Lecture 27

Compiler Algorithms for Prefetching Data

I. Prefetching for Arrays
II. Prefetching for Recursive Data Structures

Reading: ALSU 11.11.4

Advanced readings (optional):

The Memory Latency Problem

- \( \uparrow \) processor speed \( \gg \) \( \uparrow \) memory speed
- 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

Prefetching for Arrays: Overview

- Tolerating Memory Latency
- Prefetching Compiler Algorithm and Results
- Implications of These Results
Coping with Memory Latency

Reduce Latency:
- Locality Optimizations
  - reorder iterations to improve cache reuse

Tolerate Latency:
- Prefetching
  - move data close to the processor before it is needed

Tolerating Latency Through Prefetching

Without Prefetching

With Prefetching

• overlap memory accesses with computation and other accesses

Types of Prefetching

Cache Blocks:
- (-) limited to unit-stride accesses

Nonblocking Loads:
- (-) limited ability to move back before use

Hardware-Controlled Prefetching:
- (-) limited to constant-strides and by branch prediction
- (+) no instruction overhead

Software-Controlled Prefetching:
- (-) software sophistication and overhead
- (+) minimal hardware support and broader coverage

Prefetching Research 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

Reducer Prefetching Overhead

- instructions to issue prefetches
- extra demands on memory system

Steps in Locality Analysis

1. Find data reuse
   - if caches were infinitely large, we would be finished
2. Determine "localized iteration space"
   - set of inner loops where the data accessed by an iteration is expected to fit within the cache
3. Find data locality:
   - reuse \( \cap \) localized iteration space \( \Rightarrow \) locality
Data Locality Example

for \( i = 0 \) to 2
for \( j = 0 \) to 100
\[ A[i][j] = B[j][0] + B[j+1][0]; \]

Reuse Analysis: Representation

for \( i = 0 \) to 2
for \( j = 0 \) to 100
\[ A[i][j] = B[j][0] + B[j+1][0]; \]

- Map \( n \) loop indices into \( d \) array indices via array indexing function:

Finding Temporal Reuse

- Temporal reuse occurs between iterations \( \vec{i}_1 \) and \( \vec{i}_2 \) whenever:

\[ H\vec{i}_1 + \vec{c} = H\vec{i}_2 + \vec{c} \]
\[ H(\vec{i}_1 - \vec{i}_2) = \vec{0} \]
- Rather than worrying about individual values of \( \vec{i}_1 \) and \( \vec{i}_2 \) we say that reuse occurs along direction vector \( \vec{r} \) when:

\[ H(\vec{r}) = \vec{0} \]
- Solution: compute the nullspace of \( H \)

Temporal Reuse Example

for \( i = 0 \) to 2
for \( j = 0 \) to 100
\[ A[i][j] = B[j][0] + B[j+1][0]; \]

- Reuse between iterations \( (i_1,j_1) \) and \( (i_2,j_2) \) whenever:

\[
\begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
1 \\\n0
\end{bmatrix}
+ \begin{bmatrix}
1 \\
0
\end{bmatrix}
= \begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
0 \\\n0
\end{bmatrix}
+ \begin{bmatrix}
1 \\
0
\end{bmatrix}
\]
\[
\begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
1 - i_2 \\
j_2 - j_1
\end{bmatrix}
= \begin{bmatrix}
0 \\
0
\end{bmatrix}
\]
- True whenever \( j_1 = j_2 \), and regardless of the difference between \( i_1 \) and \( i_2 \).
  - i.e. whenever the difference lies along the nullspace of \( \begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix} \), which is \( \text{span}(\begin{bmatrix} 1,0 \end{bmatrix}) \) (i.e. the outer loop).
Localized Iteration Space

- Given finite cache, when does reuse result in locality?

\[
\text{for } i = 0 \text{ to } 2 \\
\text{for } j = 0 \text{ to } 8 \\
A[i][j] = B[j][0] + B[j+1][0];
\]

\[
\text{for } i = 0 \text{ to } 2 \\
\text{for } j = 0 \text{ to } 100000 \\
A[i][j] = B[j][0] + B[j+1][0];
\]

Localized: both i and j loops (i.e. span{(1,0),(0,1)})

Localize: j loop only (i.e. span{(0,1)})

- Localized if accesses less data than effective cache size

Computing Locality

- Reuse Vector Space \( \cap \) Localized Vector Space \( \Rightarrow \) Locality Vector Space

\[
\text{for } i = 0 \text{ to } 2 \\
\text{for } j = 0 \text{ to } 100 \\
A[i][j] = B[j][0] + B[j+1][0];
\]

