15-740/18-740 Computer Architecture Lecture 14: Prefetching

Prof. Onur Mutlu
Carnegie Mellon University

Announcements

- Project Milestone I
  - Due Today

- Paper Reviews
  - Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.
  - Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.
  - Due Friday October 22

Last Time

- Enhancements to improve cache performance
  - Victim caches
  - Hashing
  - Pseudo-associativity
  - Skewed associative caches
  - Software changes to improve hit rate
  - Non-blocking caches, MSHRs
  - Reducing miss cost via software
- Multiple cache accesses per cycle
  - True multiporting
  - Virtual multiporting
  - Multiple cache copies
  - Banking (interleaving)

Today: Prefetching

- Why prefetch? Why could/does it work?
- The four questions
  - What (to prefetch), when, where, how
- Software prefetching
- Hardware prefetching algorithms
- Execution-based prefetching
- Prefetching performance
  - Coverage, accuracy, timeliness
  - Bandwidth consumption, cache pollution
- Prefetcher throttling
- Issues in multi-core

Readings in Prefetching

- Required:
  - Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.
  - Joseph and Grunwald, "Prefetching using Markov Predictors," ISCA 1997.
- Recommended:
  - Mowry et al., "Design and Evaluation of a Compiler Algorithm for Prefetching," ASPLOS 1992.
  - Srinath et al., "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers", HPCA 2007.
  - Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003.

Prefetching

- Idea: Fetch the data before it is needed (i.e. pre-fetch) by the program
- Why?
    - Memory latency is high. If we can prefetch accurately and early enough we can reduce/eliminate that latency.
    - Can eliminate compulsory cache misses
    - Can eliminate all cache misses? Capacity, conflict, coherence?
- Involves predicting which address will be needed in the future
    - Works if programs have predictable miss address patterns

Prefetching and Correctness

- Does a misprediction in prefetching affect correctness?

- No, prefetched data at a “mispredicted” address is simply not used
- There is no need for state recovery

