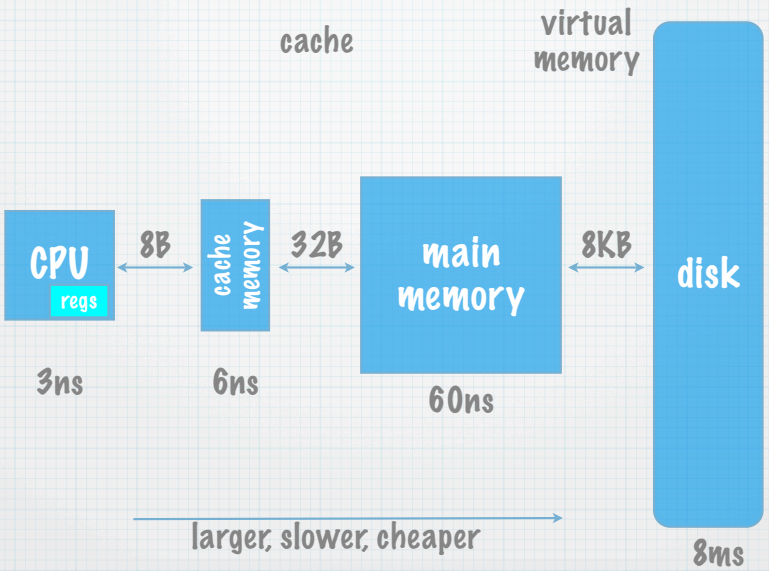
Memory Optimizations

15-745 Optimizing Compilers
Spring 2006

Peter Lee

Reminders

* T3 is due this week
  * the last one! ;-)
* Suggested projects posted
* Reading list posted



Improving cache performance

* Things to enhance
  * **temporal locality**
  * **spatial locality**
* Things to minimize
  * **conflicts** (i.e., bad replacement decisions)

What the compiler can do

- Time:
  - when is an object accessed?
- Space:
  - where does an object exist in the address space?
- These are the two "levers" a compiler can try to manipulate

Manipulating time and space

- Time: Reordering computation
  - try to determine when an object will be accessed, and predict a better time to access it
- Space: Changing data layout
  - determine an object's shape and location, and determine a better layout

Types of objects

- For small objects (scalars) and pointer-based objects, manipulating temporal and spatial locality is both difficult and usually not fruitful
- Arrays, on the other hand, often provide many opportunities for significant optimizations

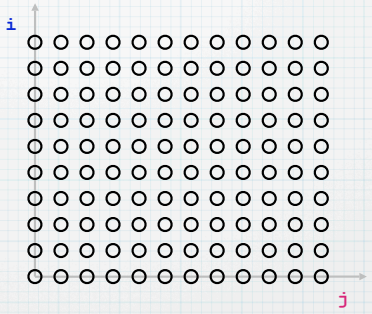
Arrays

- Often accessed within loop nests
  - makes it easy to understand "time"
- Have predictable layout
  - makes it easy to understand "space"

```
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];
```

Iteration space

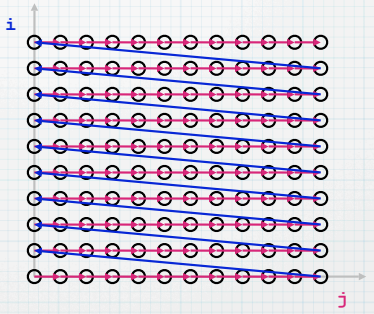
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



Each position represents a loop iteration.
Note: iteration space ≠ data space!

Iteration space

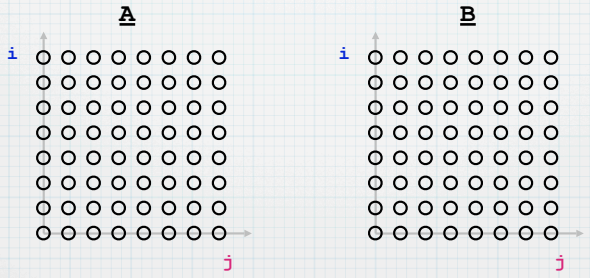
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



The visitation order

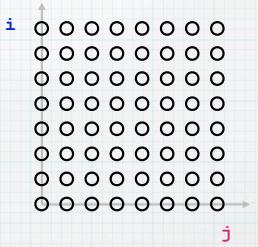
When do cache misses occur?

```
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][0] = i*j;
```



Optimizing cache behavior

* When do cache misses occur?
  * use **locality analysis**
* Can we change the visitation order to produce better behavior?
  * evaluate costs
* Does the new visitation order still produce correct results?
  * use **dependence analysis**

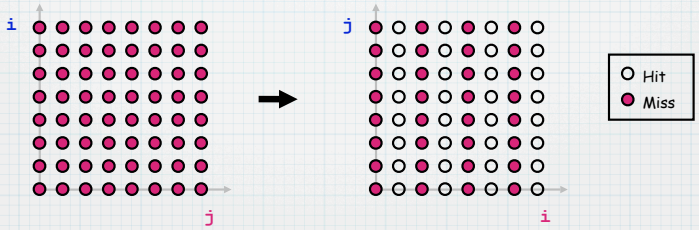
Loop transformations

* Many different ones proposed
  * loop interchange
  * cache blocking
  * skewing
  * loop reversal
  * ...

Loop interchange

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

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

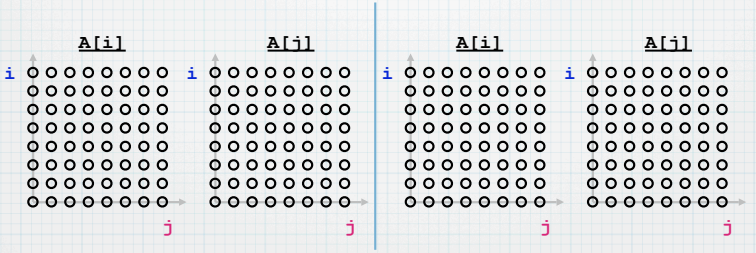


(assuming N is large, relative to the cache line size)

Cache blocking

```
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)
         f(A[i],A[j]);
```



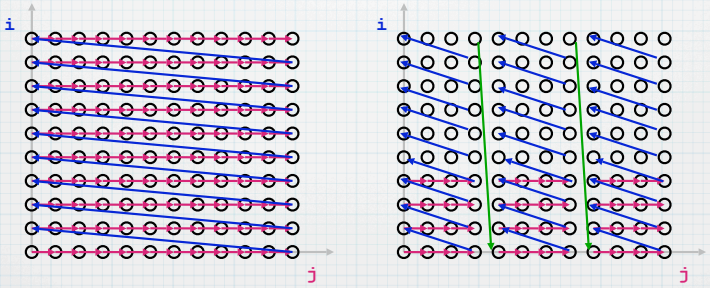
Also known as "tiling"

Impact of tiling on visit order

```
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)
         f(A[i],A[j]);
```



Tiling in two dimensions

```
for i = 0 to N-1
  for j = 0 to N-1
    for k = 0 to N-1
      c[i,k] += a[i,j]*b[j,k];
```

→

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

The goal is to bring square sub-blocks of b into the cache, using them up before moving on

Locality analysis

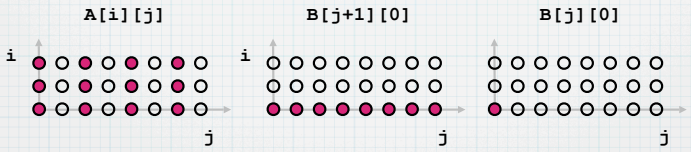
- Reuse: accessing a location that has been accessed previously
- Locality: accessing a location that is in the cache
- Observe:
  - locality only occurs when there is reuse!
  - but reuse does not imply locality