- If both loops are localized:
  - \( \text{span}((1,0)) \cap \text{span}((1,0),(0,1)) \Rightarrow \text{span}((1,0)) \)
  - i.e. temporal reuse does result in temporal locality

- If only the innermost loop is localized:
  - \( \text{span}((1,0)) \cap \text{span}((0,1)) \Rightarrow \text{span}() \)
  - i.e. no temporal locality

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 iterations (( l = \text{cache line size} ))</td>
<td>( (i \text{ mod } l) = 0 )</td>
</tr>
</tbody>
</table>

Example:

\[
\text{for } i = 0 \text{ to } 2 \\
\text{for } j = 0 \text{ to } 100 \\
A[i][j] = B[j][0] + B[j+1][0];
\]

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>1 = 0</td>
<td>Peel loop 1</td>
</tr>
<tr>
<td>Spatial</td>
<td>(i mod l) = 0</td>
<td>Unroll loop 1 by 1</td>
</tr>
</tbody>
</table>

- Apply transformations recursively for nested loops
- Suppress transformations when loops become too large – avoid code explosion

Software Pipelining

\[ \text{Iterations Ahead} = \left\lceil \frac{l}{s} \right\rceil \]

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

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

Example Revisited

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Code with Prefetching</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (j = 0; j &lt; 25; j++)</td>
<td></td>
</tr>
<tr>
<td>A[i][j] = 0;</td>
<td></td>
</tr>
<tr>
<td>for (j = 0; j &lt; 25; j++)</td>
<td></td>
</tr>
<tr>
<td>B[j+1][0] = 0;</td>
<td></td>
</tr>
<tr>
<td>for (i = 0; i &lt; 3; i++)</td>
<td></td>
</tr>
<tr>
<td>prefetch(&amp;A[i][0]);</td>
<td></td>
</tr>
<tr>
<td>for (j = 0; j &lt; 25; j++)</td>
<td></td>
</tr>
<tr>
<td>prefetch(&amp;B[j+1][0]);</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Effectiveness of Locality Analysis

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

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

Original Miss Breakdown

- Large pf-miss → 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

for (i = 0; i<100; 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

Software Pipelining for Indirections

for (i = 0; i<100; i++)
    prefetch(&A[index[i]]);

for (i = 0; i<100; i++)
    prefetch(&index[i])
    prefetch(&A[index[i]]);

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

Analysis: memory accesses stalls

Software Pipelined Loop (5 iterations ahead)

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

for (i = 0; i<100; i++)
    prefetch(&A[index[i]]);

for (i = 0; i<100; i++)
    prefetch(&index[i])
    prefetch(&A[index[i]]);

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

Analysis: memory accesses stalls

Software Pipelined Loop (5 iterations ahead)

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

for (i = 0; i<100; i++)
    prefetch(&A[index[i]]);

for (i = 0; i<100; i++)
    prefetch(&index[i])
    prefetch(&A[index[i]]);

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

Analysis: memory accesses stalls
**Indirection Prefetching Results**

- Larger overheads in computing indirection addresses
- Significant overall improvements for IS and CG

**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

**Prefetching for Arrays: Concluding Remarks**

- Demonstrated that software prefetching is effective
  - Selective prefetching to eliminate overhead
  - Dense matrices and indirections / sparse matrices
  - Uniprocessors and multiprocessors
- Hardware should focus on providing sufficient memory bandwidth

**Part II: Prefetching for Recursive Data Structures**
Recursive Data Structures

• Examples:
  – linked lists, trees, graphs, ...

• A common method of building large data structures
  – especially in non-numeric programs

• Cache miss behavior is a concern because:
  – large data set with respect to the cache size
  – temporal locality may be poor
  – little spatial locality among consecutively-accessed nodes

Goal:
• Automatic Compiler-Based Prefetching for Recursive Data Structures

Scheduling Prefetches for Recursive Data Structures

Our Goal: fully hide latency — thus achieving fastest possible computation rate of 1/W

• e.g., if L = 3W, we must prefetch 3 nodes ahead to achieve this

Performance without Prefetching

computation rate = 1 / (L+W)
Prefetching One Node Ahead

while (p) {
    pf(p->next);  
    work(p->data);  
    p = p->next;  
}

Prefetching Three Nodes Ahead

while (p) {
    pf(p->next->next->next);  
    work(p->data);  
    p = p->next;  
}

Our Goal: Fully Hide Latency

while (p) {
    pf(n+k);  
    work(n+k);  
    p = p->next;  
}

Overview

• Challenges in Prefetching Recursive Data Structures
  • Three Prefetching Algorithms
    – Greedy Prefetching
    – History-Pointer Prefetching
    – Data-Linearization Prefetching
  • Experimental Results
  • Conclusions
### Overcoming the Pointer-Chasing Problem

