Neighborhood Prefetching

David M. Koppelman

Electrical & Computer Engineering Department
Louisiana State University
Baton Rouge, LA U.S.A.
Hardware Prefetching

Prefetch:

    Fetching a cache line in anticipation of its use.

Motivation:

    Cache misses becoming larger component of execution time.
    Multiprocessors add interconnect and protocol delays.

Challenges:

    Prefetching addresses that will be used. (Easy)
    Avoiding addresses that won’t be used. (Hard)
Address Determination in Some Existing Schemes

Sequential

Infinite sequential sequence.

Sequential sequence indicated in bold.

\[ 100, 3000, 101, 2000, 3000, 102, 103, 6, \ldots \]

Stride

Infinite stride sequence.

Stride 10 sequence indicated in bold.

\[ 100, 3000, 110, 2000, 3000, 120, 130, 6, \ldots \]

Real address reference sequences are finite, guaranteeing useless prefetches.
Unneeded Prefetch Problems

Unneeded Prefetch Problems in All Systems

Resource Consumption. (Cache ports, interconnect bandwidth, etc.)

Eviction of useful lines.

Unneeded Prefetch Problems in Multiprocessors

Adds to false sharing.
Neighborhood Prefetch

Advantages

Larger variety of address sequences than stride.

Less tendency to prefetch unneeded lines.
Execution of following code fragment …

! Iteration 1: r2 = 0x2000, Line Size 0x100 Characters
!
A: load r1, [r2]
   ...
B: load r3, [r2+0x500]
   ...
C: load r4, [r2-0x700]

… generates …

Address Seq.: 0x2000, ..., 0x2500, ..., 0x1900, ...
Neighborhood Prefetch Terminology

Term:

Base: Address used as reference. 0x2000

Initiator: Instruction accessing base address. A: load r1, [r2]

Offset: Distance from base, in lines. 0, ..., +5, ..., -7

Neighborhood: Set of small-magnitude offsets. \{0, +5, -7\}
Consider ...

! Iteration 1: r2 = 0x2000 (Base), Line Size 0x100 Characters

A: load r1, [r2] !(Initiator)
   ...
B: load r3, [r2+0x500]
   ...
C: load r4, [r2-0x700]

... generates ...

Address Seq.: 0x2000, ..., 0x2500, ..., 0x1900, ...

Neighborhood: \{0, +5, −7\}

Notice that the neighborhood ...
Consider ... and

! Iteration 2: \( r2 = 0x8000 \) (Base), Line Size 0x100 Characters
!
A: load \( r1, [r2] \) ! (Initiator)
  ...
B: load \( r3, [r2+0x500] \)
  ...
C: load \( r4, [r2-0x700] \)

... generates ...

Address Seq.: \( 0x8000, \ldots, 0x8500, \ldots, 0x7900, \ldots \)

Neighborhood: \( \{0, +5, -7\} \)

Notice that the neighborhood ... is the same in both cases.

Base address and neighborhood used to construct prefetch addresses.
Neighborhood Prefetch Outline

Hardware monitors addresses that miss the cache.

Hardware determines neighborhoods and stores them in PC-indexed table.

Table checked on miss; if entry found used for prefetching.
Neighborhood Prefetching Advantages

Superset (loosely) of existing schemes.

Can detect sequential patterns: \{0, 1, 2, 3, \ldots\}.

Can detect stride patterns: \{0, 10, 20, 30, \ldots\}.

Can detect arbitrary patterns: \{0, 10, 20, 50\}.

Better at avoiding unneeded and harmful prefetches.

Prefetch candidates based on past usage by instruction.

Other schemes inevitably fetch unneeded, possibly harmful, lines.

Particularly important in multiprocessors: less false sharing.
**Main Parts**

*Recent Miss Table (RMT)*—for determining neighborhoods.

*Neighborhood History Table (NHT)*—stores neighborhoods.

Prefetch Controller—handles prefetches.
Recent Miss Table (RMT)

Entries for recently encountered neighborhoods.

Each entry holds:

- Base Address.
- Offsets.
- Initiating instruction.
- And other data.

On miss update existing entry or create new ones.

Entries occasionally moved to neighborhood history table.
Neighborhood History Table

Entry for memory access instructions.

Retrieved on miss using address of missing instruction.

Used to construct prefetch addresses.
Evaluation

Evaluated by execution driven simulation using Proteus.

Machine model:

Sixteen-node multiprocessor.

Full-map directory-based cache coherence.

In-order execution, four-way issue, SPARC ISA.

Virtual memory, nonblocking stores, blocking loads.

Mesh topology.