Steps in locality analysis

- Find data reuse
- Determine "localized iteration space"
  - set of inner loops where the data accessed by an iteration is expected to fit within the cache
- Find data locality
  - reuse ∩ localized iteration space ⇒ locality

Types of reuse/locality

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



spatial temporal group

Reuse analysis

```
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 the array indexing function: $\vec{f}(\vec{\imath}) = H\vec{\imath} + \vec{c}$

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

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

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

Finding temporal reuse

* Temporal reuse occurs between iterations $\vec{\imath}_1$ and $\vec{\imath}_2$ whenever $H\vec{\imath}_1 + \vec{c} = H\vec{\imath}_2 + \vec{c}$

  $H(\vec{\imath}_1 - \vec{\imath}_2) = \vec{0}$

* Rather than worrying about individual values of $\vec{\imath}_1$ and $\vec{\imath}_2$, we say that reuse occurs along direction vector $\vec{r}$ when

  $H(\vec{r}) = \vec{0}$

* Solve by computing 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 $\begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\begin{bmatrix}i_1\\ j_1\end{bmatrix} + \begin{bmatrix}1\\ 0\end{bmatrix} = \begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\begin{bmatrix}i_2\\ j_2\end{bmatrix} + \begin{bmatrix}1\\ 0\end{bmatrix}$

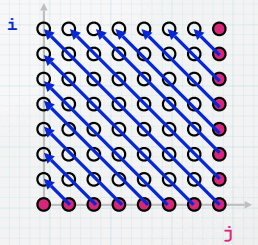
  $\begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\begin{bmatrix}i_1 - i_2\\ j_1 - j_2\end{bmatrix} = \begin{bmatrix}0\\ 0\end{bmatrix}$

* True whenever j_1=j_2, and regardless of difference between i_1 and i_2
  * i.e., when difference lies along the nullspace of $\begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}$, which is span{(1,0)}

A more complicated example

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

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



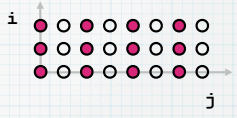
Nullspace of $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$ = span{(1,-1)}

Computing spatial reuse

* Replace last row of H with zeroes, creating H_s
* Find the nullspace of H_s
* Result: vector along which we access the same row

Spatial reuse example

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



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

$$H_s = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

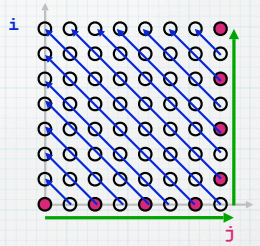
Nullspace of Hs = span{(0,1)}
- i.e., access same row of A[i][j] along inner loop

More complicated example

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

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

$$H_s = \begin{bmatrix} 0 & 0 \end{bmatrix}$$



Nullspace of H = span{(1,-1)}

Nullspace of H_s = span{(1,0),(0,1)}

Group reuse

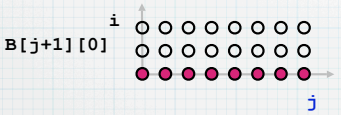
```
for i = 0 to 2
  for j = 0 to 100
    A[i][j] = B[j][0] + B[j+1][0];
```

- Consider only "uniformly generated sets"
  - i.e., index expressions differ only by constant terms
- Check whether they access the same cache line
- Only the leading reference suffers the bulk of the cache misses

Localized iteration space

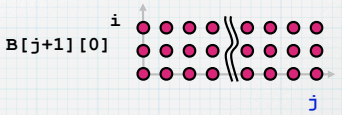
Given finite cache, when does reuse result in locality?

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



localized in both i and j loops

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



localized in j loop only

Computing locality

- Reuse vector space ∩ localized vector space ⇒ 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)} ∩ span{(1,0),(0,1)} ⇒ span{(1,0)}
    - ie, temporal reuse results in temporal locality
- If only innermost loop is localized:
  - span{(1,0)} ∩ span{(0,1)} ⇒ span{}
    - ie, no temporal locality