Tolerating Latency Through Prefetching
The Memory Latency Problem

- ↑ processor speed >> ↑ memory speed
- latency even worse for multiprocessors
- caches are not a panacea
Memory Latency in Multiprocessors

- Architecture resembling DASH multiprocessor.
  - latency = 1 : 15 : 30 : 100 processor cycles
  - 16 processors

6 of 8 spend 50% of time stalled for memory.
Overview

- Tolerating Memory Latency
- Prefetching Classification
- Compiler-Based Prefetching
- Performance Results
- Concluding Remarks
Coping with Memory Latency

Reduce Latency:

- Caches, local memory, low-latency network
- Locality optimizations

Tolerate Latency:

- Relaxed memory consistency models
  - permits buffering and pipelining of accesses
- Prefetching
  - move data close to the processor before it is needed
- Context switching
  - switch contexts on long-latency operations

☞ Complementary -- not mutually exclusive
Benefits of Prefetching

- prefetch early enough
  - completely hides latency

- issue prefetches in blocks
  - pipelining
  - only first reference suffers

- prefetch with ownership
  - reduces write latency
Prefetching Classification

- Non-binding vs. Binding prefetches

**Binding:** value of a later “real” reference is bound when prefetch is performed.
- restricts legal issue
- additional high-speed storage needed

**Non-Binding:** prefetch brings data closer, but value is not bound until later “real” reference.
- data remains visible to coherence protocol
- prefetch issue not restricted

```c
prefetch(&x);
...
LOCK(L);
x = x + 1;
UNLOCK(L);
```
Prefetching Classification (continued)

- *hardware controlled* vs. *software controlled*

**Hardware Controlled:** (no hints from software)

- multi-word cache blocks
- streaming buffers
- instruction look-ahead and stride detection

**Software Controlled:** (explicit prefetch instructions)

- prefetches inserted by programmer
- prefetches inserted by runtime system
- prefetches inserted by compiler
Hardware Controlled Prefetching

• **Large Cache Blocks:**
  - Most machines already exploit such prefetching
  - Great for codes with unit-stride accesses
  - Problems of increased traffic and false-sharing in multiprocessors

• **Streaming Buffers:**
  - Concept: Fetch a subsequent cache line, when current one is touched
  - Can completely hide latency for codes with unit-stride accesses
  - Does not help with non-unit stride access codes
• **Instruction Lookahead and Stride Detection Hardware:**
  
  - Example: Scheme by Baer and Chen (Supercomputing ‘91)
  
  - Use *Branch Prediction Table* to compute *Look-Ahead PC* (LA-PC) value
  
  - LA-PC used to lookup *Reference Prediction Table* (tag, prev-addr, stride, state)
  
  - State of entry in RPT can be *initial*, *transient*, *steady*, or *no-prediction*
  
  - Advantages:
    
    - Can handle non-unit stride accesses
    
    - No requirements of software and no direct instruction overhead
  
  - Limitations:
    
    - Complex hardware (BPT, RPT, ...) (TLB for VA --> PA)
    
    - Branch-prediction accuracy can limit amount of lookahead
    
    - Issues unnecessary prefetches, busying cache tags (e.g., spatial locality)
    
    - Can not handle indirections (e.g., A[index[i]])
Context Switching

- switch between contexts to hide long-latency operations

Advantages:

- handles complex access patterns
- no software support required

Disadvantages:

- requires additional parallel threads
- overheads in switching contexts
- requires substantial hardware support

Example:

[Diagram showing context switching with timelines and operations]
## Overall Approach to Coping with Latency

<table>
<thead>
<tr>
<th>Technique</th>
<th>Exploits</th>
<th>Hardware Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locality Optimizations</td>
<td>ability to reorder loop iterations</td>
<td>none</td>
</tr>
<tr>
<td>Software-Controlled Prefetching</td>
<td>parallelism within a single thread</td>
<td>minimal</td>
</tr>
<tr>
<td>Context Switching</td>
<td>parallelism across multiple threads</td>
<td>substantial</td>
</tr>
</tbody>
</table>
Compiler Based Prefetching
Prefetching Concepts

• possible only if addresses can be determined ahead of time
• coverage factor: fraction of misses that are prefetched
• unnecessary if data is already in the cache
• effective if data is in the cache when later referenced

Analysis: what to prefetch

• maximize coverage factor
• minimize unnecessary prefetches

Scheduling: when/how to schedule prefetches

• maximize effectiveness
• minimize overhead per prefetch
Compiler Algorithm

Analysis what to prefetch
  • Locality Analysis

Scheduling when/how to issue prefetches
  • Loop Splitting
  • Software Pipelining
Data Locality Example

for (i=0; i<3; i++)
  for (j=0; j<100; j++)
    A[i][j] = B[j][0] + B[j+1][0];

○ Cache Hit
● Cache Miss

A[i][j]

B[j][0]

B[j+1][0]

Spatial

Temporal

Group
Software Pipelining

Iterations Ahead $\left\lfloor \frac{l}{s} \right\rfloor$

where $l =$ memory latency, $s =$ shortest path through loop body.