- In contrast to branch misprediction or value misprediction

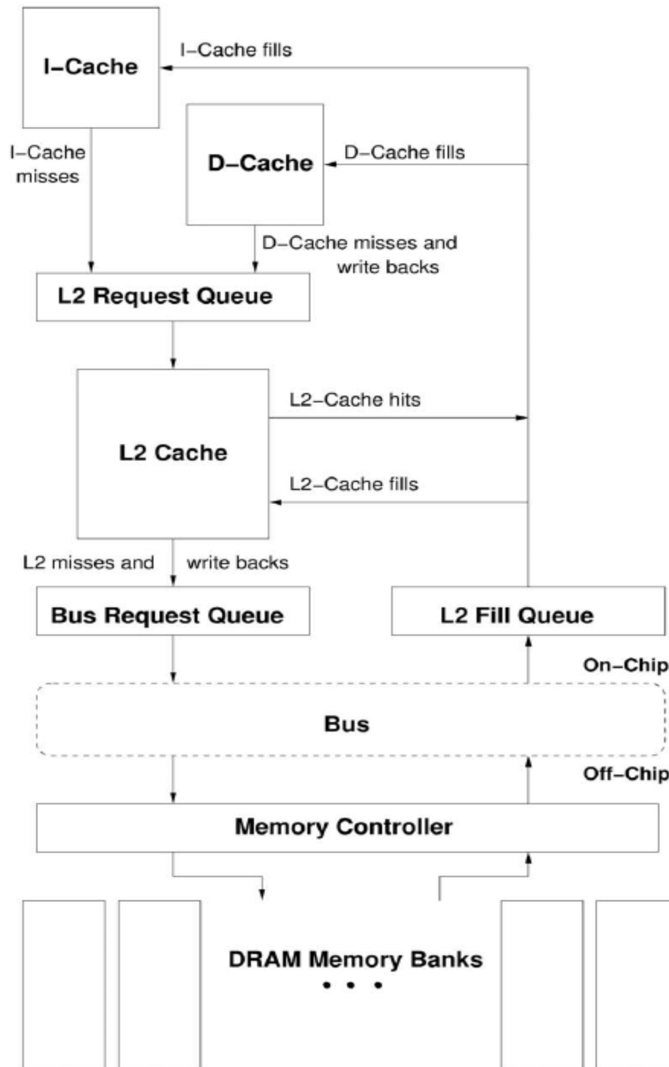
Basics

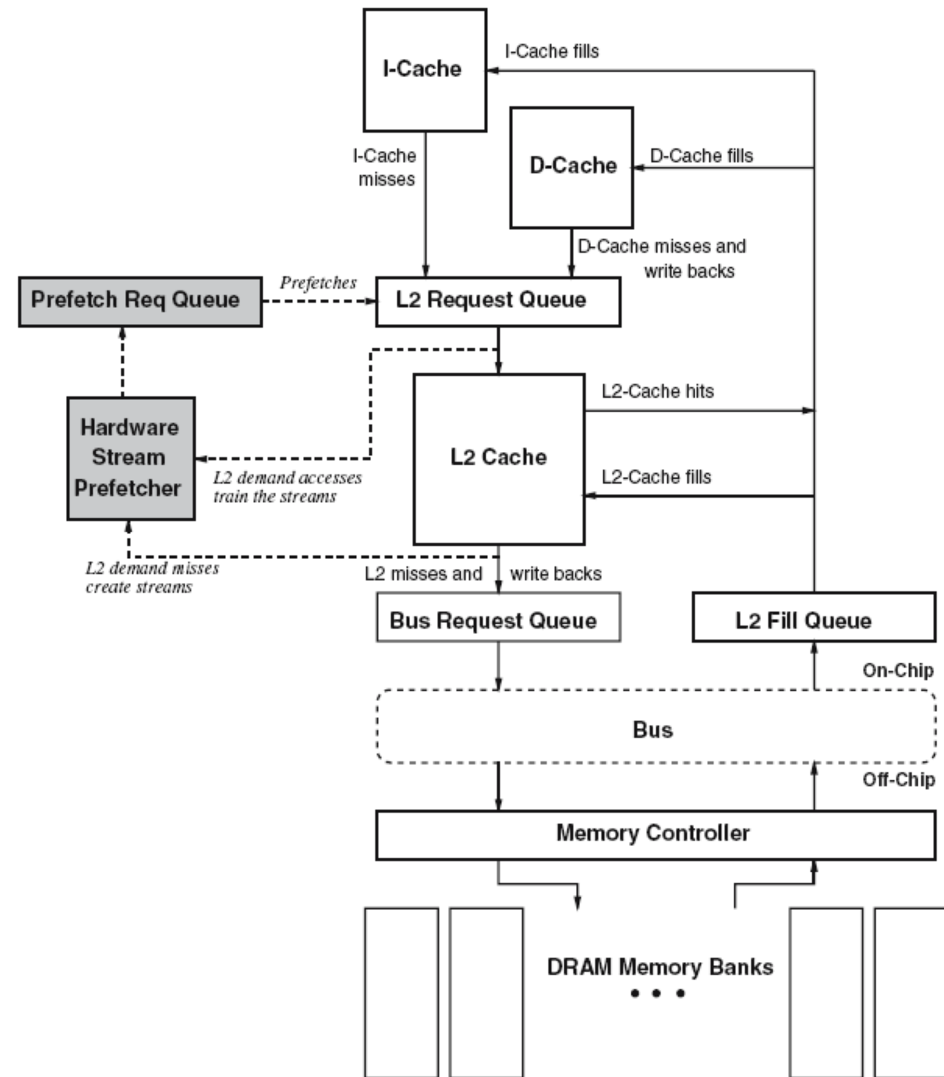
- In modern systems, prefetching is usually done in cache block granularity

- Prefetching is a technique that can reduce both
  - Miss rate
  - Miss latency

- Prefetching can be done by
  - hardware
  - compiler
  - programmer

How a Prefetcher Fits in the Memory System





Prefetching: The Four Questions

- What
  - What addresses to prefetch
- When
  - When to initiate a prefetch request
- Where
  - Where to place the prefetched data
- How
  - Software, hardware, execution-based, cooperative

