Lecture 22
Prefetching Pointer-based Structures

I. Challenges
II. Three Prefetching Algorithms
III. Experimental Results

### Recall: Prefetch Predicate

<table>
<thead>
<tr>
<th>Locality Type</th>
<th>Miss Instance</th>
<th>Predicate on Iteration Space</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 elements/cache line)</td>
<td>(i mod L) = 0</td>
</tr>
</tbody>
</table>

#### Example:

```
for i = 0 to 2
    for j = 0 to N-2
        A[i][j] = B[j][0] + B[j+1][0];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Locality</th>
<th>Predicate on Iteration Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[i][j]</td>
<td>[i] = none, [j] = spatial</td>
<td>(j mod L) = 0</td>
</tr>
<tr>
<td>B[j+1][0]</td>
<td>[i] = temporal, [j] = none</td>
<td>i = 0</td>
</tr>
</tbody>
</table>
Recall: Loop Splitting for Prefetching Arrays

- Decompose loops to isolate cache miss instances
  - cheaper than inserting IF(Prefetch Predicate) 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>

Loop peeling: split any problematic first (or last) few iterations from the loop & perform them outside of the loop body
Recall: Example Code with Prefetching

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];
```

- Cache Hit
- Cache Miss

```
prefetch(&B[0][0]);
for (j = 0; j < 6; j += 2) {
    prefetch(&B[j+1][0]);
    prefetch(&B[j+2][0]);
    prefetch(&A[0][j]);
}
for (j = 0; j < 94; j += 2) {
    prefetch(&B[j+7][0]);
    prefetch(&B[j+8][0]);
    prefetch(&A[0][j+6]);
    A[0][j] = B[j][0] + B[j+1][0];
    A[0][j+1] = B[j+1][0] + B[j+2][0];
}
for (j = 94; j < 100; j += 2) {
    A[0][j] = B[j][0] + B[j+1][0];
    A[0][j+1] = B[j+1][0] + B[j+2][0];
}
for (i = 1; i < 3; i++) {
    for (j = 0; j < 6; j += 2)
        prefetch(&A[i][j]);
    for (j = 0; j < 94; j += 2) {
        prefetch(&A[i][j+6]);
        A[i][j] = B[j][0] + B[j+1][0];
        A[i][j+1] = B[j+1][0] + B[j+2][0];
    }
    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];
    }
}
```

```
Today: Recursive (i.e., Pointer-based) 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
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 = \frac{1}{L+W}

while (p){
    work(p->data);
    p = p->next;
}
Prefetching One Node Ahead

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

• Computation is overlapped with memory accesses

computation rate = \( \frac{1}{L} \)
Prefetching Three Nodes Ahead

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

Prefetching Three Nodes Ahead

while (p){
    pf(p->next->next->next);
    work(p->data);
    p = p->next;
}
Prefetching Three Nodes Ahead

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

computation rate does not improve (still = 1/L)!

**Pointer-Chasing Problem:**

- any scheme which follows the pointer chain is limited to a rate of 1/L
Our Goal: Fully Hide Latency

• achieves the fastest possible computation rate of $1/W$

while (p){
    pf(&n_{i+3});
    work(p->data);
    p = p->next;
}
Overview

• Challenges in Prefetching Recursive Data Structures

• Three Prefetching Algorithms
  – Greedy Prefetching
  – History-Pointer Prefetching
  – Data-Linearization Prefetching

• Experimental Results
Overcoming the Pointer-Chasing Problem

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

Three Algorithms:
• use *existing* pointer(s) in \( n_i \) to approximate \&n_{i+d}
  – Greedy Prefetching

• add *new* pointer(s) to \( n_i \) to approximate \&n_{i+d}
  – History-Pointer Prefetching

• compute \&n_{i+d} *directly* from \&n_i (no ptr deref)
  – Data-Linearization 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

```c
preorder(treeNode * t){
  if (t != NULL){
    pf(t->left);
    pf(t->right);
    process(t->data);
    preorder(t->left);
    preorder(t->right);
  }
}
```

- **Reasonably effective in practice**
- **However, little control over the prefetching distance**
History-Pointer Prefetching

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

- Trade space & time for better control over prefetching distances
Data-Linearization Prefetching

- No pointer dereferences are required
- Map nodes close in the traversal to contiguous memory

![Diagram of a tree with nodes labeled 1 to 15, showing preorder traversal and prefetching distance of 3 nodes.](image-url)
### Summary of Prefetching Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Greedy</th>
<th>History-Pointer</th>
<th>Data-Linearization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control over 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
Overview

- Challenges in Prefetching Recursive Data Structures
- Three Prefetching Algorithms
- Experimental Results
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
Implementation of Prefetching Algorithms

Automated in the Stanford SUIF compiler

- Recognize RDS Accesses
  - Identify RDS types
  - Find recurrent pointer updates in loops and recursive procedures

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

- 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 3% to 31%

\[ \text{O} = \text{Original} \]
\[ \text{G} = \text{Compiler-Inserted Greedy Prefetching} \]
- 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
% 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
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
• 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
Summary

• Three schemes to overcome the pointer-chasing problem:
  – Greedy Prefetching
  – History-Pointer Prefetching
  – Data-Linearization Prefetching

• Automated greedy prefetching in Stanford SUIF compiler
  – improves performance significantly for half of Olden
  – memory feedback can further reduce prefetch overhead

• The other 2 schemes can outperform greedy in some situations
Today’s Class: Prefetching Pointer-based Structures

I. Challenges
II. Three Prefetching Algorithms
III. Experimental Results

Monday’s Class

- Thread Level Speculation