The Memory Latency Problem

- Processor speed >> memory speed
- Caches are not a panacea
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

- 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 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 prefetces

**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

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

Spatial

Temporal

Group
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:

\[
\vec{f}(\vec{i}) = H\vec{i} + \vec{c}
\]

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

• Temporal reuse occurs between iterations $\vec{v}_1$ and $\vec{v}_2$ whenever:

$$H\vec{v}_1 + \vec{c} = H\vec{v}_2 + \vec{c}$$

$$H(\vec{v}_1 - \vec{v}_2) = \vec{0}$$

• Rather than worrying about individual values $\vec{v}_1$ of $\vec{v}_2$ and, we say that reuse occurs along direction vector 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).
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</td>
<td>$(i \mod l) = 0$</td>
</tr>
<tr>
<td></td>
<td>($l =$ cache line size)</td>
<td></td>
</tr>
</tbody>
</table>

Example:  
for $i = 0$ to 2  
for $j = 0$ to 100  

$$A[i][j] = B[j][0] + B[j+1][0];$$

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

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

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

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

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

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

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

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

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]];
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
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

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

- 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

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 \( 1/W \)

```c
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

• Conclusions
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
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

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>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>
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
CSC D70: Compiler Optimization Prefetching

Prof. Gennady Pekhimenko
University of Toronto
Winter 2020

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons