Tolerating Latency Through Prefetching

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2017
The Memory Latency Problem

- processor speed >> memory speed
- caches are not a panacea

From Hennessy & Patterson, CA: AQA, 5th Edition

CMU 15-418/618, Fall 2017
• 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
Even Worse: Remote Latencies in a NUMA Multiprocessors

- Long cache miss latencies due to:
  - remote memory accesses
  - cache coherence
Impact of Memory Latency in these Parallel Machines

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

→ 6 of 8 spend > 50% of time stalled for memory!
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
- Multithreading (aka Context Switching)
  - switch between threads upon a cache miss
Tolerating Latency Through Prefetching

- overlap memory accesses with computation and other accesses
Benefits of Prefetching

• Prefetch early enough
  – completely hides memory latency

• Issue prefetches in blocks
  – pipeline the misses
  – only the first reference stalls

• Prefetch with ownership
  – reduces write latency, coherence messages
Types of Prefetching

• Large cache blocks
  – limitations: spatial locality, false sharing

• Hardware-controlled prefetching
  – modern processors detect (and prefetch) simple strided access patterns
  – limitations: simple patterns, page boundaries, potential cache pollution

• Software-controlled prefetching
  – explicit instructions in modern instruction sets
  – advantages: more sophisticated access patterns
  – limitations: instruction overhead?
Software-Controlled Prefetching: Research Phases

1. Hand-inserted Prefetching
   – understand complexity of doing it by hand
   – understand limitations of how well we can do

2. Compiler-inserted Prefetching
   – understand what can be automated successfully
Parallel Applications Used in Hand-Inserted Case Studies

- **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
Hand-Inserted Prefetching: MP3D Case Study

- Best speedup: **1.86**
  - limited by small data set, plus prefetching only into cluster cache
### Hand-Inserted Prefetching: PTHOR Case Study

<table>
<thead>
<tr>
<th>Strategy Coverage</th>
<th>nopf 0%</th>
<th>pf1 14%</th>
<th>pf2 27%</th>
<th>pf3 56%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefetches</td>
<td>16.7</td>
<td>16.7</td>
<td>16.7</td>
<td>16.7</td>
</tr>
<tr>
<td>Sync Ops</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Writes</td>
<td>11.5</td>
<td>10.0</td>
<td>9.7</td>
<td>8.9</td>
</tr>
<tr>
<td>Reads</td>
<td>16.4</td>
<td>16.7</td>
<td>17.0</td>
<td>18.1</td>
</tr>
<tr>
<td>Busy</td>
<td>55.4</td>
<td>54.4</td>
<td>47.9</td>
<td>41.7</td>
</tr>
</tbody>
</table>

- **Best speedup:** 1.14
  - linked lists are difficult to prefetch!

pf1: entire element record, readex
pf2: reorganized record + list heads
pf3: pf2 + other things based on profiling
Compiler-Based Prefetching

• How well can we do if we fully-automate prefetch insertion?

• What access patterns can we handle successfully?
  – arrays, pointers?

• Improving performance requires:
  – maximizing benefit, while
  – minimizing 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 \( \cap \) localized iteration space \( \Rightarrow \) locality
Data Locality Example

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

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

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

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

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

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

- Reuse between iterations \((i_1,j_1)\) and \((i_2,j_2)\) whenever:
  \[
  \begin{bmatrix}
  0 & 1 \\
  0 & 0
  \end{bmatrix}
  \begin{bmatrix}
  i_1 \\
  j_1
  \end{bmatrix}
  +
  \begin{bmatrix}
  1 \\
  0
  \end{bmatrix}
  \begin{bmatrix}
  0 & 1 \\
  0 & 0
  \end{bmatrix}
  \begin{bmatrix}
  i_2 \\
  j_2
  \end{bmatrix}
  +
  \begin{bmatrix}
  1 \\
  0
  \end{bmatrix}
  \begin{bmatrix}
  0 \\
  0
  \end{bmatrix}
  \]

- 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?**

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

  \[
  \text{for } i = 0 \text{ to } 2 \\
  \quad \text{for } j = 0 \text{ to } 1000000 \\
  \quad A[i][j] = B[j][0] + B[j+1][0];
  \]

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

  **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
  

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

### Example:

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

<table>
<thead>
<tr>
<th>Reference</th>
<th>Locality</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A[i][j] )</td>
<td>(i) = [ none spatial ]</td>
<td>((j \text{ 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

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

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

**Original Loop**

\[
\text{for (} i = 0; i < 100; i++) \\
\text{a}[i] = 0;
\]

**Software Pipelined Loop**

(5 iterations ahead)

\[
\text{for (} i = 0; i < 5; i++) \\
\text{prefetch(} &\text{a}[i]\text{);}
\]

\[
\text{for (} i = 0; i < 95; i++) \\
\text{prefetch(} &\text{a}[i+5]\text{);} \\
\text{a}[i] = 0;
\]

\[
\text{for (} i = 95; i < 100; i++) \\
\text{a}[i] = 0;
\]
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];
```

Code with Prefetching

```c
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]);
}
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];
}
for (i = 1; i < 3; i++) {
    prefetch(&A[i][0]);
    for (j = 0; j < 6; j += 2)
        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];
    }
}
```
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%
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 $\rightarrow$ ineffective scheduling
  - conflicts replace prefetched data (CHOLSKY, TOMCATV)
  - prefetched data still found in secondary cache

Carnegie Mellon

CMU 15-418/618, Fall 2017
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

\[
\text{for } (i = 0; i<100; i++)
\text{ sum += A[index[i]];}
\]

**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++)
    /* 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]];
```
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
Prefetching for Parallel Shared-Address-Space Machines

