Lecture 25
Memory Hierarchy Optimizations & Locality Analysis

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**

- **Locals**
  ```
  int x;
  double y;
  foo(int a)(
    int i;
    ...
    x = a*i;
    ...
  )
  ```
- **Globals**
- **Procedure arguments**
- **Is cache performance a concern here?**
- **If so, what can be done?**
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

```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];
```

- 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

Handy Representation: "Iteration Space"

- each position represents an iteration

Visitation Order in Iteration Space

- Note: iteration space ≠ data space
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
- Loop Reversal
- ...

(we will briefly discuss the first two)
Loop Interchange

\[
\begin{align*}
&\text{for } i = 0 \text{ to } N-1 \\
&\quad \text{for } j = 0 \text{ to } N-1 \\
&\quad A[j][i] = i*j; \\
\end{align*}
\]

\[
\begin{align*}
&\text{for } j = 0 \text{ to } N-1 \\
&\quad \text{for } i = 0 \text{ to } N-1 \\
&\quad A[j][i] = i*j; \\
\end{align*}
\]

- (assuming \(N\) is large relative to cache size)

Cache Blocking (aka "Tiling")

\[
\begin{align*}
&\text{for } i = 0 \text{ to } N-1 \\
&\quad \text{for } j = 0 \text{ to } N-1 \\
&\quad f(A[i], A[j]); \\
\end{align*}
\]

\[
\begin{align*}
&\text{for } j = 0 \text{ to } N-1 \\
&\quad \text{for } i = 0 \text{ to } N-1 \\
&\quad f(A[i], A[j]); \\
\end{align*}
\]

now we can exploit temporal locality

Impact on Visitation Order in Iteration Space

\[
\begin{align*}
&\text{for } i = 0 \text{ to } N-1 \\
&\quad \text{for } j = 0 \text{ to } N-1 \\
&\quad f(A[i], A[j]); \\
\end{align*}
\]

\[
\begin{align*}
&\text{for } j = 0 \text{ to } N-1 \\
&\quad \text{for } i = 0 \text{ to } N-1 \\
&\quad f(A[i], A[j]); \\
\end{align*}
\]

Cache Blocking in Two Dimensions

\[
\begin{align*}
&\text{for } i = 0 \text{ to } N-1 \\
&\quad \text{for } j = 0 \text{ to } N-1 \\
&\quad \text{for } k = 0 \text{ to } N-1 \\
&\quad c[i,k] += a[i,j]*b[j,k]; \\
\end{align*}
\]

\[
\begin{align*}
&\text{for } j = 0 \text{ to } N-1 \\
&\quad \text{for } i = 0 \text{ to } N-1 \\
&\quad \text{for } k = 0 \text{ to } N-1 \\
&\quad c[i,k] += a[i,j]*b[j,k]; \\
\end{align*}
\]

- brings square sub-blocks of matrix "b" into the cache
- completely uses them up before moving on
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 ∩ localized iteration space = locality

Types of Data Reuse/Locality

```plaintext
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>Hit</th>
<th>Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Spatial  Temporal  Group

Reuse Analysis: Representation

```plaintext
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:
  \[
  \vec{j}(\vec{i}) = H \vec{i} + \vec{c}
  \]

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>A[i][j]</th>
<th>B[j][0]</th>
<th>B[j+1][0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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:
  \[
  \begin{bmatrix}
  0 & 1 \\
  0 & 0 \\
  0 & 0 \\
  0 & 0 \\
  \end{bmatrix}
  + \begin{bmatrix}
  1 \\
  0 \\
  0 \\
  0 \\
  \end{bmatrix}
  = \begin{bmatrix}
  0 & 1 \\
  0 & 0 \\
  0 & 0 \\
  0 & 0 \\
  \end{bmatrix}
  + \begin{bmatrix}
  1 \\
  0 \\
  0 \\
  0 \\
  \end{bmatrix}
  \]
  \[
  \begin{bmatrix}
  0 & 1 \\
  0 & 0 \\
  0 & 0 \\
  0 & 0 \\
  \end{bmatrix}
  \begin{bmatrix}
  i_1 - i_2 \\
  j_1 - j_2 \\
  \end{bmatrix} = \vec{0}
  \]

- 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).

More Complicated Example

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

Hit
Miss

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

Computing Spatial Reuse

- 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

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

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

Group Reuse

- Only consider "uniformly generated sets"
  - index expressions differ only by constant terms
- Check whether they actually do access the same cache line
- Only the "leading reference" suffers the bulk of the cache misses

Computing Spatial Reuse: More Complicated Example

```plaintext
for i = 0 to N-1
  for j = 0 to N-1
    A[i+j] = i*j;
```

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

Localized Iteration Space

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

| Localized: both i and j loops (i.e. span((1,0),(0,1))) |
| Localized: j loop only (i.e. 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