<table>
<thead>
<tr>
<th>Original Loop</th>
<th>Software Pipelined Loop (5 iterations ahead)</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=0; i&lt;100; i++) a[i] = 0;</td>
<td>for (i=0; i&lt;5; i++) /* Prolog */ prefetch(&amp;a[i]);</td>
</tr>
<tr>
<td></td>
<td>for (i=0; i&lt;95; i++) /* Steady State */ prefetch(&amp;a[i+5]); a[i] = 0;</td>
</tr>
<tr>
<td></td>
<td>for (i=95; i&lt;100; i++) /* Epilog */ a[i] = 0;</td>
</tr>
</tbody>
</table>
Software Pipelining for Indirections

Original Loop
for (i=0; i<100; i++)
    sum += A[index[i]];

Software Pipelined Loop
(5 iterations ahead)

for (i=0; i<5; i++)  /* Prolog 1 */
    prefetch(&index[i]);

for (i=0; i<5; i++)  /* Prolog 2 */
    prefetch(&index[i+5]);
prefetch(&A[index[i]]);

for (i=0; i<90; i++)  /* Steady State */
prefetch(&index[i+10]);
prefetch(&A[index[i+5]]);
sum += A[index[i]];

for (i=90; i<95; i++)  /* Epilog 1 */
prefetch(&A[index[i+5]]);
sum += A[index[i]];

for (i=95; i<100; i++)  /* Epilog 2 */
sum += A[index[i]];
Example Revisited

Original Code

```c
for (i = 0; i < 3; i++)
    for (j = 0; j < 100; j++)
        A[i][j] = B[j][0] + B[j+1][0];

Code with Prefetching

```c
for (i = 0; i<6; i+= 2) {
    prefetch(&B[i+1][0]);
    prefetch(&B[i+2][0]);
    prefetch(&A[0][i+1]);
}
```

```c
for (j = 0; j<94; j += 2) {
    prefetch(&B[j+7][0]);
    prefetch(&B[j+8][0]);
    prefetch(&A[0][j+7]);
    A[0][j] = B[j][0]+B[j+1][0];
    A[0][j+1] = B[j+1][0]+B[j+2][0];
}
```

```c
for (i = 1; i<3; i++)
    for (j = 0; j<6; j += 2) {
        prefetch(&A[i][j+1]);
    }
```

```c
for (j = 0; j<94; j += 2) {
    prefetch(&A[i][j+7]);
    A[i][j] = B[i][0]+B[i+1][0];
    A[i][j+1] = B[i+1][0]+B[i+2][0];
}
```

```c
for (j = 94; j<100; j += 2) {
    A[i][j] = B[j][0]+B[j+1][0];
    A[i][j+1] = B[j+1][0]+B[j+2][0];
}
```
Prefetching for Multiprocessors

- non-binding vs. binding prefetches
  - use non-binding since data remains coherent until accessed
    
    ```
    prefetch(&x);
    ...
    LOCK(L);
    x = x + 1;
    UNLOCK(L);
    ```

  - no restrictions on when prefetches can be issued

- dealing with coherence misses
  - localized space takes explicit synchronization into account

- further optimizations
  - prefetch in exclusive-mode in read-modify-write situations
Multiprocessor Results

- Memory stalls reduced by 50% to 90%
- Synchronization stalls reduced in some cases
- 4 of 5 have speedups over 45%
Effectiveness of Software Pipelining

- Large pf-miss ineffective scheduling
  - prefetched data still found in secondary cache
Exclusive-Mode Prefetching

- message traffic reduced by 7% to 29%
- release consistency: write latency already hidden
Limitations of Compiler Algorithm

- WATER: needs procedure inlining across separate files
- BARNES: traverses an octree structure
- PTHOR: lots of pointers, very complex control flow
## Overall Approach to Coping with Latency

- techniques are complementary
  - best to combine prefetching with locality optimizations
- software-controlled prefetching is quite successful

<table>
<thead>
<tr>
<th>Technique</th>
<th>Exploits</th>
<th>Hardware Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locality Optimizations</td>
<td>ability to reorder loop iterations</td>
<td>none</td>
</tr>
<tr>
<td>Software-Controlled Prefetching</td>
<td>parallelism within a single thread</td>
<td>minimal</td>
</tr>
<tr>
<td>Context Switching</td>
<td>parallelism across multiple threads</td>
<td>substantial</td>
</tr>
</tbody>
</table>