EE 156: Advanced Topics in Computer Architecture

Spring 2023
Tufts University

Instructor: Prof. Mark Hempstead
mark@ece.tufts.edu

Lecture 3: Cache Optimizations, Cache Hiearchies and Prefetching
(Chapter 2)

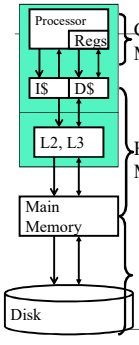
Unit 1: The Memory Hierarchy

Unit 1: The Memory Hierarchy (3-4 weeks)

- **Introduction and Performance Metrics [Chapter 1]**
- **Review of Basic Caches and Set Associativity [Appendix B]**
- **Advanced Cache Optimization Techniques and Replacement policies [Ch 2]**
  - **Cache Miss Types (3 Cs)**
  - **Cache Hierarchies**
  - **10 Optimizations (From Textbook)**
- **Prefetching [Extra Slides]**
- Memory consistency and Cache coherence [Chapter 5]
- Software interfaces and memory consistency
- Transactional memory
- Review of Virtual Memory and TLBs [Appendix B]
- Advanced Virtual Memory [SLCA: Bhattacharjee and Lustig]
- New Non-Volatile Memory (NVM) technologies

- **Textbook Reading: Read Chapter 2 Memory Systems!**

Concrete Memory Hierarchy



- 0th level: **Registers**
- 1st level: **Primary caches**
  - Split instruction (I$) and data (D$)
  - Typically 8KB to 64KB each
- 2nd level: **2nd and 3rd cache** (L2, L3)
  - On-chip, typically made of SRAM
  - 2nd level typically ~256KB to 512KB
  - "Last level cache" typically 4MB to 16MB
- 3rd level: **main memory**
  - Made of DRAM ("Dynamic" RAM)
  - Typically 1GB to 4GB for desktops/laptops
    - Servers can have 100s of GB
- 4th level: **disk (swap and files)**
  - Uses magnetic disks or flash drives
  - 10s of GB or over 1 TB

4 Questions for Memory Hierarchy at any level

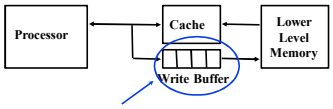
- Q1: Where can a block be placed in the upper level? *(Block placement)*
- Q2: How is a block found if it is in the upper level? *(Block identification)*
- Q3: Which block should be replaced on a miss? *(Block replacement)*
- Q4: What happens on a write? *(Write strategy)*

Q4: What happens on a write?

	Write-Through	Write-Back
Policy	Data written to cache block also written to lower-level memory	Write data only to the cache Update lower level when a block falls out of the cache
Debug	Easy	Hard
Do read misses create victim writes?	No	Yes
Do repeated writes make it to lower level?	Yes	No

Additional option -- let writes to an un-cached address allocate a new cache line ("write-allocate").

Write Buffers for Write-Through Caches



Holds data awaiting write-through to lower level memory

Q. Why a write buffer ? A. So CPU doesn't stall

Q. Why a buffer, why not just one register ? A. Bursts of writes are common.

Q. Are Read After Write (RAW) hazards an issue for write buffer? A. Yes! Drain buffer before next read, or check write buffer on read misses.

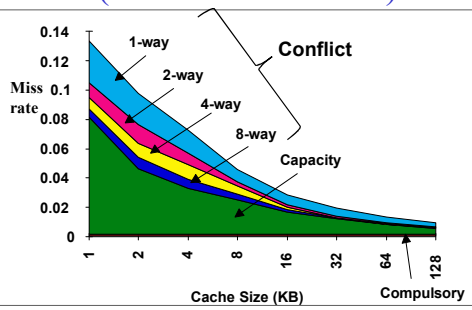
Write misses?

- Write Allocate
  - Block is allocated on a write miss
  - Standard write hit actions follow the block allocation
  - Write misses, like read misses, require a victim evict
  - Goes well with write-back
- No-write Allocate
  - Write misses do not allocate a block
  - Only update lower-level memory
  - Blocks only allocate on Read misses!
  - Goes well with write-through

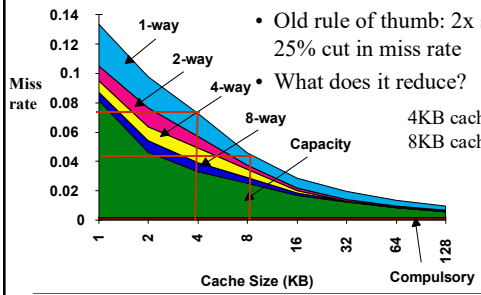
Where do misses come from?

