Lecture 18
Memory Hierarchy Optimizations &
Locality Analysis

[ALSU 7.4.2-7.4.3, 11.2-11.5]

Caches: A Quick Review

• How do they work?
• Why do we care about them?
• What are typical configurations today?
• What are some important cache parameters that will affect performance?

Optimizing Cache Performance

• Things to enhance:
  • temporal locality
  • spatial locality

• Things to minimize:
  • conflicts (i.e. bad replacement decisions)

What can the compiler do to help?

Two Things We Can Manipulate

• Time:
  • When is an object accessed?

• Space:
  • Where does an object exist in the address space?

How do we exploit these two levers?
**Time: Reordering Computation**

- What makes it difficult to know *when* an object is accessed?
- How can we predict a *better time* to access it?
  - What information is needed?
- How do we know that this would be *safe*?

**Space: Changing Data Layout**

- What do we know about an object’s *location*?
  - scalars, structures, pointer-based data structures, arrays, code, etc.
- How can we tell what a *better layout* would be?
  - how many can we create?
- To what extent can we *safely* alter the layout?

**Types of Objects to Consider**

- Scalars
- Structures & Pointers
- Arrays

** Scalars**

```c
int x;
double y;
foo(int a) {
    int i;
    ...
    x = a*i;
    ...
}
```
**Structures and Pointers**

- What can we do here?
  - within a node
  - across nodes

- What limits the compiler’s ability to optimize here?

```c
struct {
    int count;
    double velocity;
    double inertia;
    struct node *neighbors[N];
} node;
```

**Arrays / Matrices**

- usually accessed within loops nests
  - makes it easy to understand “time”
- what we know about array element addresses:
  - start of array?
  - relative position within array

```c
double A[N][N], B[N][N];
...
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] = B[j][i];
```

**Handy Representation: “Iteration Space”**

- each position represents an iteration

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

**Visitation Order in Iteration Space**

- Note: iteration space ≠ data space

```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] = B[j][i];
```
When Do Cache Misses Occur?

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

Optimizing the Cache Behavior of Array Accesses

• We need to answer the following questions:
  • when do cache misses occur?
  • use "locality analysis"
  • can we change the order of the iterations (or possibly data layout) to produce better behavior?
  • evaluate the cost of various alternatives
  • does the new ordering/layout still produce correct results?
  • use "dependence analysis"

Examples of Loop Transformations

• Loop Interchange
• Cache Blocking
• Skewing: iterate thru iteration space in the loops at an angle
• Loop Reversal: execute iterations in a loop in reverse order
  • ...
  
(we will briefly discuss the first two; see ALSU II.7.8 for others)
**Loop Interchange**

- (assuming 2 elements/cache line & \( N \) is large relative to cache size)

```
for \( i = 0 \) to \( N-1 \)
for \( j = 0 \) to \( N-1 \)
\[ A[j][i] = i*j; \]
```

**Cache Blocking (aka "Tiling")**

```
for \( i = 0 \) to \( N-1 \)
for \( j = 0 \) to \( N-1 \)
\[ f(A[i],A[j]); \]
```

- now we can exploit temporal locality

**Impact on Visitation Order in Iteration Space**

```
for \( i = 0 \) to \( N-1 \)
for \( j = 0 \) to \( N-1 \)
\[ f(A[i],A[j]); \]
```

- brings square sub-blocks of matrix "\( b \)" into the cache
- completely uses them up before moving on
- reduces the number of misses from \( \frac{L^3}{c} \) or \( \frac{N^3}{L} \) to only \( \frac{2N^2}{L} \)

\( (C=\text{cache size}, L=\text{line size}) \)
Predicting Cache Behavior through "Locality Analysis"

- Definitions:
  - **Reuse:** accessing a location that has been accessed in the past
  - **Locality:** accessing a location that is now found in the cache

- Key Insights
  - Locality only occurs when there is reuse!
  - BUT, reuse does not necessarily result in locality.
    - why not?

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 \(\land\) localized iteration space \(\Rightarrow\) locality

Types of Data Reuse/Locality

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

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>A[i][j]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Hit</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>Miss</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Hit</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>Miss</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Hit</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>Miss</td>
</tr>
</tbody>
</table>

Temporal

Spatial

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:
  \[
  j'(i) = Hi + c
  \]

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

- 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 \( \begin{bmatrix}
  0 & 1 \\
  0 & 0 \\
  \end{bmatrix} \) (i.e. the outer loop).

More Complicated Example

for \( i = 0 \) to \( N-1 \)
for \( j = 0 \) to \( N-1 \)
\[
A[i+j][0] = i*j;
\]

- Nullspace of \( \begin{bmatrix}
  1 & 1 \\
  1 & 0 \\
  0 & 0 \\
  \end{bmatrix} = \text{span}\{[1,-1]\} \)
  i.e. when \( \Delta i = -\Delta j \).

Computing Spatial Reuse

- Assume two array elements share the same cache line iff they differ only in the last dimension
  - E.g., share the same row in a 2-dimensional array
- Why is this a reasonable approximation?
  - What are its limitations?

- Replace last row of \( H \) with zeros, creating \( H_s \)
- Find the nullspace of \( H_s \)
- Result: vector along which we access the same row
Computing Spatial Reuse: Example

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

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

- $H_s = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$
- Nullspace of $H_s = \text{span}\{(0,1)\}$, i.e., the inner loop
  - access same row of $A[i][j]$ along inner loop

Computing Spatial Reuse: More Complicated Example

for $i = 0$ to $N-1$
for $j = 0$ to $N-1$
\[ A[i+j] = i \cdot j; \]

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

- $H_s = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$
- Nullspace of $H = \text{span}\{(1,-1)\}$
- Nullspace of $H_s = \text{span}\{(1,0),(0,1)\}$

Group Reuse (reuse from different static accesses)

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

- Limit analysis to consider only accesses with same $H$
  - i.e., index expressions differ only in their constant terms
- Determine when access same location (temporal) or same row (spatial)
- Only the "leading reference" suffers the bulk of the cache misses

Localized Iteration Space

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

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

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

Localized: both $i$ and $j$ loops

Localized: $j$ loop only

- 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:
  - span{(1,0)} $\cap$ span{(1,0),(0,1)} $\Rightarrow$ span{(1,0)}
  - i.e. temporal reuse does result in temporal locality

- If only the innermost loop is localized:
  - span{(1,0)} $\cap$ span{(0,1)} $\Rightarrow$ span{}
  - i.e. no temporal locality