A Primer on Hardware Prefetching
Synthesis Lectures on Computer Architecture

Editor

Margaret Martonosi, Princeton University
Founding Editor Emeritus
Mark D. Hill, University of Wisconsin, Madison

Synthesis Lectures on Computer Architecture publishes 50- to 100-page publications on topics pertaining to the science and art of designing, analyzing, selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals. The scope will largely follow the purview of premier computer architecture conferences, such as ISCA, HPCA, MICRO, and ASPLOS.

A Primer on Hardware Prefetching
Babak Falsafi and Thomas F. Wenisch
June 2014

On-Chip Photonic Interconnects: A Computer Architect's Perspective
Christopher J. Nitta, Matthew K. Farrens, Venkatesh Akella
October 2013

Optimization and Mathematical Modeling in Computer Architecture
Tony Nowatzki, Michael Ferris, Karthikeyan Sankaralingam, Cristian Estan, Nilay Vaish, David Wood
September 2013

Security Basics for Computer Architects
Ruby B. Lee
September 2013

The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition
Luiz André Barroso, Jimmy Clidaras, Urs Hölzle
July 2013
Shared-Memory Synchronization
Michael L. Scott
June 2013

Resilient Architecture Design for Voltage Variation
Vijay Janapa Reddi, Meeta Sharma Gupta
June 2013

Multithreading Architecture
Mario Nemirovsky, Dean M. Tullsen
January 2013

Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)
Hyesoon Kim, Richard Vuduc, Sara Baghsorkhi, Jee Choi, Wen-mei Hwu
November 2012

Automatic Parallelization: An Overview of Fundamental Compiler Techniques
Samuel P. Midkiff
January 2012

Phase Change Memory: From Devices to Systems
Moinuddin K. Qureshi, Sudhanva Gurumurthi, Bipin Rajendran
November 2011

Multi-Core Cache Hierarchies
Rajeev Balasubramonian, Norman P. Jouppi, Naveen Muralimanohar
November 2011

A Primer on Memory Consistency and Cache Coherence
Daniel J. Sorin, Mark D. Hill, David A. Wood
November 2011

Dynamic Binary Modification: Tools, Techniques, and Applications
Kim Hazelwood
March 2011

Quantum Computing for Computer Architects, Second Edition
Tzvetan S. Metodi, Arvin I. Faruque, Frederic T. Chong
March 2011

High Performance Datacenter Networks: Architectures, Algorithms, and Opportunities
Dennis Abts, John Kim
March 2011
A Primer on Hardware Prefetching

Babak Falsafi
EPFL
Thomas F. Wenisch
University of Michigan

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #28
ABSTRACT
Since the 1970’s, microprocessor-based digital platforms have been riding Moore’s law, allowing for doubling of density for the same area roughly every two years. However, whereas microprocessor fabrication has focused on increasing instruction execution rate, memory fabrication technologies have focused primarily on an increase in capacity with negligible increase in speed. This divergent trend in performance between the processors and memory has led to a phenomenon referred to as the “Memory Wall.”

To overcome the memory wall, designers have resorted to a hierarchy of cache memory levels, which rely on the principal of memory access locality to reduce the observed memory access time and the performance gap between processors and memory. Unfortunately, important workload classes exhibit adverse memory access patterns that baffle the simple policies built into modern cache hierarchies to move instructions and data across cache levels. As such, processors often spend much time idling upon a demand fetch of memory blocks that miss in higher cache levels.

Prefetching—predicting future memory accesses and issuing requests for the corresponding memory blocks in advance of explicit accesses—is an effective approach to hide memory access latency. There have been a myriad of proposed prefetching techniques, and nearly every modern processor includes some hardware prefetching mechanisms targeting simple and regular memory access patterns. This primer offers an overview of the various classes of hardware prefetchers for instructions and data proposed in the research literature, and presents examples of techniques incorporated into modern microprocessors.