Challenges in Prefetching: What

- What addresses to prefetch
  - Prefetching useless data wastes resources
    - Memory bandwidth
    - Cache or prefetch buffer space
    - Energy consumption
    - These could all be utilized by demand requests or more accurate prefetch requests
  - Accurate prediction of addresses to prefetch is important
    - Prefetch accuracy = used prefetches / sent prefetches
- How do we know what to prefetch
  - Predict based on past access patterns
  - Use the compiler's knowledge of data structures

- Prefetching algorithm determines what to prefetch

Challenges in Prefetching: When

- When to initiate a prefetch request
  - Prefetching too early
    - Prefetched data might not be used before it is evicted from storage
  - Prefetching too late
    - Might not hide the whole memory latency

- When a data item is prefetched affects the timeliness of the prefetcher
- Prefetcher can be made more timely by
  - Making it more aggressive: try to stay far ahead of the processor's access stream (hardware)
  - Moving the prefetch instructions earlier in the code (software)

Challenges in Prefetching: Where (I)

- Where to place the prefetched data
  - In cache

    \+ Simple design, no need for separate buffers

    -- Can evict useful demand data → cache pollution
  - In a separate prefetch buffer

    \+ Demand data protected from prefetches → no cache pollution

    -- More complex memory system design

    - Where to place the prefetch buffer
    - When to access the prefetch buffer (parallel vs. serial with cache)
    - When to move the data from the prefetch buffer to cache
    - Keeping the prefetch buffer coherent

- Many modern systems place prefetched data into the cache
  - Intel Pentium 4, Core2's, AMD systems, IBM POWER4,5,6, …

Challenges in Prefetching: Where (II)

- Which level of cache to prefetch into?
  - Memory to L2, memory to L1. Advantages/disadvantages?
  - L2 to L1? (a separate prefetcher between levels)

- Where to place the prefetched data in the cache?
  - Do we treat prefetched blocks the same as demand-fetched blocks?
  - Prefetched blocks are not known to be needed
    - With LRU, a demand block is placed into the MRU position

- Do we skew the replacement policy such that it favors the demand-fetched blocks?
  - E.g., place all prefetches into the LRU position in a way?

Challenges in Prefetching: Where (III)

- Where to place the hardware prefetcher in the memory hierarchy?
  - In other words, what access patterns does the prefetcher see?
  - L1 hits and misses
  - L1 misses only
  - L2 misses only

- Seeing a more complete access pattern:

  \+ Potentially better accuracy and coverage in prefetching

  -- Prefetcher needs to examine more requests (bandwidth intensive, more ports into the prefetcher?)

Challenges in Prefetching: How

- Software prefetching
  - ISA provides prefetch instructions
  - Programmer or compiler inserts prefetch instructions (effort)
  - Usually works well only for "regular access patterns"

- Hardware prefetching
  - Hardware monitors processor accesses
  - Memorizes or finds patterns/strides
  - Generates prefetch addresses automatically

- Execution-based prefetchers
  - A "thread" is executed to prefetch data for the main program
  - Can be generated by either software/programmer or hardware

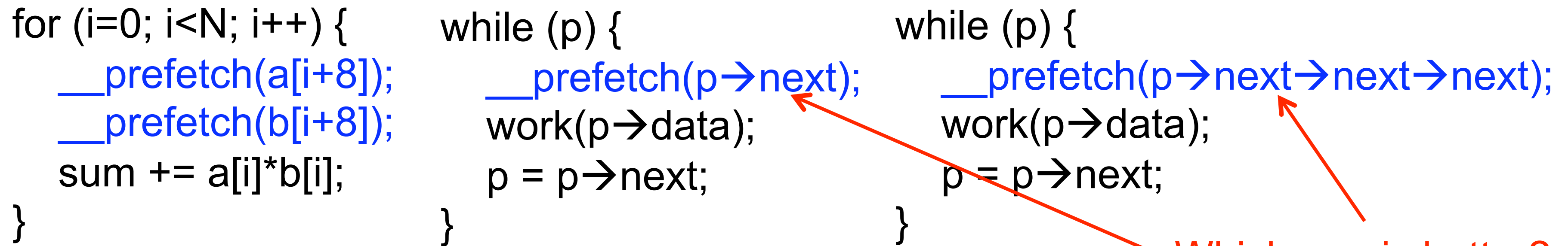
Software Prefetching (I)

