity.

Several prefetching techniques for linked data structures rely on compiler and/or programmer support [65, 75, 76, 78]. In contrast, our scheme is transparent to the programmer and uses information readily available to the runtime system.

Address-correlating prefetchers indirectly address pointer accesses [2, 79]. In the presence of garbage collection, that restructures the object memory layout, address correlation becomes more difficult.

The research most similar, in spirit, to our proposal is the software prefetching technique introduced by Adl-Tabatabai et al., which identifies delinquent pointer links in Java applications and generates prefetches for these links [62]. The main difference between the two proposals is how the prefetches are generated. Adl-Tabatabai et al. rely on the ability of the runtime system to reposition the objects in memory and try to preserve the distances between objects. The prefetches are generated in software using the constant distances between objects. In our approach, the software layer communicates the relationships between objects to the hardware, which manipulates this metadata and generates pointer prefetches. Our scheme does not depend on any memory object layout and does not require repositioning of objects.
in memory. For applications that have complex object layouts, it is hard to preserve distances between certain objects. Delegating the responsibility of prefetching to the hardware alleviates any requirement on the object position, which simplifies the modifications in the runtime system.

6.4.2 DPC: Data Prefetching Champion

In this section, we describe the details of the DPC prefetching scheme used in the experiments presented in Section 6.3.7. DPC won the Journal of Instruction-Level Parallelism Data Prefetching Championship in 2009 [66].

DPC uses memory access footprints to extract patterns in the application access stream and generates prefetches for subsequent memory addresses in the pattern. Similarly to spatial memory streaming [2], DPC splits the memory space into fixed-size regions called zones. The zones most recently accessed by the application are called hot zones. Application accesses within hot zones are recorded at cache-line granularity.

Upon a demand access to address $A$ within a zone $Z$, the footprints for zones $Z-1$, $Z$, $Z+1$ are searched for a striding pattern with stride $K$, for all possible values of $K$ within the memory region under consideration. If such a pattern is discovered, prefetches to addresses $A+K$ and $A-K$ are generated. If several strides $K$ are discovered, the prefetcher prioritizes for smaller values of $K$.

The key component used by the DPC is a table that stores zone footprints for the most recently accessed zones. The size of the table is fixed, and replacement of hot zones in the table is performed following an LRU policy. Each entry in the table is associated with a zone and stores a memory access map for the corresponding zone.

The memory access map records information at cache-line granularity about the accesses within a zone. Two bits are used for each cache-line within a zone to record one of the following four states: init, access, prefetch, success. All cache-lines within a zone start in the state init. Prefetched lines are marked as prefetch. Upon a demand access to a cache line in state init,
the two-bit state is marked as *access*. If the demand access requests a previously prefetched cache-line, the corresponding state is promoted to the state *success*.

Upon a demand access to a zone, the current zone together with the preceding and the subsequent zone in memory are searched for striding patterns. Only states *access* and *success* are used to determine stride patterns within the adjacent zones. Both states indicate that the application demanded the corresponding cache-line.

The implementation used in the data prefetching championship employs zones of 16KB. For a 64-byte cache line size, each access map records 256 entries with two bits per entry for encoding the state of the cache line. The table stores up to 52 zones for a configuration with a total budget of 4KB and up to 13 zones for a configuration with a budget of 1KB. While the total budget accounts for storage required by MSHRs and other control counters, the bulk of the storage is consumed by the access map table.

DPC claims that its performance benefit stems from extracting striding patterns across large memory regions without considering the order in which the memory accesses occur, nor the addresses of the instructions that generate the accesses.

### 6.5 Concluding Remarks

In this chapter, we presented a software-hardware collaborative scheme for software runtime assisted object pointer prefetching. Managed runtimes record rich metadata on object memory layout that is not available to the underlying hardware. Through a thin interface, we communicate the links between objects to the hardware. Using this information, the hardware generates pointer prefetches based on the memory access stream of the application. To achieve a cost-effective hardware implementation, the object metadata is spilled to the memory hierarchy and retrieved only when needed. This alleviates the need for large on-chip structures and makes the prefetching scheme practical.

We experimented with object metadata filtering based on delinquent pointer links. We
analyze the pointer links that lead to most misses observed by the L2 cache and focus the prefetching only on the class pointer fields that generate the most misses. This filtering reduces the amount of object metadata stored in the hardware and results in slightly higher application performance with less traffic to the L2 cache and main memory. The analysis of the delinquent pointer links is performed at the runtime level, with information that is readily available. Such an analysis is impractical at the hardware level since the underlying hardware is oblivious to the object memory layout or relationships between objects.

