Tolerating Latency Through Prefetching

The Memory Latency Problem

• ↑ processor speed >> ↑ memory speed
• latency even worse for multiprocessors
• caches are not a panacea

Uniprocessor Cache Performance on Scientific Code

• Applications from SPEC, SPLASH, and NAS Parallel.
• Memory subsystem typical of MIPS R4000 (100 MHz):
  - 8K / 256K direct-mapped caches, 32 byte lines
  - miss penalties: 12 / 75 cycles

8 of 13 spend > 50% of time stalled for memory.

Overview

• Tolerating Memory Latency
• Prefetching Classification
• Programmer Inserted Prefetching
• Compiler-Based Prefetching
• Increasing Scope of Prefetching
• 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)
  - prefetched inserted by programmer
  - prefetched inserted by runtime system
  - prefetched 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

Software Controlled Prefetching

• **Programmer assisted prefetching:**
  - Understand complexity of doing it by hand
  - Understand what is the best that we can do

• **Compiler based prefetching:**
  - Understand what we can automate

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:
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</td>
<td>parallelism within a single thread</td>
<td>minimal</td>
</tr>
<tr>
<td>Prefetching</td>
<td>parallelism across multiple threads</td>
<td>substantial</td>
</tr>
</tbody>
</table>

Programmer Assisted Prefetching

**Experimental Framework**

- Processor
- Cache
- Memory
- Directory Memory & Controller
- Interconnection Network
- Processor
- Cache
- Memory
- Directory Memory & Controller

**System Architecture**

- Processor Environment
  - Processor
  - Primary Cache
  - Write Buffer
  - Secondary Cache

- Processor
  - Primary Cache
  - Write Buffer

**Benchmarks**

- **MP3D**: particle-based simulator for wind tunnel
  - 10,000 particles
  - 64x8x8 space array
  - 5 time steps

- **PTHOR**: digital logic simulator
  - small RISC processor
  - 11,000 two-input gates
  - simulated 10 clock cycles

- latency = 1 : 12 : 22 : 61 : 80 processor cycles
- 16 processors
- detailed simulation, contention modeled
**MP3D Case Study**

pf1: particles, same iteration  
pf2: particles+space cells, same iteration  
pf3: particles+space cells, pipelined  
pf4: pf3+ time-step boundary references  

- best speedup = 1.86 (invals due to small data set and prefetch only into cluster cache)

**PTHOR Case Study**

pf1: entire element record, rd-ex  
pf2: reorganized record + heads of lists  
pf3: pf2 + other things based on profiling  

- best speedup = 1.14 (linked lists are difficult to prefetch)

---

**Prefetching into Primary Cache**

- improves primary cache hit rate (82% --> 97% for MP3D, 75% --> 81% for PTHOR)  
- benefits of prefetching into primary cache outweigh overhead of tag-busy stalls  
- significantly more speedup (1.86 --> 2.53 for MP3D, 1.14 --> 1.30 for PTHOR)

---

**Compiler Based Prefetching**
Compiler-Based Prefetching Goals

- Domain of Applicability
- Performance Improvement
  - maximize benefit
  - minimize overhead

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

Reducing Prefetching Overhead

- instructions to issue prefetches
- extra demands on memory system

Important to minimize unnecessary prefetching
Compiler Research on Dense Matrix Prefetching

State of the Art:

• **Callahan, Kennedy, and Porterfield** (ASPLOS ’91)
  - prefetch one iteration ahead
  - reduce unnecessary prefetching
• **Klaiber and Levy** (ISCA ’91)
  - prefetch multiple iterations ahead

This Work:

• new algorithm for reducing unnecessary prefetching
• full compiler implementation
• combine prefetching with locality optimizations

