Memory Hierarchy Optimizations

Todd C. Mowry
CS745: Optimizing Compilers

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
- Internals:
- Global:
- Procedure arguments
- Is cache performance a concern here?
- If so, what can be done?

```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?

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

Arrays

- 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

```
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
When Do Cache Misses Occur?

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

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

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

Impact on Visitation Order in Iteration Space

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

```
for JJ = 0 to N-1 by B
  for i = 0 to N-1
    for j = JJ to max(N-1, JJ+B-1)
      for k = 0 to N-1
        c[i,k] += a[i,j]*b[j,k];
```

Cache Blocking (aka “Tiling”)

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

```
for JJ = 0 to N-1 by B
  for i = 0 to N-1
    for jj = 0 to max(N-1, JJ+B-1)
      f(A[i], A[j]);
```

Cache Blocking in Two Dimensions

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

```
for KK = 0 to N-1 by B
  for i = 0 to N-1
    for j = max(0, KK-B) to max(N-1, KK+B-1)
      for k = 0 to N-1
        c[i,k] += a[i,j]*b[j,k];
```

• 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 and localized iteration space \(\Rightarrow\) locality

Types of Data Reuse/Locality

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

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:
  \[\hat{f}(v) = Hv + c\]

\[A[i][j] = A \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\]
\[B[j][0] = B \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\]
\[B[j+1][0] = B \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\]
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 of \( \vec{v}_1 \) and \( \vec{v}_2 \), 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

- Reuse between iterations \((i_1,j_1)\) and \((i_2,j_2)\) whenever:
  \[
  \text{True whenever } j_1 = j_2, \text{ and regardless of the difference between } i_1 \text{ and } i_2.
  \]
  - i.e. whenever the difference lies along the nullspace of \( H \), which is \( \text{span}\{(1,0)\} \) (i.e. the outer loop).

More Complicated Example

- \[
  \text{for } i = 0 \text{ to } N-1 \text{ and } j = 0 \text{ to } N-1 \\
  A[i+j][0] = i*j;
  \]

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

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

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

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

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

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

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

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

Localized Iteration Space

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

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

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

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

- 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