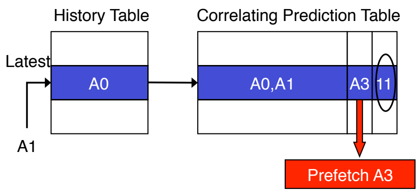
Lecture 7

Prefetching



Spring 2016

Pejman Lotfi-Kamran

Adapted from slides originally developed by Profs. Falsafi, Hill, Hoe, Lipasti, Shen, Smith, Sohi, and Vijaykumar of Carnegie Mellon University, EPFL, Purdue University, and University of Wisconsin.

Where Are We?

Fr	Sa	Su	Mo	Tu
	27-Shahrivar		29-Shahrivar	
	3-Mehr		5-Mehr	
	10-Mehr		12-Mehr	
	17-Mehr		19-Mehr	
	24-Mehr		26-Mehr	
	1-Aban		3-Aban	
	8-Aban		10-Aban	
	15-Aban		17-Aban	
	22-Aban		24-Aban	
	29-Aban		1-Azar	
	6-Azar		8-Azar	
	13-Azar		15-Azar	
	20-Azar		22-Azar	
	27-Azar		29-Azar	
	4-Dey		6-Dey	

This Lecture
- Prefetching

Next Lecture:
- Virtual Memory

Why Prefetch?

Cache miss is either expensive or very expensive

If we can foretell which address the program will reference in the future then we can ensure the location is in the cache ahead of time $\Rightarrow$ *No cache miss!!*

Prefetching takes advantage of regular/repeatable program behavior (in this case the memory reference pattern)

Prefetching is quite safe -- only involves prediction but no "speculative execution"

Prefetching the wrong location can only affect processor performance (more misses) but not correctness

What is Prefetching?

- Fetch memory ahead of time
- Targets compulsory, capacity, & coherence misses

Big challenges:

1. knowing "what" to fetch
   - Fetching useless info wastes valuable resources
2. "when" to fetch it
   - Fetching too early clutters storage
   - Fetching too late defeats the purpose of "pre"-fetching

Software Prefetching

Compiler/programmer places prefetch instructions
- requires ISA support
- why not use regular loads?
- found in ISA's such as SPARC V-9

Prefetch into
- register (binding)
- caches (non-binding): preferred in multiprocessors

Software Prefetching (Cont.)

e.g.,

```
for (I = 1; I < rows; I++)
    for (J = 1; J < columns; J++)
    {
        prefetch(&x[I+1,J]);
        sum = sum + x[I,J];
    }
```


Software Prefetching Support

PowerPC *Data Cache Block Touch* Instruction (dcbt *EA*)

"a hint that performance will probably be improved if the block containing the byte addressed by EA is fetched into the data cache"

A correct implementation of dcbt is to do nothing

Or, as a load instruction with no destination register

except it should not trigger page or protection faults

Where should compilers insert dcbt?
- in front of every load: *wastes I-cache and D-cache bandwidth*
- where are loads likely to miss
  - When traversing large data sets (arrays in scientific code)
- where load misses would really hurt performance
  - pointer arguments to functions
  - linked-list traversal - find loads whose data address is itself the result of a previous load

Hardware Prefetching

What to prefetch?
- one block spatially ahead?
- use address predictors → work well for regular patterns (e.g., x, x+8, x+16,.)

When to prefetch?
- on every reference
- on every miss
- when prior prefetched data is referenced

Where to put prefetched data?
- auxiliary buffers
- caches

Spatial Locality and Sequential Prefetching

Works well for I-cache
- Instruction fetching tend to access memory sequentially

Doesn’t work very well for D-cache
- More irregular access pattern
- regular patterns may have non-unit stride (e.g. matrix code)

Relatively easy to implement
- Large cache block size already have the effect of prefetching
- After loading one-cache line, start loading the next line automatically if the line is not in cache and the bus is not busy

What if you fetch at the wrong time ….

Imagine if you started sequential prefetching of a long cache line and so happens you get a load miss to the middle of that line?

A critical-word-first reload triggered by the load miss itself may actually have restarted computation sooner!!

Stride Prefetchers

Access pattern for a particular static load is more predictable

Reference Prediction Table

Load Inst PC →

Load Inst. PC (tag)	Last Address Referenced	Last Stride	Flags
....	....	......	

Remembers previously executed loads, their PC, the last address referenced, stride between the last two references

When executing a load, look up in RPT and compute the distance between the current data addr and the last addr
- if the new distance matches the old stride
  ⇒ found a pattern, go ahead and prefetch “current addr+stride”
- update “last addr” and “last stride” for next lookup

Stream Buffers

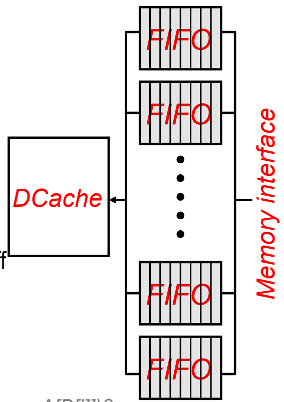
Each stream buffer holds one stream of sequentially prefetched cache lines

No cache pollution

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

Stream buffers can incorporate stride prediction mechanisms to support non-unit-stride streams



Indirect array accesses (e.g., A[B[i]])?

