Lectures 26-27

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

- \( \uparrow \text{processor speed} \gg \uparrow \text{memory speed} \)
- \text{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

- Load A
- Fetch A
- Load B
- Fetch B

With Prefetching

- Prefetch A
- Prefetch B
- Load A
- Load B
- Fetch A
- Fetch B

- Executing Instructions
- Stalled Waiting for Data

- 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
Reducing Prefetching Overhead

- instructions to issue prefetches
- extra demands on memory system

Hit Rates for Array Accesses

- important to minimize unnecessary prefetches
Compiler Algorithm

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

**Scheduling**: when/how to issue prefetches
- Loop Splitting
- Software Pipelining
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 ∩ localized iteration space ⇒ locality
Data Locality Example

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

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

- Map $n$ loop indices into $d$ array indices via array indexing function:
\[
\vec{f}(\vec{i}) = H\vec{i} + \vec{c}
\]

\[
A[i][j] = A \left( \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)
\]

\[
B[j][0] = B \left( \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)
\]

\[
B[j+1][0] = B \left( \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right)
\]
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:
\[
\begin{bmatrix}
0 & 1 \\
0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
i_1 \\
j_1 \\
\end{bmatrix}
+ \begin{bmatrix}
1 \\
0 \\
\end{bmatrix}
= \begin{bmatrix}
0 & 1 \\
0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
i_2 \\
j_2 \\
\end{bmatrix}
+ \begin{bmatrix}
1 \\
0 \\
\end{bmatrix}
\]
\[
\begin{bmatrix}
0 & 1 \\
0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
i_1 - i_2 \\
j_1 - j_2 \\
\end{bmatrix}
= \begin{bmatrix}
0 \\
0 \\
\end{bmatrix}
\]

• 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 \(\text{span}\{(1,0)\}\) (i.e. the outer loop).
Localized Iteration Space

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

  for \( i = 0 \) to 2
  for \( j = 0 \) to 8
  \[
  A[i][j] = B[j][0] + B[j+1][0];
  \]

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

  for \( i = 0 \) to 2
  for \( j = 0 \) to 1000000
  \[
  A[i][j] = B[j][0] + B[j+1][0];
  \]

  **Localized:** \( j \) loop only
  (i.e. \( \text{span}((0,1)) \))

- **Localized** if accesses less data than *effective cache size*
Computing Locality

- **Reuse Vector Space** $\cap$ **Localized Vector Space** $\Rightarrow$ **Locality Vector Space**

- **Example:**
  
  ```
  for i = 0 to 2
  for j = 0 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 $l$ iterations ($l = cache line size$)</td>
<td>$(i \mod l) = 0$</td>
</tr>
</tbody>
</table>

**Example:**

$$
\begin{align*}
\text{for } i &= 0 \text{ to } 2 \\
\text{for } j &= 0 \text{ to } 100 \\
A[i][j] &= B[j][0] + B[j+1][0];
\end{align*}
$$

<table>
<thead>
<tr>
<th>Reference</th>
<th>Locality</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A[i][j]$</td>
<td>$i$ = [none spatial]</td>
<td>$(j \mod 2) = 0$</td>
</tr>
<tr>
<td>$B[j+1][0]$</td>
<td>$i$ = [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{l}{s} \right\rfloor

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

Original Loop

```c
for (i = 0; i<100; i++)
a[i] = 0;
```

Software Pipelined Loop

(5 iterations ahead)

```c
for (i = 0; i<5; i++)
    /* Prolog */
prefetch(&a[i]);

for (i = 0; i<95; i++)
    /* Steady State*/
prefetch(&a[i+5]);
a[i] = 0;
}

for (i = 95; i<100; i++)
    /* Epilog */
a[i] = 0;
```
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];
```

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

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

Code with Prefetching

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

Cache Hit

Cache Miss

i = 0

i > 0

15-745: Data Prefetching

Todd C. Mowry
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 typically low
- 6 of 13 have speedups over 45%

(N = No Prefetching, S = Selective Prefetching)
Effectiveness of Locality Analysis

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

(I = Indiscriminate Prefetching, S = Selective Prefetching)
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 → 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

GMTRY (Cache Blocking)

VPENTA (Loop Interchange)
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

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++)
    prefetch(&index[i]);

for (i = 0; i<5; i++)
    prefetch(&index[i+5]);
    prefetch(&A[index[i]]);
}

for (i = 0; i<90; i++)
    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]];
```
• 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
Overview

• Challenges in Prefetching Recursive Data Structures
• Three Prefetching Algorithms
• Experimental Results
• Conclusions
**Scheduling Prefetches for Recursive Data Structures**

Currently visiting

\[
p \rightarrow n_i \rightarrow n_{i+1} \rightarrow n_{i+2} \rightarrow n_{i+3} \rightarrow \cdots
\]

Want to prefetch

Our Goal: fully hide latency
– thus achieving fastest possible computation rate of \(1/W\)

\[
p = &n_0
\]
while \((p)\){
  \[
  \text{work}(p \rightarrow \text{data});
  
  p = p \rightarrow \text{next};
  
  \]
}

load *p here

\[
L\quad L
\]

\[
W\quad W
\]

\[
\text{loading a node}
\]

\[
\text{work()}
\]

• e.g., if \(L = 3W\), we must prefetch 3 nodes ahead to achieve this
Performance without Prefetching

\[ \text{computation rate} = \frac{1}{(L+W)} \]

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

- Computation is overlapped with memory accesses

\[ \text{computation rate} = \frac{1}{L} \]

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

Pointer-Chasing Problem:
• any scheme which follows the pointer chain is limited to a rate of 1/L

\[
\text{computational rate does not improve (still } = \frac{1}{L})!
\]
Our Goal: Fully Hide Latency

- achieves the fastest possible computation rate of $1/W$
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 $\&n_{i+d}$ without referencing the d-1 intermediate nodes

Our proposals:
- 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)
  - History-Pointer Prefetching

$A = \text{Address generating function}$

Todd C. Mowry
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

preorder traversal

prefetching distance = 3 nodes
## 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><strong>Control over Prefetching Distance</strong></td>
<td>little</td>
<td>more precise</td>
<td>more precise</td>
</tr>
<tr>
<td><strong>Applicability to Recursive Data Structures</strong></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><strong>Overhead in Preparing Prefetch Addresses</strong></td>
<td>none</td>
<td>space + time</td>
<td>none in practice</td>
</tr>
<tr>
<td><strong>Ease of Implementation</strong></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
Overview

• Challenges in Prefetching Recursive Data Structures
• Three Prefetching Algorithms
• Experimental Results
• Conclusions
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 Our Prefetching Algorithms

Automated in the SUIF compiler

Recognize RDS Accesses

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%

**Chart:**
- \( O \) = Original
- \( G \) = Compiler-Inserted Greedy Prefetching

**Legend:**
- Red = load stall
- Green = store stall
- Grey = inst. stall
- Blue = busy
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