KEYWORDS
hardware prefetching, next-line prefetching, branch-directed prefetching, discontinuity prefetching, stride prefetching, address-correlated prefetching, Markov prefetcher, global history buffer, temporal memory streaming, spatial memory streaming, execution-based prefetching
Contents

Preface ............................................................ xiii

1 Introduction ..................................................... 1
1.1 The Memory Wall ............................................ 1
1.2 Prefetching ................................................... 3
  1.2.1 Predicting Addresses .................................. 3
  1.2.2 Prefetch Lookahead .................................. 4
  1.2.3 Placing Prefetched Values ............................ 4

2 Instruction Prefetching ........................................ 7
2.1 Next-Line Prefetching ....................................... 7
2.2 Fetch-Directed Prefetching ................................ 8
2.3 Discontinuity Prefetching ................................. 10
2.4 Prescient Fetch ............................................. 12
2.5 Temporal Instruction Fetch Streaming .................. 12
2.6 Return-Address Stack-Directed Instruction Prefetching 13
2.7 Proactive Instruction Fetch ............................... 14

3 Data Prefetching ................................................ 15
3.1 Stride and Stream Prefetchers for Data .................. 15
3.2 Address-Correlating Prefetchers ......................... 17
  3.2.1 Jump Pointers ....................................... 17
  3.2.2 Pair-Wise Correlation ............................... 18
  3.2.3 Markov Prefetcher ................................... 18
  3.2.4 Improving Lookahead via Prefetch Depth ............ 19
  3.2.5 Improving Lookahead via Dead Block Prediction ...... 20
  3.2.6 Addressing On-Chip Storage Limitations ............ 21
  3.2.7 Global History Buffer ............................... 22
  3.2.8 Stream Chaining .................................... 24
  3.2.9 Temporal Memory Streaming ....................... 24
  3.2.10 Irregular Stream Buffer ............................. 25
3.3 Spatially Correlated Prefetching ......................... 26
  3.3.1 Delta-Correlated Lookup ............................. 27
Since their inception in the 1970’s, microprocessor-based digital platforms have been riding Moore’s law, allowing for doubling of density for the same area roughly every two years. Microprocessors and memory fabrication technologies, however, have been exploiting this increase in density in two somewhat opposing ways. Whereas microprocessor fabrication has focused on increasing the rate at which machine instructions execute, memory fabrication technologies have focused primarily on an increase in capacity with negligible increase in speed. This divergent trend in performance between the processors and memory has led to a phenomenon referred to as the “Memory Wall” [1].

To overcome the memory wall, designers have resorted to a hierarchy of cache memory levels where at each level access latency is traded off for capacity. Caches rely on the principal of memory access locality to reduce the observed memory access time and the performance gap between processors and memory. Unfortunately, there are a number of important classes of workloads that exhibit adverse memory access patterns that baffle the simple policies built into modern cache hierarchies to move instructions and data across the cache levels. As such, processors often spend much time idling upon a demand fetch of memory blocks that miss in higher cache levels.

Prefetching—predicting future memory accesses and issuing requests for the corresponding memory blocks in advance of explicit accesses by a processor—is quite promising as an approach to hide memory access latency. There have been a myriad of hardware and software approaches to prefetching. A number of effective hardware prefetching mechanisms targeting simple and regular memory access patterns have been incorporated into modern microprocessors to prefetch instructions and data.

This primer offers an overview of the various classes of hardware prefetchers for instructions and data that have been proposed over the years, and presents examples of techniques incorporated into modern microprocessors. Although the techniques covered in this book are by no means comprehensive, they cover important instances of techniques from each class and as such the book serves as a suitable survey for those who plan to familiarize themselves with the domain. We cover prefetching for instruction and data caches, but many of the techniques we discuss may also be applicable to prefetching memory translations into translation lookaside buffers (see, e.g., [2]).

This primer is broken down into four chapters. In Chapter 1, we present an introduction to the memory hierarchy and general prefetching concepts. In Chapter 2, we describe techniques to prefetch instructions. Chapter 3 covers techniques to prefetch data, and we give concluding remarks in Chapter 4. The instruction prefetching techniques cover next-line prefetchers, branch-directed prefetching, discontinuity prefetchers, and temporal instruction streaming. The data prefetchers in-
clude stride and stream-based data prefetchers, address-correlated prefetching, spatially correlated prefetching, and execution-based prefetching.

We assume the reader is familiar with the basics of processor architecture and caches and has some familiarity with more advanced topics like out of order execution. This book enumerates the key issues in designing hardware prefetchers and provides high-level descriptions of a variety of prefetching techniques. We refer the reader to the cited publications for more complete microarchitectural details and performance evaluations of the prefetching schemes.