Execution Bottlenecks

Cache and Network Latency (all accesses)

Hot Spots at Memories (application dependent)
## Base Configuration Parameters

<table>
<thead>
<tr>
<th>Simulation Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Size</td>
<td>16 processors</td>
</tr>
<tr>
<td>Network Topology</td>
<td>$4 \times 4$ mesh</td>
</tr>
<tr>
<td>VM Page Size</td>
<td>$2^{12}$ bytes</td>
</tr>
<tr>
<td>TLB Capacity</td>
<td>64 entries</td>
</tr>
<tr>
<td>TLB Replacement</td>
<td>LRU, fully assoc.</td>
</tr>
<tr>
<td>Cache Size</td>
<td>$2^7$ sets</td>
</tr>
<tr>
<td>Cache Associativity</td>
<td>8, LRU Repl.</td>
</tr>
<tr>
<td>Cache Line Size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L1 Cache Hit Latency</td>
<td>1 cycle</td>
</tr>
<tr>
<td>L2 Cache Hit Latency</td>
<td>7 cycles</td>
</tr>
<tr>
<td>Total L2 Miss Latency</td>
<td>50 (min), 135 (typ)</td>
</tr>
<tr>
<td>Directory Size</td>
<td>full map</td>
</tr>
<tr>
<td>Completion Buffer</td>
<td>5 stores</td>
</tr>
<tr>
<td>Raw Memory Latency</td>
<td>10 cycles</td>
</tr>
<tr>
<td>Protocol Message Size</td>
<td>8 bytes (plus data)</td>
</tr>
<tr>
<td>Network Interface Width</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Network Link Width</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Hop Latency</td>
<td>20 cycles (plus waiting)</td>
</tr>
<tr>
<td>Neighborhood Size</td>
<td>16 offsets</td>
</tr>
<tr>
<td>NHT Size</td>
<td>256 entries</td>
</tr>
<tr>
<td>RMT Size</td>
<td>8 entries</td>
</tr>
</tbody>
</table>
SPLASH-2 Benchmarks Used

**Cholesky - Processor States**

**FFT: Processor States (65536 elements)**

**LU: Processor States (128 x 128)**

**Processor States, Radix 262,144 Keys**

**Processor States: Barnes, 16384 Particles**

**Processor States: FMM, 16384 Particles**

**Processor States: Ocean, 258 x 258**

**Processor States: Water N Squared, 512 Molec.**

Formatted 0:02, 17 October 2000 from pren/ppact.
Ran SPLASH-2 Benchmarks

Compared:

- Neighborhood Prefetch (Cache Size $\frac{7}{8}$ Other Systems)
- Adaptive Sequential Prefetch (Preupgrades, Tuned for System)
- Conventional System

Experiments Presented

- Base: 64KB: 8 way assoc. $\times$ 64 byte lines $\times$ 128 sets.
- Cache Size: 8KB ($2^4$ sets) to 1MB ($2^{11}$ sets).
- Line Size: 8 bytes to 1024 bytes.
Base: Prefetch Outcome

Pref. Outcome Per Original Miss*

-0.8 -0.6 -0.4 -0.2 0.0 0.2 0.4 0.6 0.8

Chl FFT LU Rdx Bar FMM Ocn WSQ Avg

Unused +Miss +Inv Hit Sl Hit Sl Upg

Base: Normalized Stall and Execution Time

![Graph showing normalized stall and execution time for various benchmarks.

- Access Stall Time
- Execution Time
- Neighb. Pref.
- Adap. Seq. Pref.

Chl, FFT, LU, Rdx, Bar, FMM, Ocn, WSQ, Avg]
Base: Miss Latency

Normalized Miss Latency

- Chl
- FFT
- LU
- Rdx
- Bar
- FMM
- Ocn
- WSQ
- Avg

- Miss
- Upgrade
- No Pref.
- Neighb. Pref.
- Adap. Seq. Pref.
Cache Size: Normalized Stall and Execution Time

![Graph showing normalized stall and execution times]
Cache Size: Prefetch Outcome

Pref. Outcome Per Original Miss*
Line Size: Execution Time

![Graph showing execution time versus line size with three different preferences: No Pref., Neighb. Pref., and Adap. Seq. Pref. The graph includes a legend with black circles for No Pref., blue diamonds for Neighb. Pref., and orange triangles for Adap. Seq. Pref. The x-axis represents line size (3 to 10), and the y-axis represents execution time in megacycles (0 to 30). The data points are plotted for each preference, illustrating the performance differences across line sizes.]
Line Size: Normalized Stall and Execution Time

Access Stall Time
Execution Time
Neighb. Pref.
Adap. Seq. Pref.
Line Size: Miss Ratio


Level Two Miss Ratio

Formatted 0:02, 17 October 2000 from pren.ppact.
Conclusions

Neighborhood Prefetch handles wider variety of reference patterns.

Less prefetching of unneeded lines.

Reduces miss ratio by nearly 50%.

Reduces execution time by 10% or more.

Performance better than adaptive sequential prefetch . . .

. . . though implementation considerably more complex.

Future Work: Prefetch on Hit

Wallpaper prefetch? (Repeat neighborhoods.)