Generalized Access Pattern Prefetchers

How do you prefetch
1. Heap data structures?
2. Indirect array accesses?
3. Generalized memory access patterns?

Current proposals:
- Precomputation prefetchers
- Address correlating prefetchers

Runahead Prefetchers

Proposed for I/O prefetching first (Gibson et al.)

Duplicate the program

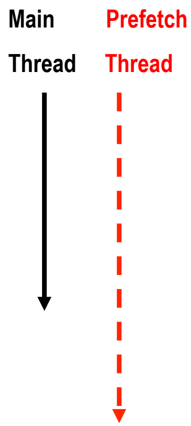
- Only execute the address generating stream
- Let it run ahead

May run as a thread on

- A separate processor
- The same multithreaded processor

Or custom address generation logic

Many names: slipstream, precomp., runahead, …



Runahead Prefetcher

To get ahead:

- Must avoid waiting
- Must compute less

Predict

1. Control flow thru branch prediction
2. Data flow thru value prediction
3. Address generation computation only

+ Prefetch any pattern (need not be repetitive)
– Prediction only as good as branch + value prediction

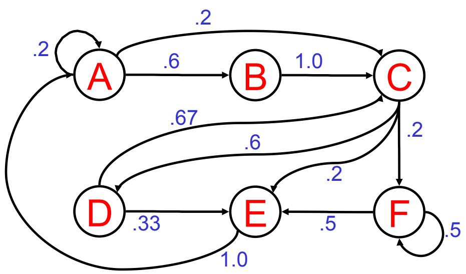
How much prefetch lookahead?

Correlation-Based Prefetching

Consider the following history of Load addresses emitted by a processor

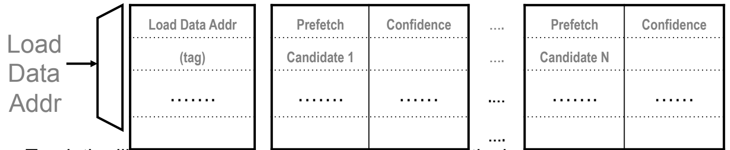
A, B, C, D, C, E, A, C, F, F, E, A, A, B, C, D, E, A ,B C, D, C

After referencing a particular address (say A or E), are some addresses more likely to be referenced next



Markov Model

Correlation-Based Prefetching



Track the likely next addresses after seeing a particular addr.

Prefetch *accuracy* is generally low so prefetch up to N next addresses to increase *coverage* (but this wastes bandwidth)

Prefetch accuracy can be improved by using longer history

- Decide which address to prefetch next by looking at the last K load addresses instead of just the current one
- e.g. index with the XOR of the data addresses from the last K loads
- Using history of a couple loads can increase accuracy dramatically

This technique can also be applied to just the load miss stream

Example Address Correlating: Markov Prefetchers

Markov Prefetchers (Joseph & Grunwald, ISCA'97)

- Correlate subsequent cache misses
- Trigger prefetch on miss
- Predict & prefetch 4 candidates: predicting 1 results in low coverage!
- Prefetch into a buffer

Insufficient Lookahead in Markov

Distance between two misses is usually small

Smaller in out-of-order cores

load/store A1 (miss)

load/store A1 (hit)

...

load/store C3 (miss)

lookahead

load/store A3 (miss)

Fetch on miss

Prefetch on Last Touch

Predict & fetch on last touch

Evict dead block

Enhance fetch lookahead

Fetch directly into L1

load/store A1 (miss)

load/store A1 (hit)

...

load/store C3 (miss)

lookahead

load/store A3 (miss)

Fetch on miss

load/store A1 (miss)

load/store A1 (hit)

...

load/store C3 (miss)

load/store A3 (miss)

Fetch on last touch

Dead-Block Prediction

Correlate a trace of memory accesses to a block

Uniquely identify different dead-times

Access stream to a block frame

PC_0: load/store A0 (hit)

PC_1: load/store A1 (miss) First touch

PC_3: load/store A1 (hit)

PC_3: load/store A1 (hit) Last touch

PC_5: load/store A3 (miss)

Trace = (PC_1, PC_3, PC_3)

Miss-Address Prediction

Correlate last 2 misses within a cache block frame

PC_0: load/store A0 (hit)

…

PC_1: load/store A1

PC_3: load/store A1 (hit)

PC_3: load/store A1 (hit)

…

PC_5: load/store A3

Correlation = (A0,A1) → (A3)

⇩

(A0,A1,PC_1,PC_3,PC_3) → (A3)

⇧

Trace = (PC_1,PC_3, PC_3)

Improving Cache Performance: Summary

Miss rate
- large block size
- higher associativity
- victim caches
- skewed-/pseudo-associativity
- hardware/software prefetching
- compiler optimizations

Miss penalty
- give priority to read misses over writes/writebacks
- subblock placement
- early restart and critical word first
- non-blocking caches
- multi-level caches

Hit time (difficult?)
- small and simple caches
- avoiding translation during L1 indexing (later)
- subblock placement for fast write hits in write through caches