CSE 502: Computer Architecture

Prefetching
Prefetching (1/3)

- Fetch block ahead of demand
- Target compulsory, capacity, (& coherence) misses
  - Why not conflict?

- Big challenges:
  - Knowing “what” to fetch
    - Fetching useless blocks wastes resources
  - Knowing “when” to fetch
    - Too early → clutters storage (or gets thrown out before use)
    - Fetching too late → defeats purpose of “pre”-fetching
Prefetching (2/3)

• Without prefetching:

• With prefetching:

• Or:

Prefetching must be accurate and timely
Prefetching (3/3)

- Without prefetching:

- With prefetching:

  Prefetching removes loads from critical path
Common “Types” of Prefetching

• Software
• Next-Line, Adjacent-Line
• Next-N-Line
• Stream Buffers
• Stride
• “Localized” (e.g., PC-based)
• Pointer
• Correlation
Software Prefetching (1/4)

- Compiler/programmer places prefetch instructions

- Put prefetched value into...
  - Register (binding, also called “hoisting”)
    - May prevent instructions from committing
  - Cache (non-binding)
    - Requires ISA support
    - May get evicted from cache before demand
Hoisting must be aware of dependencies

(Cache misses in red)

Hopefully the load miss is serviced by the time we get to the consumer

Using a prefetch instruction can avoid problems with data dependencies
for (I = 1; I < rows; I++)
{
    for (J = 1; J < columns; J++)
    {
        prefetch(&x[I+1,J]);
        sum = sum + x[I,J];
    }
}
Software Prefetching (4/4)

• Pros:
  – Gives programmer control and flexibility
  – Allows time for complex (compiler) analysis
  – No (major) hardware modifications needed

• Cons:
  – Hard to perform timely prefetches
    • At IPC=2 and 100-cycle memory → move load 200 inst. earlier
    • Might not even have 200 inst. in current function
  – Prefetching earlier and more often leads to low accuracy
    • Program may go down a different path
  – Prefetch instructions increase code footprint
    • May cause more I$ misses, code alignment issues
Hardware Prefetching (1/3)

- Hardware monitors memory accesses
  - Looks for common patterns
- Guessed addresses are placed into *prefetch queue*
  - Queue is checked when no demand accesses waiting
- Prefetchers look like READ requests to the hierarchy
  - Although may get special “prefetched” flag in the state bits

- Prefetchers trade bandwidth for latency
  - Extra bandwidth used *only* when guessing incorrectly
  - Latency reduced *only* when guessing correctly

No need to change software
Hardware Prefetching (2/3)
• Real CPUs have multiple prefetchers
  – Usually closer to the core (easier to detect patterns)
  – Prefetching at LLC is hard (cache is banked and hashed)
Next-Line (or Adjacent-Line) Prefetching

- On request for line X, prefetch X+1 (or X^0x1)
  - Assumes spatial locality
    - Often a good assumption
  - Should stop at physical (OS) page boundaries
- Can often be done efficiently
  - Adjacent-line is convenient when next-level block is bigger
  - Prefetch from DRAM can use bursts and row-buffer hits
- Works for I$ and D$
  - Instructions execute sequentially
  - Large data structures often span multiple blocks

Simple, but usually not timely
Next-N-Line Prefetching

• On request for line X, prefetch X+1, X+2, ..., X+N
  – N is called “prefetch depth” or “prefetch degree”

• Must carefully tune depth N. Large N is ...
  – More likely to be useful (correct and timely)
  – More aggressive → more likely to make a mistake
    • Might evict something useful
  – More expensive → need storage for prefetched lines
    • Might delay useful request on interconnect or port

Still simple, but more timely than Next-Line
Stream Buffers (1/3)

• What if we have multiple inter-twined streams?
  – A, B, A+1, B+1, A+2, B+2, ...

• Can use multiple *stream buffers* to track streams
  – Keep next-N available in buffer
  – On request for line X, shift buffer and fetch X+N+1 into it

• Can extend to “quasi-sequential” stream buffer
  – On request Y in [X...X+N], advance by Y-X+1
  – Allows buffer to work when items are skipped
  – Requires expensive (associative) comparison
Stream Buffers (2/3)

Figures from Jouppi “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,” ISCA’90
Stream Buffers (3/3)

Can support multiple streams in parallel
Access patterns often follow a stride
- Accessing column of elements in a matrix
- Accessing elements in array of structs

Detect stride $S$, prefetch depth $N$
- Prefetch $X+1 \cdot S, X+2 \cdot S, \ldots, X+N \cdot S$
Stride Prefetching (2/2)

• Must carefully select depth $N$
  – Same constraints as Next-$N$-Line prefetcher

• How to determine if $A[i] \rightarrow A[i+1]$ or $X \rightarrow Y$?
  – Wait until $A[i+2]$ (or more)
  – Can vary prefetch depth based on confidence
    • More consecutive strided accesses $\rightarrow$ higher confidence

New access to $A+3N$

Last Addr Stride Count

| $A+2N$ | $N$ | $2$ |

$A+4N$

Update count

Do prefetch?

