Lectures 21
Prefetching Arrays

I. Tolerating Memory Latency
II. Prefetching Compiler Algorithm
III. Experimental 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

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

II. Compiler Algorithm

Analysis: what to prefetch
- Locality Analysis

Scheduling: when/how to issue prefetches
- Loop Splitting
- Software Pipelining
Recall: 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

Recall: Types of Data Reuse/Locality

for \( i = 0 \) to 2
for \( j = 0 \) to 99
\[ A[i][j] = B[j][0] + B[j+1][0]; \]

<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>
</tbody>
</table>

Example:
for \( i = 0 \) to 2
for \( j = 0 \) to 99
\[ A[i][j] = B[j][0] + B[j+1][0]; \]

<table>
<thead>
<tr>
<th>Reference</th>
<th>Locality</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[i][j]</td>
<td>[none]</td>
<td>( j ) mod L = 0</td>
</tr>
<tr>
<td>B[j+1][0]</td>
<td>[Temporal]</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(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></td>
</tr>
<tr>
<td>Spatial</td>
<td>(i mod L) = 0</td>
<td>(L elements/cache line)</td>
</tr>
</tbody>
</table>

Loop peeling: split any problematic first (or last) few iterations from the loop & performs them outside of the loop body

![Loop peeling diagram]

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

**Prefetching via Software Pipelining**

Iterations Ahead = \( \lceil \frac{l}{s} \rceil \) where \( l \) = memory latency, \( s \) = shortest path through loop body

**Original Loop**

\[
\text{for (i = 0; i<100; i++) for (i = 0; i<6; i++) \quad /\!\!/ \text{Prolog} */} \\
\text{a[i] = 0; \quad \text{prefetch}(\&a[i]); \quad /\!\!/ \text{Steady State}*/} \\
\text{a[i] = 0; \quad \text{a[i+6]; \quad */} \text{Epilog} */} \\
\text{a[i] = 0; \quad a[i+1] = 0; \quad \text{a[i]} = 0; \quad \text{a[i+2]; \quad a[i+1] = 0; \quad */} \text{Epilog} */}
\]

Are there any wasted prefetches?

**Software Pipelined Loop**

\[
\text{for (i = 0; i<100; i++) for (i = 0; i<6; i++) \quad /\!\!/ \text{Prolog} */} \\
\text{prefetch(\&a[i]); \quad /\!\!/ \text{Steady State}*/} \\
\text{a[i] = 0; \quad \text{a[i+6]; \quad */} \text{Epilog} */} \\
\text{a[i] = 0; \quad a[i+1] = 0; \quad \text{a[i]} = 0; \quad \text{a[i+2]; \quad a[i+1] = 0; \quad */} \text{Epilog} */}
\]

(2 elements/cache line)
### 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];
```

#### Prefetch Code
```c
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];
    }
```

### III. Experimental Framework

#### 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 / Applications:
- Detailed cache simulator driven by pixified object code
- Memory subsystem:
  - 8K L1 / 256K L2 direct-mapped caches, 32 byte lines
  - miss penalties: 12 / 75 cycles
- Applications from SPEC, SPLASH, and NAS Parallel

### Experimental Results (Dense Matrix Uniprocessor)

#### 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

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

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
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++)
    /* Prelog 1 */
    prefetch(&index[i]);
for (i = 0; i<10; i++)
    /* Prelog 2 */
    prefetch(&A[index[i]]);
for (i = 90; i<95; i++)
    /* Epilog 1 */
    prefetch(&A[index[i]]);
for (i = 95; i<100; i++)
    sum += A[index[i]];

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

Wednesday's Class

- Prefetching Pointer-based Structures