- Classifying Misses: 3 Cs
  - *Compulsory*—First access to a block. Also called *cold start misses* or *first reference misses*. *(Misses in even an Infinite Cache)*
  - *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. *(Misses in Fully Associative Cache)*
  - *Conflict*—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called *collision misses* or *interference misses*. *(Remaining Misses)*
- 4th "C": *Coherence* - Misses caused by cache coherence.

3Cs Absolute Miss Rate (SPEC Benchmarks)



Cache Size



- Old rule of thumb: 2x size => 25% cut in miss rate
- What does it reduce?

4KB cache => .075
8KB cache => .045

Cache Organization?

Assume total cache size not changed. Which of 3Cs is obviously affected (if any) if we

1) Change Block Size: Increased block size means fewer compulsory misses. It also means fewer blocks, which might increase conflict misses.

2) Change Associativity: Increasing associativity means fewer conflict misses.

3) Change Compiler: A compiler might do a better job of making accesses local, which would lessen compulsory and conflict misses.

5 Basic Cache Optimizations (B.3)

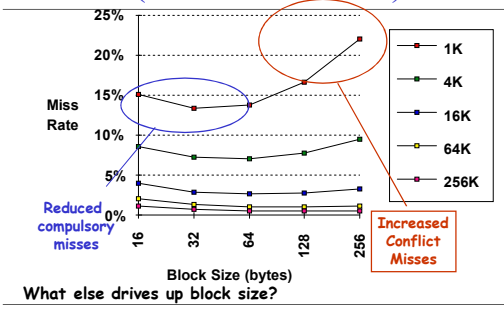
- Reducing Miss Rate

1. Larger Block size (compulsory misses)
2. Higher Associativity (conflict misses)
3. Larger Cache size (capacity misses)

- Reducing Miss Penalty

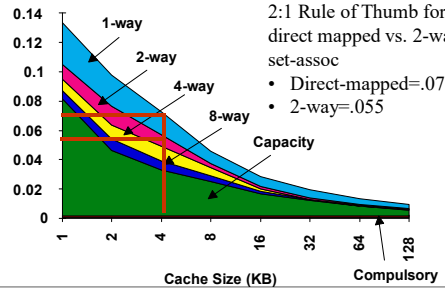
4. Multilevel Caches
5. Giving Reads Priority over Writes
   - E.g., Read complete before earlier writes in write buffer
6. Giving Reads Priority over Writes

Larger Block Size (fixed size&assoc)



What else drives up block size?

Associativity



2:1 Rule of Thumb for direct mapped vs. 2-way set-assoc

- Direct-mapped=.07
- 2-way=.055

Example: Avg. Memory Access Time vs. Miss Rate

- Example: assume Cache Cycle Time = 1.10 for 2-way, 1.12 for 4-way, 1.14 for 8-way vs. CCT direct mapped

Cache Size (KB)	Associativity 1-way	2-way	4-way	8-way
1	2.33	2.15	2.07	2.01
2	1.98	1.86	1.76	1.68
4	1.72	1.67	1.61	1.53
8	1.46	1.48	1.47	1.43
16	1.29	1.32	1.32	1.32
32	1.20	1.24	1.25	1.27
64	1.14	1.20	1.21	1.23
128	1.10	1.17	1.18	1.20

(Red means A.M.A.T. not improved by more associativity)

Explanation: if miss rate is low, then a slight increase in cycle time (which applies to hits as well as misses) outweighs a small miss-rate improvement.