We would like to acknowledge those who helped contribute to this book. Thanks to Mark Hill for shepherding us through the writing and editorial process. Thanks to Cansu Kaynak and Michael Ferdman for providing figures and comments on drafts of this book. Thanks to Margaret Martonosi, Calvin Lin, and other anonymous reviewers for their detailed feedback that helped to improve this book.
CHAPTER 1

Introduction

1.1 THE MEMORY WALL

Figure 1.1 depicts the growing disparity between processor and memory performance in the past four decades. Innovations in microarchitecture, circuits, and fabrication technologies have led to an exponential increase in processor performance over this period. Meanwhile, DRAM has primarily benefitted from increases in density and DRAM speeds have improved only nominally. While future projections indicate that processor performance improvement may not continue at the same rate, the current gap in performance will necessitate techniques to mitigate long memory access latencies for years to come.

---

Computer architects have historically attempted to bridge this performance gap using a hierarchy of cache memories. Figure 1.2 depicts the anatomy of a modern computer’s cache hierarchy. The hierarchy consists of cache memories that trade off capacity for lower latency at each level. The purpose of the hierarchy is to improve the apparent average memory access time by frequently handling a memory request at the cache, avoiding the comparatively long access latency of DRAM. The cache levels closer to the cores are smaller but faster. Each level provides a temporary repository for recently accessed memory blocks to reduce the effective memory access latency. The more frequently memory blocks are found in levels closer to the cores, the lower the access latency. We refer to the cache(s) closest to the core as the L1 caches and then number cache levels successively, referring to the final cache as the last level cache (LLC).

---

Figure 1.1: The growing disparity between processor and memory performance. From [3].
1. INTRODUCTION

The hierarchy relies on two types of memory reference locality. Temporal locality refers to memory that has been recently accessed and is likely to be accessed again. Spatial locality refers to memory in physical proximity that is likely to be accessed because near-neighbor instructions and data are often related.

While locality is extremely powerful as a concept to exploit and reduce the effective memory access latency, it relies on two basic premises that do not necessarily hold for all workloads, particularly as the cache hierarchies grow deeper. The first premise is that one cache size fits all workloads and access patterns. In fact, the capacity demands of modern workloads vary drastically, and differing workloads benefit from different trade-offs in the capacity and speed of cache hierarchy levels. The second premise is that a single strategy for allocating and replacing cache entries (typically allocating on demand and replacing entries that have not been recently used) is suitable for all workloads. However, again, there is enormous variation in memory access patterns for which a simple strategy for deciding which blocks to cache may fare poorly.

There are a myriad of techniques that have been proposed from the algorithmic, compiler-level, and system software level all the way down to hardware to overcome the Memory Wall. These techniques include cache-oblivious algorithms, code and data layout optimizations at the compiler level, to hardware-centric approaches. Moreover, many software-based techniques have been proposed for prefetching. In this book, we focus on hardware-based techniques for prefetching instructions and data. For a more comprehensive treatment of the memory system, we refer the reader to the synthesis lecture by Jacob [4].

Figure 1.2: A modern memory hierarchy.
1.2 PREFETCHING

One way to hide memory access latency is to prefetch. Prefetching refers to the act of predicting a subsequent memory access and fetching the required values ahead of the memory access to hide any potential long latency. In the limit, a memory access does not incur any additional overhead and memory appears to have a performance equal to a processor register. In practice, however, prefetching may not always be timely or accurate. Late or inaccurate prefetches waste energy and, in the worst case, can hurt performance.

To hide latency effectively, a prefetching mechanism must: (1) predict the address of a memory access (i.e., be accurate), (2) predict when to issue a prefetch (i.e., be timely), and (3) choose where to place prefetched data (and, potentially, which other data to replace).

1.2.1 PREDICTING ADDRESSES

Predicting the correct memory addresses is a key challenge for prefetching mechanisms. If addresses are predicted correctly, the prefetching mechanism will have the opportunity to fetch them in advance and hide the memory access latency. If addresses are not predicted accurately, prefetching may cause pollution in the cache hierarchy (i.e., prefetched cache blocks would evict potentially useful cache blocks) and generate excessive traffic and contention in the memory system.