---

**Compiler Algorithm**

Analysis: what to prefetch
• Locality Analysis

Scheduling: when/how to issue prefetches
• Loop Splitting
• Software Pipelining

---

**Data Locality Example**

```c
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

- Spatial
- Temporal

- Group

---

**Reuse Analysis**

Goal: find *intrinsic* data reuse

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

Vector space representation.

\[
B[j][0] \to B \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}
\]

Reuse between iterations \((i, j)\) and \((i', j')\) whenever:

\[
\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}
\]

\[\Rightarrow\] Find the nullspace of \( \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \) (i.e. \( i \) loop)
Localized Iteration Space

**Goal:** given finite cache, when does reuse result in locality

```
for (i=0; i<3; i++)
  for (j=0; j<8; j++)
    A[i][j] = B[j][0]+B[j+1][0];

for (i=0; i<3; i++)
  for (j=0; j<10000; j++)
    A[i][j] = B[j][0]+B[j+1][0];
```

- Cache Hit
- Cache Miss

**Localized:** both \( i \) and \( j \) loops

**Localized:** \( j \) loop only

\( \iff \) loop localized if accesses less data than effective cache size

---

Prefetch Predicate

<table>
<thead>
<tr>
<th>Locality Type</th>
<th>Miss Instance</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Every Iteration</td>
<td>True</td>
</tr>
<tr>
<td>Temporal</td>
<td>First Iteration</td>
<td>( i = 0 )</td>
</tr>
<tr>
<td>Spatial</td>
<td>Every ( l ) iterations ((l = cache line size))</td>
<td>((i \mod l) = 0)</td>
</tr>
</tbody>
</table>

**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