Cache Hierarchies
[B.2]

Designing a Cache Hierarchy

- For any memory component: t_{hit} vs. $\%_{miss}$ tradeoff
- Upper components (I$, D$) emphasize low t_{hit}
  - Frequent access → t_{hit} important. And you're close to the core, so a small absolute difference in t_{hit} is relatively large.
  - t_{miss} is not bad → $\%_{miss}$ less important
  - Lower capacity and lower associativity (to reduce t_{hit})
  - Small-medium block-size (to reduce conflicts)
- Moving down (L2, L3) emphasis turns to $\%_{miss}$
  - Infrequent access → t_{hit} less important. And the communication time to the core is already so big that a slower t_{hit} is not very noticeable.
  - t_{miss} is bad → $\%_{miss}$ important
  - Higher capacity, associativity, and block size (to reduce $\%_{miss}$)
- Larger t_{hit} lets you reduce power on L2, L3
  - Very important since they're so large.

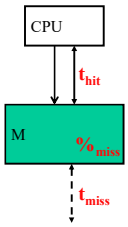
Memory Hierarchy Parameters

Parameter	I$/D$	L2	L3	Main Memory
t_{hit}	2ns	10ns	30ns	100ns
t_{miss}	**10ns**	**30ns**	**100ns**	**10ms (10M ns)**
Capacity	8KB–64KB	256KB–8MB	2–16MB	1-4GBs
Block size	16B–64B	32B–128B	32B-256B	NA
Associativity	1–4	4–16	4-16	NA

Split vs. Unified Caches

- **Split I$/D$**: instructions and data in different caches
  - To minimize structural hazards and t_{hit}
  - Larger unified I$/D$ would be slow, 2nd port even slower
  - Optimize I$ to fetch multiple instructions, no writes
  - Why is 486 I/D$ unified?
- **Unified L2, L3**: instructions and data together
  - To minimize $\%_{miss}$
  + Fewer capacity misses: unused insn capacity can be used for data
  - More conflict misses: insn/data conflicts
    - A much smaller effect in large caches
  - Insn/data structural hazards are rare: simultaneous I$/D$ miss
  - Go even further: unify L2, L3 of multiple cores in a multi-core

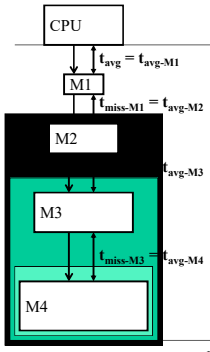
Memory Performance Equation



- For memory component M
  - **Access**: read or write to M
  - **Hit**: desired data found in M
  - **Miss**: desired data not found in M
    - Must get from another (slower) component
  - **Fill**: action of placing data in M
  - **$\%_{miss}$** (miss-rate): #misses / #accesses
  - **t_{hit}**: time to read data from (write data to) M
  - **t_{miss}**: time to read data into M
- Performance metric
  - **t_{avg}**: average memory access time (AMAT)

$$t_{avg} = t_{hit} + (\%_{miss} * t_{miss})$$

Calculating Performance of the Hierarchy



t_{avg}

t_{avg-M1}

$t_{hit-M1} + (\%_{miss-M1} * t_{miss-M1})$

$t_{hit-M1} + (\%_{miss-M1} * t_{avg-M2})$

$t_{hit-M1} + (\%_{miss-M1} * (t_{hit-M2} + (\%_{miss-M2} * t_{miss-M2})))$

$t_{hit-M1} + (\%_{miss-M1} * (t_{hit-M2} + (\%_{miss-M2} * t_{avg-M3})))$

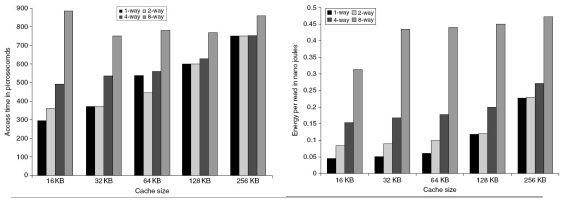
…

Note: These rates are relative to the number of accesses to each level (not total instructions)

TEN ADVANCED CACHE OPTIMIZATIONS
[SECTION 2.3 IN 6TH ED]

Optimization 1: Small simple L1 Caches for reduced hit time and power

- Remember AMAT: $t_{avg} = t_{hit} + (\%_{miss} * t_{miss})$
- Direct mapped small caches are faster and lower energy

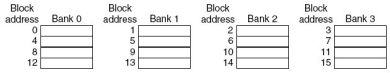


Optimization 2: Way Prediction

- We already talked about this one.
- Prediction accuracy
  - > 90% for two-way
  - > 80% for four-way
  - I-cache has better accuracy than D-cache
- First used on MIPS R10000 in mid-90s (used on ARM Cortex-A8)