Our experimental results show significant performance improvements for a series of microbenchmarks and real benchmarks. More notably, when applied in conjunction with traditional prefetching schemes, such as next-line and stride, the software runtime assisted object prefetching obtains a 53% performance improvement, which represents a 23% improvement over traditional prefetching alone. Our evaluation also shows that the new proposal complements a state-of-the-art pattern-based data prefetcher, improving overall application performance. While our case study used the Java runtime system, we did not rely on Java specific features and, in consequence, the ideas described in this chapter have wide applicability to any runtime system with automatic memory management and garbage collection.
Chapter 7

Concluding Remarks

Processing systems have reached a point where they cannot sustain sheer performance by technology scaling alone. Thus, to maximize performance, innovative techniques are required to better utilize the available on-chip resources. Traditionally, processor designers employ a one-size-fits-all approach when building predictor-based hardware optimizations: each optimization has a fixed portion of the on-chip resources allocated to the predictor tables. Thus, one important challenge is to decide which hardware optimizations to include and how much resources to be dedicated to a specific optimization. This trade-off often leads to sub-optimal designs where: 1) resources are wasted for applications that do not benefit from a particular predictor or only require small predictor tables, or 2) predictors under-perform for applications that need larger tables that can not be built due to area-latency-power constraints.

This thesis introduced Predictor Virtualization (PV), a technique that breaks the limitations of the traditional “one-size-fits-all” design for hardware predictors. PV exploits the current trend of unprecedentedly large on-chip secondary caches to emulate large, flexible prediction tables for hardware optimizations. PV is a timely proposal, taking advantage of on-chip caches, which have become sufficiently large to accommodate allocating, on demand, a small percentage of their capacity for caching predictor metadata. Virtualizing existing predictors reduces the cost of dedicated on-chip resources, while maintaining the effectiveness of the predictor.
More importantly, PV breaks the limitations of conventional designs and allows the implementation of otherwise impractical predictors.

In this chapter, we present the summary of the thesis and the research contributions, followed by directions for future research.

7.1 Thesis Summary

Predictor virtualization relies on the existing memory hierarchy to emulate large hardware predictors without dedicating expensive on-chip resources. Instead of using fixed, large, dedicated tables, PV assigns a small portion of the on-chip real estate to implement a cache for a large, virtual predictor table that is stored in physical memory address space. The predictor entries from the virtual table are brought into the dedicated on-chip table as needed, using regular memory requests injected in the memory hierarchy. As a natural consequence, PV dynamically shares the capacity of on-chip caches with the application data and instructions.

Unlike traditional approaches, PV avoids wasting resources by allocating only those portions of the virtual table that are actively accessed. The allocation of metadata in the caches is performed on demand, as the application requires more predictor resources. In addition, stale predictor metadata is silently replaced in the on-chip caches as a natural consequence of cache replacement policies. Thus, in contrast with the current “one-size-fits-all” approach, PV represents a pay-as-you-go alternative for designing hardware predictors.

Implementing predictor virtualization techniques in real silicon is tractable. PV does not require intrusive design modifications to the existing memory hierarchy. The only important addition to current designs is the ability of the PV components to communicate with the memory hierarchy starting at the lower cache levels. Arbitration is already supported at this level, and adding a new component should be a straightforward extension. Thus, we believe PV is a viable design option for hardware predictors.

To demonstrate the benefits of virtualizing hardware predictors, we applied the technique to
three different hardware optimizations: a state-of-the-art data prefetcher, conventional branch target buffers and object pointer prefetchers. While these hardware optimizations exhibit different characteristics which lead to different virtualized design, they all benefit from virtualization. Specifically, the contributions of this thesis are as follows:

- This thesis introduces a novel technique, *Predictor Virtualization (PV)*, to emulate large predictor tables used in hardware optimizations without dedicating precious on-chip resources. PV leverages the conventional memory hierarchy to implement cost-effective hardware predictors. The thesis describes the various advantages of virtualized predictors over the traditional ones.

- This thesis demonstrates the benefits of virtualization when applied to a state-of-the-art data prefetcher, the spatial memory streaming (SMS) [2]. The virtualized SMS design matches the performance of the original scheme within 1% on average, while reducing the dedicated on-chip predictor table from 60 kilobytes to less than one kilobyte per each core.

- This thesis introduces a virtualized design for conventional branch target buffers (BTB) [15] that employs metadata prefetching to increase the perceived capacity of the BTB without dedicated extra on-chip resources. The application performance obtained with the virtualized design is similar, on average, to the performance obtained with a larger BTB, that utilizes 3.6 times more on-chip storage. Experimental results show that virtualization remains effective with larger dedicated BTBs, as long as the applications benefit from additional BTB resources.

- This thesis proposes an object pointer prefetcher for managed languages. The novelty of the prefetcher is two-fold. First, the runtime system is responsible for collecting and communicating application metadata to the underlying hardware. Second, the hardware employs virtualization to implement a cost-effective object pointer prefetcher. The virtualized prefetcher show promising results. For example, when run with next line and
striding prefetchers, the pointer prefetching scheme achieves 53% performance improvement for SpecJBB2005 compared to no prefetching, which represents an increase by 23% compared to traditional prefetching alone.

7.2 Future Work

This thesis introduces a novel technique for virtualizing hardware predictors and shows its benefits with three different case studies. Predictor virtualization can be viewed as a framework that enables several directions of research. In this section, we briefly discuss possible research projects that are enabled by virtualizing predictors.