```
B[j+1][0]
```

**Reference**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Locality</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[i][j]</td>
<td>[( i ) ( j )] = [none spatial]</td>
<td>((j \mod 2) = 0)</td>
</tr>
<tr>
<td>B[j+1][0]</td>
<td>[( i ) ( j )] = [temporal none]</td>
<td>( i = 0 )</td>
</tr>
</tbody>
</table>

---

Compiler Algorithm

**Analysis:** what to prefetch
- Locality Analysis

**Scheduling:** when/how to issue prefetches
- Loop Splitting
- Software Pipelining

---

Loop Splitting

- Decompose loops to isolate cache miss instances.
  - cheaper than inserting IF statements

<table>
<thead>
<tr>
<th>Locality Type</th>
<th>Predicate</th>
<th>Loop Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>True</td>
<td>None</td>
</tr>
<tr>
<td>Temporal</td>
<td>( i = 0 )</td>
<td>Peel loop ( i )</td>
</tr>
<tr>
<td>Spatial</td>
<td>((i \mod l) = 0)</td>
<td>Unroll loop ( i ) by ( l )</td>
</tr>
</tbody>
</table>

- Apply transformations recursively for nested loops.
- Suppress transformations when loops become too large.
  - avoid code explosion
**Software Pipelining**

Iterations Ahead = \[
\left\lfloor \frac{s}{l} \right\rfloor
\]

where \(l\) = memory latency, \(s\) = shortest path through loop body.

Software Pipelined Loop

Original Loop

\[
\begin{align*}
&\text{for } (i=0; i < 100; i++)
&\text{for } (i=0; i < 5; i++) \\
&a[i] = 0;
&\text{prefetch}(&a[i]);
&\text{for } (i=0; i < 95; i++) \\
&A[i][j] = 0;
&B[j+1][0];
&\text{prefetch}(&A[i+5]);
&\text{for } (i=95; i < 100; i++) \\
&a[i] = 0;
&A[i+1] = 0;
&B[j+2][0];
&\text{prefetch}(&A[0][j+7]);
\end{align*}
\]

Example Revisited

Original Code

\[
\begin{align*}
&\text{for } (i = 0; i < 3; i++)
&\text{for } (j = 0; j < 100; j++) \\
&A[i][j] = B[j][0] + B[j+1][0];
&\text{prefetch}(&A[i][j+1]);
&\text{for } (j = 0; j < 94; j += 2) \\
&A[j][j+1];
&B[j+2][0];
&\text{prefetch}(&A[0][j+7]);
&B[j+3][0];
&\text{prefetch}(&A[0][j+8]);
&\text{for } (j = 94; j < 100; j++) \\
&A[j][j+1];
&B[j+2][0];
&\text{prefetch}(&A[0][j+8]);
&A[j+1][j+1];
&B[j+3][0];
&\text{prefetch}(&A[0][j+7]);
&\text{for } (i = 1; i < 3; i++) \\
&\text{for } (j = 0; j < 6; j += 2) \\
&A[i][j+1];
&B[j+2][0];
&\text{prefetch}(&A[i][j+7]);
&A[i][j+1];
\end{align*}
\]

Code with Prefetching

\[
\begin{align*}
&\text{for } (i = 0; i < 3; i++)
&\text{for } (j = 0; j < 100; j++) \\
&A[i][j] = B[j][0] + B[j+1][0];
&\text{prefetch}(&A[i][j+1]);
&\text{for } (j = 0; j < 94; j += 2) \\
&A[j][j+1];
&B[j+2][0];
&\text{prefetch}(&A[0][j+7]);
&B[j+3][0];
&\text{prefetch}(&A[0][j+8]);
&\text{for } (j = 94; j < 100; j++) \\
&A[j][j+1];
&B[j+2][0];
&\text{prefetch}(&A[0][j+8]);
&A[j+1][j+1];
&B[j+3][0];
&\text{prefetch}(&A[0][j+7]);
&\text{for } (i = 1; i < 3; i++) \\
&\text{for } (j = 0; j < 6; j += 2) \\
&A[i][j+1];
&B[j+2][0];
&\text{prefetch}(&A[i][j+7]);
&A[i][j+1];
\end{align*}
\]

Experimental Framework (Uniprocessor)

Architectural Extensions:

- Prefetching support:
  - lockup-free caches
  - 16-entry prefetch issue buffer
  - prefetch directly into both levels of cache

- Contention:
  - memory pipelining rate = 1 access every 20 cycles
  - primary cache tag fill = 4 cycles

- Misses get priority over prefetches

Simulator:

- detailed cache simulator driven by pixified object code.

Experimental Results (Dense Matrix Uniprocessor)

- Performance of Prefetching Algorithm
  - Locality Analysis
  - Software Pipelining

- Interaction with Locality-Optimizer
Performance of Prefetching Algorithm

• memory stalls reduced by 50% to 90%
• instruction and memory overheads are typically low
  6 of 13 have speedups over 45%

Effectiveness of Locality Analysis

Selective vs. Indiscriminate prefetching:
• similar reduction in memory stalls
• significantly less overhead
  6 of 13 have speedups over 20%

Effectiveness of Locality Analysis (Continued)

• fewer unnecessary prefetches
• comparable coverage factor
• reduction in prefetches ranges from 1.5 to 21 (average = 6)

Effectiveness of Software Pipelining

• Large pf-miss \(\Rightarrow\) ineffective scheduling
  - conflicts replace prefetched data (CHOLSKY, TOMCATV)
  - prefetched data still found in secondary cache
Interaction with Locality Optimizer

- locality optimizations reduce number of cache misses
- prefetching hides any remaining latency
  - best performance through a combination of both

Prefetching Indirections

```c
for (i=0; i<n; i++)
    sum += A[index[i]];  
```

**Analysis:** what to prefetch
- both dense and indirect references
- difficult to predict whether indirections hit or miss

**Scheduling:** when/how to issue prefetches
- modification of software pipelining algorithm

Domain of Applications

Software Pipelining for Indirections

**Original Loop**

```c
for (i=0; i<100; i++)
    sum += A[index[i]];  
```

**Software Pipelined Loop (5 iterations ahead)**

