Memory Prefetching

Nima Honarmand
The memory wall


Today: 1 mem access ≈ 500 arithmetic ops

How to reduce memory stalls for existing SW?
Techniques We’ve Seen So Far

• Use Caching to reduce memory latency

• Use wide out-of-order execution to hide memory latency
  – By overlapping misses with other useful work
  – Cannot efficiently go much wider than several instructions

• Neither is enough for server applications
  – Not much spatial locality (mostly accessing linked data structures)
  – Not much ILP and MLP
  → Server apps spend 50-66% of their time stalled on memory

Need a different strategy
Prefetching (1)

• Fetch data ahead of demand

Main challenges:

• Knowing “what” to fetch
  – Fetching useless blocks wastes resources

• Knowing “when” to fetch
  – Too early $\rightarrow$ clutters storage (or gets thrown out before use)
  – Fetching too late $\rightarrow$ defeats purpose of “pre”-fetching

Prefetching must be **accurate** and **timely**
Prefetching (2)

• Without prefetching:

• With prefetching:

• Or:

Prefetching must be **accurate** and **timely**
Types of Prefetching

• Software
  – By compiler
  – By programmer

• Hardware
  – Next-Line, Adjacent-Line
  – Next-N-Line
  – Stream Buffers
  – Stride
  – Localized (PC-based)
  – Pointer
  – Correlation
  – ...
Software Prefetching (1)

• Prefetch data using explicit instructions
  – Inserted by compiler and/or programmer

• Put prefetched value into...
  – Register (*binding prefetch*)
    • Also called “hoisting”
    • Basically, just moving the load instruction up in the program
  – Cache (*non-binding prefetch*)
    • Requires ISA support
    • May get evicted from cache before demand
Software Prefetching (2)

- Hoisting is prone to many problems:
  - May prevent earlier instructions from committing
  - Must be aware of dependences
  - Must not cause exceptions not possible in the original execution
  - Increases register pressure for the compiler

- Using a **prefetch instruction** can avoid all these problems
Software Prefetching (3)

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

• Prefetch instruction reads the containing block from the memory and puts it in the cache
Software Prefetching (4)

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

• Cons:
  – Prefetch instructions increase code footprint
    • May cause more I$ misses, code alignment issues
  – 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 (block B in prev. slides)
Hardware Prefetching

• Hardware monitors memory accesses
  – Looks for common patterns → Makes predictions

• Predicted addresses are placed into *prefetch queue*
  – Queue is checked when no demand accesses waiting

• Prefetches look like READ requests to the mem. hierarchy

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

No need to change software
Hardware Prefetcher Design Space

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

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

• Where to put prefetched data?
  – Prefetch buffers
  – Caches
Prefetching at Different Levels

• Real CPUs have multiple prefetchers w/ different strategies
  – 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
  - Assumes spatial locality
  - Should stop at physical (OS) page boundaries (why?)

- Can often be done efficiently
  - 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 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
Stride Prefetching (1)

- Access patterns often follow a *stride*
  - Example 1: Accessing column of elements in a matrix
  - Example 2: Accessing elements in array of `struct` s

- Detect stride $S$, prefetch depth $N$
  - Prefetch $X+S$, $X+2S$, ..., $X+NS$
Stride Prefetching (2)

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

• How to tell the diff. between \( A[i] \rightarrow A[i+1] \) and \( X \rightarrow Y \)?
  – Wait until you see the *same stride* a few times
  – Can vary prefetch depth based on confidence
  • More consecutive strided accesses \( \rightarrow \) higher confidence
“Localized” Stride Prefetchers (1)

• What if multiple strides are interleaved?
  – No clearly-discernible stride

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

  Y = A + X?

  A, X, Y, A+S, X+S, Y+S, A+2S, X+2S, Y+2S, ...

  (A+S-Y)
  (A+S-Y)
  (A+S-Y)

• Observation: Accesses to structures usually localized to an instruction

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

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

PC: 0x409A34  Load R1 = [R2]
PC: 0x409A38  Load R3 = [R4]
PC: 0x409A40  Store [R6] = R5

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

If confident about the stride (count > C_min), prefetch (A+4S)
Stream Buffers (1)

- Used to avoid cache pollution caused by deep prefetching

- Each buffer holds one stream of sequentially prefetched lines
  - Keep next-N available in buffer

- On a load miss, check the head of all buffers
  - if match, pop the entry from FIFO, fetch the N+1\textsuperscript{st} line into the buffer
  - if miss, allocate a new stream buffer (use LRU for recycling)
Stream Buffers (2)

• Can incorporate stride prediction mechanisms to support non-unit-stride streams

• 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
Other Prefetch Patterns

• Sometimes accesses are highly predictable, but no strides
  – Linked data structures (e.g., lists or trees)
Pointers usually “look different”

Data filled on cache miss

(512 bits of data)

Nope
Nope
Nope
Nope
Nope
Maybe!
Maybe!
Nope
Nope

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)

```c
struct bintree_node_t {
    int data1;
    int data2;
    struct bintree_node_t * left;
    struct bintree_node_t * right;
};
```
Pointer Prefetching (2)

- Relatively cheap to implement
  - Don’t need extra hardware to store patterns
  - But can fetch a lot of junk

- Limited **lookahead** makes timely prefetches hard
  - Can’t get next pointer until fetched data block

Stride Prefetcher:

```
<table>
<thead>
<tr>
<th>Access Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
<tr>
<td>X+S</td>
</tr>
<tr>
<td>X+2S</td>
</tr>
</tbody>
</table>
```

Pointer Prefetcher:

```
<table>
<thead>
<tr>
<th>Access Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>
```
Pair-wise Temporal Correlation (1)

• Accesses exhibit *temporal correlation*
  – If E followed D in the past \(\rightarrow\) if we see D, prefetch E
  – Somewhat similar to history-based branch prediction

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

• Like branch prediction, longer history can provide more accuracy
  – And increases training time

• Use history hash for lookup
  – E.g., XOR the bits of the addresses of the last K accesses

DFS traversal: ABDBEBACFCGCA

Better accuracy 😊, larger storage cost 😞
Example: Prefetchers in Intel Sandy Bridge

• Data L1
  1) PC-localized stride prefetcher
  2) Next line prefetcher
     • Only on an ascending access to very recently loaded data

• L2
  3) Spatial prefetcher: Prefetch the cache line which pairs with current one to make a 128-byte aligned chunk
  4) Stream prefetcher: detects streams of requests made by L1 (I and D) and prefetches lines down the stream
     • # of lines to prefetch depends on # of outstanding requests from L1
     • Far lines are only prefetched to L3; closer ones are brought to L2

See Intel Architectures Optimization Reference Manual for more details