Power and Energy Studies for Virtualized Predictors

In this thesis, we focused our analysis of predictors on the hardware-cost/performace trade-off. As power and energy dissipation are prominent constraints in processor design [80], one important future project is the analysis of the performance/power/hardware-cost trade-off in the context of virtualized predictors. Several inter-related factors resulting from virtualization should be considered in the study, such as:

- PV reduces the size of the tables used in hardware optimizations, which results in lower leakage power. As technology scaling continues, leakage power is predicted to become the predominant factor in total power [81].

- Virtualization introduces new accesses to the memory hierarchy, which increases the dynamic power consumed by the on-chip caches.

- For same size dedicated tables, PV increases the accuracy of the prediction used in hardware optimizations compared to traditional designs. Higher prediction accuracy has two important effects. First, improved speculation reduces the amount of useless work,
which, in return, lowers the dynamic power. Second, more accurate predictors often imply faster application execution, which may result in lower energy consumption.

**Adaptive Virtual Table Sizes**

In all case studies presented in this thesis, the virtual table configuration (e.g., size, associativity, entry content) is fixed at design time. Usually, we chose the largest size that proved to perform well across the benchmarks used in our study. This tends to work since, inherently, only the portions of the virtual table that are active are actually allocated in the on-chip caches.

PV can benefit further from adapting the virtual table size and configuration at runtime, based on the observed application behavior. In particular, if the application exhibits a lot of conflict misses and entries in the virtual table are overwritten, a large table may perform better, by allowing more metadata to be utilized in the prediction. In addition to the feedback loop for adapting the size of the virtual table, a programmable indexing scheme is required for allocating and retrieving application metadata. This addition may be justified by the increased flexibility and performance.

**Metadata Sharing Across Multi-cores**

Some application metadata can be recreated easily, while certain metadata can be expensive to recreate. For example, memory access patterns of complex application may require long runs to be recreated entirely. Same applies to pointer links between objects that can be recreated easily only at garbage collection time. For this type of application metadata, sharing information across cores in a chip multi-processor may be beneficial.

Since the virtual table is partially stored in the memory hierarchy, virtualization can naturally take advantage of the shared on-chip caches to share metadata across cores. Thus, the patterns learned on one core can be utilized on a different core without any additional learning required. This sharing can be enabled by utilizing only one virtual table across all cores. Retrieving entries from the common virtual table ensures metadata propagation from one core to
the other.

Optimizations are speculative in nature, and, thus, coherence for the virtual table is not mandatory. However, if the prediction accuracy highly depends on up-to-date metadata, the virtual table needs to be coherent to ensure proper delivery of metadata modifications across multiple cores. As the virtual table resides in the on-chip cache hierarchy, piggy-backing on the cache coherence mechanism should be straightforward.

We expect metadata sharing to be beneficial for predictors where the application metadata is time-consuming to learn or create. In particular, prefetchers (e.g., data, instruction or TLB prefetchers) are a promising candidate for such a technique.

**Virtualizing Different Predictors and Processor Components**

Virtualization allows for extra resources to be allocated on demand, depending on the behavior of the application run by the processor. This feature can enable adaptability in current predictors or the design of new predictors that do not prove beneficial with constrained resources.

In this thesis, virtualization was applied to different hardware predictors. However, virtualization can be applied to other processor components. One component that could benefit from virtualization is the TLB (Translation Look-aside Buffer). TLBs are currently used to translate virtual memory addresses into physical ones. As the application footprints grow, current TLBs show lower performance due to the high demands on their capacity [82].

TLBs could benefit from virtualization in several ways. First, virtualization can be used to implement TLB prefetchers. We expect temporal correlation for TLB accesses that repeats over long sequences of memory accesses. Virtualization could allow capturing and replaying these sequences similarly to the temporal instruction streaming [31]. Virtualization allows for longer memory access sequences to be recorded, which we expect to be beneficial for TLB prefetching. Second, the TLBs themselves can be virtualized to offer additional capacity when needed. The main difference between applying virtualization to TLBs compared to other hardware predictors is that the TLB content is needed for the correctness of the application.
Thus, the content of the dedicated TLBs and the virtualized storage need to be coherent to allow for correct application execution. Since TLB updates are not frequent, keeping TLB entries up-to-date in a virtualized design will not be difficult.

Other examples of predictors that can benefit from virtualization are: indirect branch predictors, instruction scheduling for SMT processors, and instruction prefetchers. A couple of different research groups already employed virtualization to study the cost-performance trade-off in instruction prefetchers [31] and branch direction predictors [32].

The fundamental novelty introduced by PV is the adaptability of a central component in modern processors: the predictor table used in speculative hardware optimizations. As processors continue to be used for a variety of applications, we believe that adapting to the application’s requirements for predictor resources is crucial. This adaptability allows for more flexible designs, that allocate resources depending on the application’s particular needs, as opposed to average trends across applications.
Bibliography