Optimization 3: Pipelining Cache and Multibanked Caches

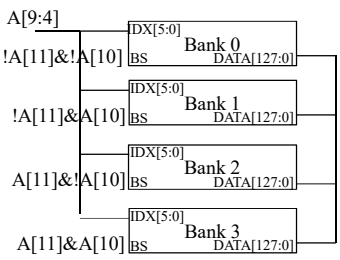
- All modern caches are pipelined. But you can change the # of stages (and then change the frequency accordingly)
- Organize cache as independent banks to support simultaneous access
  - ARM Cortex-A8 supports 1-4 banks for L2
  - Intel i7 supports 4 banks for L1 and 8 banks for L2
  - Practically, nearly all caches are multi-bank anyway; you simply cannot build one huge bank for timing reasons.
  - This allows us to sleep entire banks, saving substantial power.
  - It also allows one cache to service two requests at once if they're in different banks.
- Interleave banks by block address (picture below)



Banked caches

- A large cache (L2, L3) is typically built from numerous banks.
- Various options to build the cache
- Keep each line in one bank.
  - Why? Lets you sleep unused cache banks.
- Divide one cache line among multiple banks.
  - Why? Uses many banks at once, prevents hot spots.
- And anything else in between!

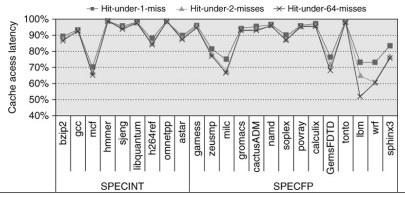
Banking picture



- We picked A[9:4] for index and [11:10] for bank select. Why not make bank select the LSB?
- We built a tri-state data mux; uncommon but easier to draw!
- We showed an entire line of data output.
- BS=0 can turn off clocks in a bank and (for a tristate) turn off output drivers.

Optimization 4: Nonblocking Caches

- Allow hits before previous misses complete: "hit under miss" or "hit under multiple miss"
- Useful when one cache supports multiple threads, or when one thread can issue OOO.
- In general, processors can hide L1 miss penalty but not L2 miss penalty



Optimization 5: Critical Word First, Early Restart

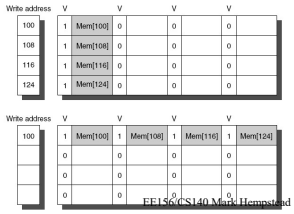
- Problem: wide busses between cache & CPU waste area & power.
- Solution:
  - use a narrower bus, return data over multiple cycles.
  - But what if the CPU did a load on some data that won't arrive until the end of the transfer?
  - Return the critical word first, and then the others.
- The narrower bus hurts bandwidth – but critical-word-first ensures we don't hurt latency.

How do we implement critical-word-first?

- We've shown our caches as having a mux that selects one byte from a line.
  - This is a simplification that has just run out of steam.
  - An L1 cache can return a byte, word, longword, etc. to the register file.
  - An L2 cache returns an entire line to the L1.
  - Why is the L2 line size usually = L1 line size?
    It makes coherency easier.
- If the line is 8B, and we return 2B/cycle over 4 cycles, what bits would drive our 4:1 mux?
  - Address[2:1].

Optimization 6: Merging Write Buffer

- We already talked about a WB hit.
- Next optimization: when storing to a block that is already pending in the write buffer, just update the existing block rather than adding a new one.
- Reduces stalls due to full write buffer
- Do not apply to I/O addresses
- Notice here the each write buffer entry holds FOUR 64-bit words



No write buffering

Write buffering

Optimization 7: Compiler Optimizations

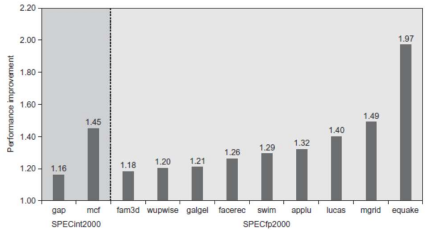
- Loop Interchange
  - Swap nested loops to access memory in sequential order
- Blocking
  - Instead of accessing entire rows or columns, subdivide matrices into blocks
  - Requires more memory accesses but improves locality of accesses
  - Commonly done for large matrices in packages such as MATLAB.