$>2$
"Localized" Stride Prefetchers (1/2)

• What if multiple strides are interleaved?
  – No clearly-discernible stride
  – Could do multiple strides like stream buffers
    • Expensive (must detect/compare many strides on each access)
  – Accesses to structures usually *localized* to an instruction

Load R1 = [R2]
Load R3 = [R4]
Add R5, R1, R3
Store [R6] = R5

Use an array of strides, indexed by PC
“Localized” Stride Prefetchers (2/2)

• Store PC, last address, last stride, and count in RPT
• On access, check RPT (Reference Prediction Table)
  – Same stride? \( \rightarrow \) count++ if yes, count-- or count=0 if no
  – If count is high, prefetch (last address + stride*N)

PCa: 0x409A34 Load R1 = [R2]

PCb: 0x409A38 Load R3 = [R4]

PCc: 0x409A40 Store [R6] = R5

<table>
<thead>
<tr>
<th>Tag</th>
<th>Last Addr</th>
<th>Stride</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x409</td>
<td>A+3N</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>0x409</td>
<td>X+3N</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>0x409</td>
<td>Y+2N</td>
<td>N</td>
<td>1</td>
</tr>
</tbody>
</table>

If confident about the stride (count > \( C_{\text{min}} \)), prefetch (A+4N)
Other Patterns

• Sometimes accesses are regular, but no strides
  – Linked data structures (e.g., lists or trees)
Pointer Prefetching (1/2)

Data filled on cache miss → (512 bits of data)

8029 | 0 | 1 | 4128 | 90120230 | 90120758 | 14 | 4128

Nope | Nope | Nope | Nope | Maybe! | Maybe! | Nope | Nope

struct bintree_node_t {
    int data1;
    int data2;
    struct bintree_node_t * left;
    struct bintree_node_t * right;
};

Go ahead and prefetch these (needs some help from the TLB)

This allows you to walk the tree (or other pointer-based data structures which are typically hard to prefetch)

Pointers usually “look different”
Pointer Prefetching (2/2)

- Relatively cheap to implement
  - Don’t need extra hardware to store patterns
- Limited **lookahead** makes timely prefetches hard
  - Can’t get next pointer until fetched data block

Stride Prefetcher:

```
X
X+N
X+2N
```

Pointer Prefetcher:

```
A
B
C
```
Pair-wise Temporal Correlation (1/2)

- Accesses exhibit *temporal correlation*
  - If E followed D in the past → if we see D, prefetch E

Can use recursively to get more lookahead 😊
Pair-wise Temporal Correlation (2/2)

- Many patterns more complex than linked lists
  - Can be represented by a *Markov Model*
  - Required tracking *multiple* potential successors

- Number of candidates is called **breadth**

Recursive breadth & depth grows exponentially 😞
Increasing Correlation History Length

- Longer history enables more complex patterns
  - Use history hash for lookup
  - Increases training time

DFS traversal: ABDBEBACFCGCA

Much better accuracy 😊, exponential storage cost 😞
Spatial Correlation (1/2)

Database Page in Memory (8kB)

- Irregular layout $\rightarrow$ non-strided
- Sparse $\rightarrow$ can’t capture with cache blocks
- But, repetitive $\rightarrow$ predict to improve MLP

Large-scale *repetitive* spatial access patterns
Spatial Correlation (2/2)

- Logically divide memory into regions
- Identify region by base address
- Store spatial pattern (bit vector) in correlation table
Evaluating Prefetchers

• Compare against larger caches
  – Complex prefetcher vs. simple prefetcher with larger cache

• Primary metrics
  – **Coverage**: prefetched hits / base misses
  – **Accuracy**: prefetched hits / total prefetches
  – **Timeliness**: latency of prefetched blocks / hit latency

• Secondary metrics
  – **Pollution**: misses / (prefetched hits + base misses)
  – Bandwidth: total prefetches + misses / base misses
  – Power, Energy, Area...
Hardware Prefetcher Design Space

• What to prefetch?
  – Predictors regular patterns (x, x+8, x+16, ...)

• When to prefetch?
  – On every reference → lots of lookup/prefetcher overhead
  – On every miss → patterns filtered by caches
  – On prefetched-data hits (positive feedback)

• Where to put prefetched data?
  – *Prefetch buffers*
  – Caches
What’s Inside Today’s Chips

• Data L1
  – PC-localized stride predictors
  – Short-stride predictors within block $\rightarrow$ prefetch next block

• Instruction L1
  – Predict future PC $\rightarrow$ prefetch

• L2
  – Stream buffers
  – Adjacent-line prefetch