**Key:**
- \( n_i \) needs to know \( \Delta n_{i+d} \) without referencing the \( d-1 \) intermediate nodes

**Our proposals:**
- Use existing pointer(s) in \( n_i \) to approximate \( \Delta n_{i+d} \)
  - Greedy Prefetching
- Add new pointer(s) to \( n_i \) to approximate \( \Delta n_{i+d} \)
  - History-Pointer Prefetching
- Compute \( \Delta n_{i+d} \) directly from \( n_i \) (no ptr. deref.)
  - History-Pointer Prefetching

### Greedy Prefetching

- Prefetch all neighboring nodes (simplified definition)
  - Only one will be followed by the immediate control flow
  - Hopefully, we will visit other neighbors later

### History-Pointer Prefetching

- Add new pointer(s) to each node
  - History-pointers are obtained from some recent traversal

### Data-Linearization Prefetching

- No pointer dereferences are required
- Map nodes close in the traversal to contiguous memory
Summary of Prefetching Algorithms

<table>
<thead>
<tr>
<th>Control over Prefetching Distance</th>
<th>Greedy</th>
<th>History-Pointer</th>
<th>Data-Linearization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefetching Distance</td>
<td>little</td>
<td>more precise</td>
<td>more precise</td>
</tr>
<tr>
<td>Applicability to Recursive Data Structures</td>
<td>any RDS</td>
<td>revisited: changes only slowly</td>
<td>must have a major traversal order; changes only slowly</td>
</tr>
<tr>
<td>Overhead in Preparing Prefetch Addresses</td>
<td>none</td>
<td>space + time</td>
<td>none in practice</td>
</tr>
<tr>
<td>Ease of Implementation</td>
<td>relatively straightforward</td>
<td>more difficult</td>
<td>more difficulty</td>
</tr>
</tbody>
</table>

- Greedy prefetching is the most widely applicable algorithm
  - fully implemented in SUIF

Experimental Framework

Benchmarks
- Olden benchmark suite
  - 10 pointer-intensive programs
  - covers a wide range of recursive data structures

Simulation Model
- Detailed, cycle-by-cycle simulations
- MIPS R10000-like dynamically-scheduled superscalar

Compiler
- Implemented in the SUIF compiler
- Generates fully functional, optimized MIPS binaries

Overview

- Challenges in Prefetching Recursive Data Structures
- Three Prefetching Algorithms
- Experimental Results
- Conclusions

Implementation of Our Prefetching Algorithms

Automated in the SUIF compiler

- Schedule Greedy Prefetches
- Schedule History-Pointer Prefetches
- Schedule Data-Linearization Prefetches

- identify RDS types
- find recurrent pointer updates in loops and recursive procedures
  - insert prefetches at the earliest possible places
  - minimize prefetching overhead
Performance of Compiler-Inserted Greedy Prefetching

- Eliminates much of the stall time in programs with large load stall penalties
  - Half achieve speedups of 4% to 45%

Coverage Factor

- Coverage factor = pf_hit + pf_miss
  - 7 out of 10 have coverage factors > 60%
    - em3d, power, voronoi have many array or scalar load misses
  - Small pf_miss fractions → effective prefetch scheduling

Unnecessary Prefetches

- % dynamic pfs that are unnecessary because the data is in the D-cache
  - 4 have >80% unnecessary prefetches
  - Could reduce overhead by eliminating static pfs that have high hit rates

Reducing Overhead Through Memory Feedback

- Eliminating static pfs with hit rate >95% speeds them up by 1-8%
  - However, eliminating useful prefetches can hurt performance
  - Memory feedback can potentially improve performance
Performance of History-Pointer Prefetching

- Applicable because a list structure does not change over time
- 40% speedup over greedy prefetching through:
  - better miss coverage (64% → 100%)
  - fewer unnecessary prefetches (41% → 29%)
- Improved accuracy outweighs increased overhead in this case

Performance of Data-Linearization Prefetching

- Creation order equals major traversal order in `treeadd` & `perimeter`
  - hence data linearization is done without data restructuring
- 9% and 18% speedups over greedy prefetching through:
  - fewer unnecessary prefetches:
    - 94% → 78% in `perimeter`, 87% → 81% in `treeadd`
    - while maintaining good coverage factors:
      - 100% → 80% in `perimeter`, 100% → 93% in `treeadd`

Conclusions

- Propose 3 schemes to overcome the pointer-chasing problem:
  - Greedy Prefetching
  - History-Pointer Prefetching
  - Data-Linearization Prefetching
- Automated greedy prefetching in SUIF
  - improves performance significantly for half of Olden
  - memory feedback can further reduce prefetch overhead
- The other 2 schemes can outperform greedy in some situations