- Idea: Compiler/programmer places prefetch instructions into appropriate places in code

- Mowry et al., "Design and Evaluation of a Compiler Algorithm for Prefetching," ASPLOS 1992.

- Two types: binding vs. non-binding
  - Binding: Prefetch into a register (using a regular load)
    - + No need for a separate "prefetch" instruction
    - -- Takes up registers. Exceptions?
    - -- What if another processor modifies the data value before it is used?
  - Non-binding: Prefetch into cache (special instruction?)
    - + No coherence issues since caches are coherent
    - -- Prefetches treated differently from regular loads

Software Prefetching (II)



- Can work for very regular array-based access patterns. Issues:
  -- Prefetch instructions take up processing/execution bandwidth
  - How early to prefetch? Determining this is difficult
    -- Prefetch distance depends on hardware implementation (memory latency, cache size, time between loop iterations) → portability?
    -- Going too far back in code reduces accuracy (branches in between)
  - Need "special" prefetch instructions in ISA?
    - Not really. Alpha load into register 31 treated as prefetch (r31==0)
    - PowerPC *dcbt* (data cache block touch) instruction

  -- Not easy to do for pointer-based data structures

X86 PREFETCH Instruction

PREFETCH*h*—Prefetch Data Into Caches

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 18 /1	PREFETCHT0 *m8*	Valid	Valid	Move data from *m8* closer to the processor using T0 hint.
0F 18 /2	PREFETCHT1 *m8*	Valid	Valid	Move data from *m8* closer to the processor using T1 hint.
0F 18 /3	PREFETCHT2 *m8*	Valid	Valid	Move data from *m8* closer to the processor using T2 hint.
0F 18 /0	PREFETCHNTA *m8*	Valid	Valid	Move data from *m8* closer to the processor using NTA hint.

Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
  - — Pentium III processor—1st- or 2nd-level cache.
  - — Pentium 4 and Intel Xeon processors—2nd-level cache.
- T1 (temporal data with respect to first level cache)—prefetch data into level 2 cache and higher.
  - — Pentium III processor—2nd-level cache.
  - — Pentium 4 and Intel Xeon processors—2nd-level cache.
- T2 (temporal data with respect to second level cache)—prefetch data into level 2 cache and higher.
  - — Pentium III processor—2nd-level cache.
  - — Pentium 4 and Intel Xeon processors—2nd-level cache.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.
  - — Pentium III processor—1st-level cache
  - — Pentium 4 and Intel Xeon processors—2nd-level cache

microarchitecture dependent specification

different instructions for different cache levels

Software Prefetching (III)

- Where should a compiler insert prefetches?

  - Prefetch for every load access?
    - Too bandwidth intensive (both memory and execution bandwidth)

  - Profile the code and determine loads that are likely to miss
    - What if profile input set is not representative?

  - How far ahead before the miss should the prefetch be inserted?
    - Profile and determine probability of use for various prefetch distances from the miss
      - What if profile input set is not representative?
      - Usually need to insert a prefetch far in advance to cover 100s of cycles of main memory latency → reduced accuracy

Hardware Prefetching (I)

- Idea: Specialized hardware observes load/store access patterns and prefetches data based on past access behavior

- Tradeoffs:

  + Can be tuned to system implementation

  + No code portability issues (in terms of performance variation between implementations)

  + Does not waste instruction execution bandwidth

  -- More hardware complexity to detect patterns

    - Software can be more efficient in some cases