Predicting memory addresses may not be so simple. A data reference may be an access to a standalone variable or an element of a data structure and the nature of the reference depends on what the program is doing at a particular instance of execution. There are algorithms and data structure traversals that lend themselves well to both repetitive and predictable patterns (e.g., reading every element of an array sequentially). There are also a number of ways in which memory addresses can be hard to predict. These include, but are not limited to, interleaving of accesses to variables, multiple data structures, and control-flow dependent traversals (e.g., searching a binary tree).

Similarly, an instruction reference will depend on whether the program is executing sequentially or it is taking a branch (i.e., following a discontinuity). While sequential instruction fetch is straightforward, the control-flow behavior and its predictability in the program can impact how effective instruction prefetching can be.

Predicting addresses accurately also depends on the level of the cache hierarchy at which the prefetching is performed. At the highest level, the interface between the processor and level-one cache (Figure 1.2) contains all memory reference information that could enable highly accurate prefetch, but could also lead to a waste of resources recording prefetch information for accesses that will hit in the first level cache anyway, and thus do not require prefetch. Conversely, at lower hierarchy levels, the access sequence is filtered, observing only the misses from higher levels. Thus, otherwise effective prefetching algorithms may be confused by access-sequence perturbations from effects like cache placement and replacement policy.
Finally, there is typically a trade-off between the aggressiveness of a prefetch strategy and its accuracy; more aggressive prefetching will predict a higher fraction of the addresses actually requested by the processor at the cost of also fetching many more addresses erroneously. For this reason, many evaluation studies of prefetchers report two key metrics that jointly characterize the prefetcher’s effectiveness at predicting addresses. Coverage measures the fraction of explicit processor requests for which a prefetch is successful (i.e., fraction of demand misses eliminated by prefetching). Accuracy measures the fraction of accesses issued by the prefetcher that turn out to be useful (i.e., fraction of correct prefetches over all prefetches). Many simple prefetchers can improve coverage at the expense of accuracy, whereas an ideal prefetcher provides both high accuracy and coverage.

1.2.2 PREFETCH LOOKAHEAD

Ideally, a prefetching mechanism issues a prefetch well in advance and provides enough storage for prefetched data so as to hide all memory access latency. Predicting precisely when to prefetch in practice, however, is a major challenge. Even if addresses are predicted correctly, a prefetcher that issues prefetches too early may not be able to hold all prefetched memory close to the processor long enough prior to access. In the best case, prefetching too early will be useless because the prefetched information will be evicted away from the processor prior to use. In the worst case, it may evict other useful information (e.g., other prefetched memory or useful blocks in higher-level caches). If memory is prefetched late, then it will diminish the effectiveness of prefetching by exposing the memory access latency upon the memory access. In the limit, late prefetches may lead to performance degradation due to additional memory system traffic and poor interaction with mechanisms designed to prioritize time-critical demand accesses.

1.2.3 PLACING PREFETCHED VALUES

The simplest and perhaps oldest software strategy for prefetching data is to load it into a processor register much like any other explicit load operation. Many architectures, in particular modern out-of-order processors, do not stall execution when a load is issued, but rather stall dependent instructions only when the value of a load is consumed by another instruction. Such a prefetch strategy is often called a binding prefetch because the value of subsequent uses of the data is bound at the time the prefetch is issued. This approach comes with a number of drawbacks: (1) it consumes precious processor registers, (2) it obligates the hardware to perform the prefetch, even if the memory system is heavily loaded, (3) it leads to semantic difficulties in the case the prefetch address is erroneous (e.g., should a prefetch of an invalid address result in a memory protection fault?), and (4) it is unclear how to apply this strategy to instructions.
Instead, most hardware prefetching techniques place prefetched values either directly into the cache hierarchy, or into supplemental buffers that augment the cache hierarchy, and are accessed concurrently. In multicore and multiprocessor systems, these caches and buffers participate in the cache coherence protocol, and hence the value of a prefetched memory location may change during the interval between the prefetch and a subsequent access; it is the hardware’s responsibility to ensure the access sees the up-to-date value. Such prefetching strategies are referred to as non-binding. In these schemes, prefetching is purely a performance optimization and does not affect the semantics of a program.

All the hardware prefetched we consider in this book fall into this latter, non-binding category. They differ in precisely where they place prefetched values and what they replaced to make room for newly prefetched memory.