Optimization 8: Hardware Prefetching

- Fetch two blocks on miss (include next sequential block)
- Why would this work better than just doubling the block size?
  - Because you can then invalidate each independently.
- When would it work well or work poorly?
  - Works well if you tend to access memory from lots of sequential locations.
  - Often done for instructions, and for data if the hardware thinks it's very clever.

Hardware Prefetching

- Fetch two blocks on miss (include next sequential block)



Pentium 4 Pre-fetching

Optimization 9: Compiler Prefetching

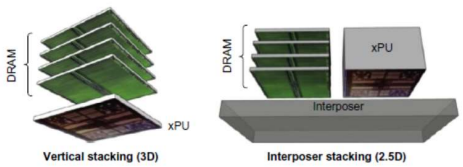
- Insert prefetch instructions before data is needed
- Non-faulting: prefetch doesn't cause exceptions

- Register prefetch
  - Loads data into register
- Cache prefetch
  - Loads data into cache

- Combine with loop unrolling and software pipelining
- Very useful for streaming multi-media data under software control.

Optimization 10: HBM Stacked/Embedded DRAMs

- Stacked DRAMs in same package as processor
  - High Bandwidth Memory (HBM)



Summary of Advanced Cache Optimizations

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined & banked caches	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs
HBM as additional level of cache		+/–	–	+	+	3	Depends on new packaging technology. Effects depend heavily on hit rate improvements

Victim Cache Paper

- We read the victim cache paper

One question are they still used

"Intel's Crystalwell[24] variant of its Haswell processors, equipped with Intel's Iris Pro GT3e embedded graphics and 128 MB of eDRAM, introduced an on-package Level 4 cache which serves as a victim cache to the processors's Level 3 cache.[25] In the Skylake processors the Level 4 cache no longer works as a victim cache.[26]"

https://en.wikipedia.org/wiki/CPU_cache#Specialized_caches
https://en.wikipedia.org/wiki/Victim_cache

Optional Backup Slides: Prefetching Details

We will probably not cover these in class

These Slides are designed to give you the background to understand the paper on software prefetching and also stream buffers

Slides from Onur Mutlu, CMU now ETHZ

Outline of Prefetching Lecture(s)

- 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 (if we get to it)

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 it eliminate all cache misses? Capacity, conflict?
- Involves predicting which address will be needed in the future
  - Works if programs have predictable miss address patterns

Prefetching and Correctness

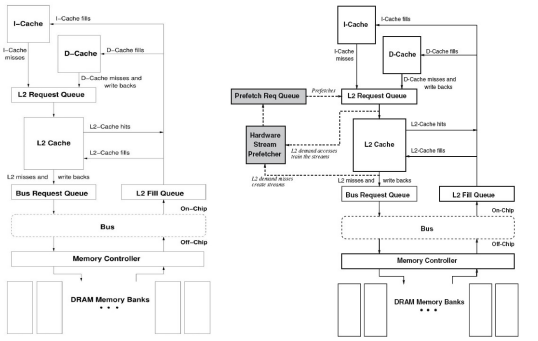
- Does a misprediction in prefetching affect correctness?
- No, prefetched data at a "mispredicted" address is simply not used
- There is no need for state recovery
  - In contrast to branch misprediction or value misprediction

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

43

How a HW Prefetcher Fits in the Memory System



44

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

45

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

46

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)

47

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
      - How to size the prefetch buffer
      - 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, …

48

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.
- Prefetch instructions prefetch data into caches
- Compiler or programmer can insert such instructions into the program

52

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

53

Software Prefetching (II)

```
for (i=0; i<N; i++) {
    __prefetch(a[i+8]);
    __prefetch(b[i+8]);
    sum += a[i]*b[i];
}
```

```
while (p) {
    __prefetch(p→next);
    work(p→data);
    p = p→next;
}
```

```
while (p) {
    __prefetch(p→next→next→next);
    work(p→data);
    p = p→next;
}
```

Which one is better?