- Main memory is *physically distributed* (aka NUMA)
  - but logically a single, *shared address space*
- Hardware cache coherence
Prefetching for Multiprocessors

• **Non-binding vs. binding prefetches:**
  – use non-binding since data remains coherent until accessed later

```c
prefetch(&x);
...
LOCK(L);
x = x + 1;
UNLOCK(L);
```

→ no restrictions on when prefetches can be issued

• **Dealing with coherence misses:**
  – localized iteration space takes explicit synchronization into account

• **Further optimizations:**
  – prefetching in exclusive-mode in read-modify-write situations
• 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

Normalized Message Traffic

- Message traffic reduced by 7% to 29%
- Relaxed memory consistency → 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
Prefetching for Databases

- Hash Join
- Prefetching + SIMD in full queries
Simple Hash Join

- Build a hash table to index all tuples of the smaller relation
- Probe this hash table using all tuples of the larger relation

Random access patterns: little spatial or temporal locality
Challenges

- **Naïve approach**: prefetch within the processing of a single tuple
  - e.g., prefetch within a single hash table visit

- **Does not work!**
  - dependencies essentially form a critical path
  - addresses would be generated too late for prefetching
  - randomness makes prediction almost impossible
A Simplified Probing Algorithm

```java
foreach tuple in probe partition {
    compute hash bucket number;
    visit the hash bucket header;
    visit the hash cell array;
    visit the matching build tuple to compare keys and produce output tuple;
}
```
An Intuitive Way to Represent the Algorithm

```python
foreach tuple in probe partition {
    compute hash bucket number;
    visit the hash bucket header;
    visit the hash cell array;
    visit the matching build tuple to compare keys and produce output tuple;
}
```
Group Prefetching

```plaintext
foreach group of tuples in probe partition {
    foreach tuple in the group {
        compute hash bucket number;
        prefetch the target bucket header;
    }
    foreach tuple in the group {
        visit the hash bucket header;
        prefetch the hash cell array;
    }
    foreach tuple in the group {
        visit the hash cell array;
        prefetch the matching build tuple;
    }
    foreach tuple in the group {
        visit the matching build tuple to compare keys and produce output tuple;
    }
}
```
Applying Prefetching & SIMD to Queries: e.g., TPC-C Q19

```
SELECT SUM(...) AS revenue
FROM LineItem JOIN Part ON l_partkey = p_partkey
WHERE (CLAUSE1) OR (CLAUSE2) OR (CLAUSE3)
```

- SIMD and prefetching are complementary: over 2X speedup
A More Complex Query from TPC-C: Q3

- SIMD + Prefetching → large improvement
Prefetching Only Works if there is Sufficient Memory Bandwidth

- If you are already bandwidth-limited, then prefetching cannot help
How Can We Provide Sufficient Memory Bandwidth?

Recent enabling technology: 3D stacking of DRAM chips

- DRAMs connected via through-silicon-vias (TSVs) that run through the chips
  - TSVs provide highly parallel connection between logic layer and DRAMs
- Base layer of stack “logic layer” is memory controller, manages requests from processor
- Silicon “interposer” serves as high-BW interconnect between DRAM stack and processor

Technologies:

- Micron/Intel’s Hybrid Memory Cube (HMC)
- AMD’s High-Bandwidth Memory (HBM): 1024 bit interface to stack
GPUs Have Been Adopting HBM Technologies

**AMD Radeon Fury GPU (2015)**
- 4096-bit interface:
  - 4 HBM chips x 1024 bit interface per chip
  - 512 GB/sec BW

**NVIDIA P100 GPU (2016)**
- 4096-bit interface:
  - 4 HBM2 chips x 1024 bit interface per chip
  - 720 GB/sec peak BW
  - 4 x 4 GB = 16 GB capacity
Xeon Phi (Knights Landing) MCDRAM

- 16 GB in package stacked DRAM
- Can be configured as either:
  - 16 GB last level cache
  - 16 GB separate address space
    - aka “flat mode”
- Intel’s claims:
  - ~ same latency at DDR4
  - ~5x bandwidth of DDR4
  - ~5x less energy cost per bit transferred

```c
// allocate buffer in MCDRAM ("high bandwidth" memory malloc)
float* foo = hbw_malloc(sizeof(float) * 1024);
```
Prefetching for 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

Currently visiting \( p \)  

want to prefetch

\( n_i \) \( n_{i+1} \) \( n_{i+2} \) \( n_{i+3} \) 

\( p = &n_0 \)
while (p){
    work(p -> data);
    p = p -> next;
}

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

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

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

\[
\text{while (p)}\{
\text{pf(p->next->next->next);}
\text{work(p->data);}
\text{p = p->next;}
\}
\]

\[L \]

\[W_i, W_{i+1}, W_{i+2}, W_{i+3}\]

\[n_i, n_{i+1}, n_{i+2}, n_{i+3}\]

**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

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

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

- Schedule Greedy Prefetches
  - Insert prefetches at the earliest possible places
  - Minimize prefetching overhead

- Schedule History-Pointer Prefetches

- Schedule Data-Linearization Prefetches
Eliminates much of the stall time in programs with large load stall penalties
   - half achieve speedups of 4% to 45%
**Coverage Factor**

- coverage factor = \( \text{pf\_hit} + \text{pf\_miss} \)
- 7 out of 10 have coverage factors > 60%
  - em3d, power, voronoi have many array or scalar load misses
- small pf\_miss fractions \( \rightarrow \) 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

G = greedy prefetching
Fxx = greedy prefetching where static pfs with
hit rate > xx% are eliminated

Carnegie Mellon
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

\[ O = \text{original} \]
\[ G = \text{greedy prefetching} \]
\[ D = \text{data-linearization prefetching} \]
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