```c
for (i=0; i<5; i++) /* Prolog 1 */
    sum += A[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]];  
```
Indirection Prefetching Results

(N = No Prefetching, D = Dense-Only Prefetching, I = Indirection Prefetching)

- larger overheads in computing indirection addresses
- significant overall improvements for IS and CG

Large-Scale Shared-Memory Multiprocessors

- distributed main memory
- single address space
- hardware-coherent caches

Domain of Applications

- Large
- Sparse Matrix
- Parallel Matrix-Based

Memory Latency in Multiprocessors

- Architecture resembling DASH multiprocessor.
  - latency = 1 : 15 : 30 : 100 : 130 processor cycles
  - 16 processors

  6 of 8 spend > 50% of time stalled for memory.
Prefetching for Multiprocessors

- *non-binding vs. binding* prefetches
  - use non-binding since data remains coherent until accessed later

  ```c
  prefetch (&x);
  ```

  ```c
  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 $\Rightarrow$ ineffective scheduling
  - prefetched data still found in secondary cache

---

Exclusive-Mode Prefetching

- message traffic reduced by 7% to 29%
- release consistency $\Rightarrow$ write latency already hidden
Summary of Results

**Dense Matrix Code:**
- eliminated 50% to 90% of memory stall time
- overheads remain low due to prefetching selectively
- significant improvements in overall performance (6 over 45%)

**Indirections, Sparse Matrix Code:**
- expanded coverage to handle some important cases

**Parallel Matrix-Based Code:**
- large performance improvements (28% to 120% faster)
- exclusive-mode prefetching reduces message traffic
- successfully overlapping computation and communication

Overview

Tolerating Memory Latency

Prefetching Compiler Algorithm and Results

Increasing Scope of Prefetching

Implications of These Results

Limitations of Compiler Algorithm

<table>
<thead>
<tr>
<th>Normalized Execution Time</th>
<th>prefetched memory overhead</th>
<th>synchronization</th>
<th>memory access stalls</th>
<th>instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>140</td>
<td>100</td>
<td>100</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>120</td>
<td>100</td>
<td>92</td>
<td>81</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>80</td>
<td>100</td>
<td>81</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>60</td>
<td>100</td>
<td>81</td>
<td>0</td>
<td>81</td>
</tr>
<tr>
<td>40</td>
<td>100</td>
<td>81</td>
<td>0</td>
<td>81</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>81</td>
<td>0</td>
<td>81</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>81</td>
<td>0</td>
<td>81</td>
</tr>
</tbody>
</table>

(N = No Prefetching, H = Hand-Inserted Prefetching)

- WATER: needs procedure inlining across separate files
- BARNES: traverses an octree structure
- PTHOR: lots of pointers, very complex control flow

Increasing Scope of Prefetching Algorithm

Pushing Static Analysis Further:
- global, generalized locality analysis
- understanding communication for multiprocessors

Incorporating Dynamic Information into Compilation:
- control-flow feedback
- memory behavior feedback

Using Dynamic Information at Runtime:
- check loop bounds, data alignment, hardware miss counters
### Profiling Feedback Example

```c
foo(col) {
  for (i=0; i<n; i++)
    for (j=0; j<m; j++)
      f1 = A[col][j]+A[col+1][j]
       + A[col-1][j] + ...
}
```

(N = No Prefetching, S = Prefetching w/ Static Analysis, F = Prefetching w/ Feedback)

- feedback may compensate for limitations of static analysis

### 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>

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

### Overview

Tolerating Memory Latency

Prefetching Compiler Algorithm and Results

Increasing Scope of Prefetching

Implications of These Results

### Concluding Remarks

- Demonstrated that software prefetching is effective.
  - compiler works well for matrix-based codes
  - hand-inserted prefetching also not too difficult
  - uniprocessors and multiprocessors
- Techniques requiring complex hardware appear unnecessary.
  - hardware-based prefetching, context-switching
- Hardware should focus on providing sufficient memory bandwidth.