- 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?
    - 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

54

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

55

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

  \+ Does not waste instruction execution bandwidth

  -- More hardware complexity to detect patterns

  \- Software can be more efficient in some cases

56

Next-Line Prefetchers

- Simplest form of hardware prefetching: always prefetch next N cache lines after a demand access (or a demand miss)
  - Next-line prefetcher (or next sequential prefetcher)
  - Tradeoffs:

    \+ Simple to implement. No need for sophisticated pattern detection

    \+ Works well for sequential/streaming access patterns (instructions?)

    -- Can waste bandwidth with irregular patterns

    -- And, even regular patterns:

    \- What is the prefetch accuracy if access stride = 2 and N = 1?

    \- What if the program is traversing memory from higher to lower addresses?

    \- Also prefetch "previous" N cache lines?

    Typically implementations of hardware prefetchers do not prefetch across page boundaries. Why?

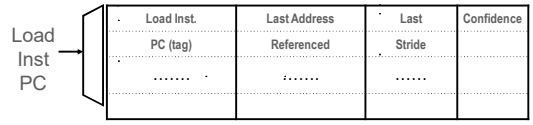
57

Stride Prefetchers

- Two kinds
  - Instruction program counter (PC) based
  - Cache block address based
- Instruction based:
  - Baer and Chen, "An effective on-chip preloading scheme to reduce data access penalty," SC 1991.
  - Idea:
    - Record the distance between the memory addresses referenced by a load instruction (i.e. stride of the load) as well as the last address referenced by the load
    - Next time the same load instruction is fetched, prefetch last address + stride

58

Instruction Based Stride Prefetching



- What is the problem with this?
  - How far can the prefetcher get ahead of the demand access stream?
  - Initiating the prefetch when the load is fetched the next time can be too late
    - Load will access the data cache soon after it is fetched!
  - Solutions:
    - Use lookahead PC to index the prefetcher table (decouple frontend of the processor from backend)
    - Prefetch ahead (last address + N*stride)
    - Generate multiple prefetches

59

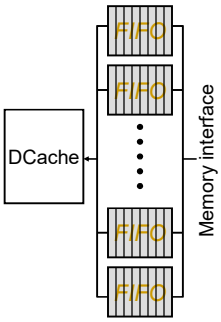
Cache-Block Address Based Stride Prefetching

- Can detect
  - A, A+N, A+2N, A+3N, …
  - Stream buffers are a special case of cache block address based stride prefetching where N = 1

60

Stream Buffers (Jouppi, ISCA 1990)

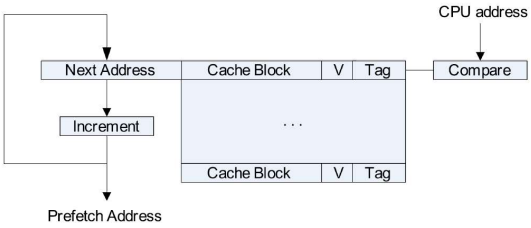
- Each stream buffer holds one stream of sequentially prefetched cache lines
- On a load miss check the head of all stream buffers for an address match
  - if hit, pop the entry from FIFO, update the cache with data
  - if not, allocate a new stream buffer to the new miss address (may have to recycle a stream buffer following LRU policy)
- Stream buffer FIFOs are continuously topped-off with subsequent cache lines whenever there is room and the bus is not busy



Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.

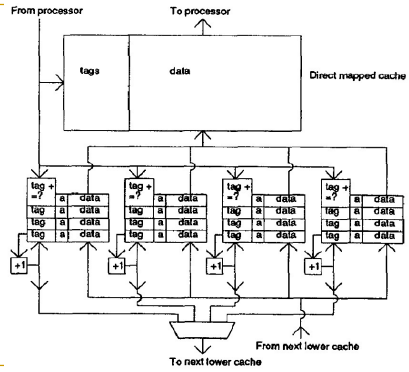
61

Stream Buffer Design



62

Stream Buffer Design



63

Prefetcher Performance (I)

- Accuracy (used prefetches / sent prefetches)
- Coverage (prefetched misses / all misses)
- Timeliness (on-time prefetches / used prefetches)

- Bandwidth consumption
  - Memory bandwidth consumed with prefetcher / without prefetcher
  - Good news: Can utilize idle bus bandwidth (if available)

- Cache pollution
  - Extra demand misses due to prefetch placement in cache
  - More difficult to quantify but affects performance

64