Dependence Analysis & Memory Hierarchy Optimizations

Todd C. Mowry/Seth Goldstein/Tim Callahan
CS745: Optimizing Compilers
Spring 2007

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?

An Example Memory Hierarchy

- Registers: 32 registers x 8 Bytes real reg
- Cache: 512 lines x 128 bytes (64kB)
- TLB: 128 entries
- L2: up to 4MB
- Main Memory: 32K pages x 4k bytes
- DISK: 1M pages x 8k bytes

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**
- **Globals**
- **Procedure arguments**

- Is cache performance a concern here?
- If so, what can be done?

```
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**
  - Considering cache?

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

Arrays

- Usually accessed within **loop 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];
```

Data Dependence in Loops

- Dependence can flow across iterations of the loop.
- Dependence information is annotated with iteration information.
- If dependence is across iterations, it is **loop carried** otherwise **loop independent**.

```
for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
}
```
Data Dependence in Loops

- Dependence can flow across iterations of the loop.
- Dependence information is annotated with iteration information.
- If dependence is across iterations it is loop carried otherwise loop independent.

```
for (i=0; i<n; i++) {
  A[i] = B[i];
  B[i+1] = A[i];
}
```

δ loop carried
δ loop independent

Distance/Direction of the dependence is also important.

Iteration Space

Every iteration generates a point in an n-dimensional space, where n is the depth of the loop nest.

```
for (i=0; i<n; i++) {
  for (j=0; j<4; j++) {
    ...
  }
}
```

Distance vector is the difference between the target and source iterations.

Distance/Direction of the dependence is also important.

```
for (i=0; i<n; i++) {
  A[i] = B[i];
  B[i+1] = A[i];
}
```

Distance vector is the difference between the target and source iterations.

```
A[0] = B[0];
B[1] = A[0];
```

\[ d = l_t - l_s \]

Exactly the distance of the dependence, i.e.,

\[ l_s + d = l_t \]
Example of Distance Vectors

```
for (i=0; i<n; i++)
    for (j=0; j<m; j++){
        A[i][j] = i
        B[i][j+1] = j
        C[i+1][j] = k
        
        A[i][j] = A[i][j]
        B[i][j+1] = B[i][j]
        C[i+1][j] = C[i][j+1]
    }

A yields: [0,0]  B yields: [0,1]  C yields: [1,1]
```

Direction Vectors

Less exact than distance vectors
- can’t analyze exactly, or
- summary of multiple distance vectors

```
[0,0]   [1,inf]   [-inf,-1]   [-inf,inf]
=       +         -           +/-
<       >         *
```

Example:
- \{<1,-1>, <1,0>, <1,1>\} \rightarrow \{1,*\}

Handy Representation:
“Iteration Space”

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

- each position represents an iteration
**Visitation Order in Iteration Space**

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

- Note: iteration space is not data space

**When Do Cache Misses Occur?**

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

**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
- ...

Loop Interchange

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

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

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

Cache Blocking (aka “Tiling”)

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

\[
\begin{align*}
\text{for } i &= 0 \text{ to } N-1 \\
\text{for } j &= J_0 \text{ to } \max(N-1, J_0 + B-1) \\
f(A[i], A[j]);
\end{align*}
\]

now we can exploit temporal locality

Impact on Visitation Order in Iteration Space

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

\[
\begin{align*}
\text{for } i &= 0 \text{ to } N-1 \\
\text{for } j &= J_0 \text{ to } \max(N-1, J_0 + B-1) \\
f(A[i], A[j]);
\end{align*}
\]
**Cache Blocking in Two Dimensions**

- brings square sub-blocks of matrix "b" into the cache
- completely uses them up before moving on

```c
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];
```

**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

**Types of Data Reuse/Locality**

- **Spatial Temporal Group** (spatial)

```
for i = 0 to 2
    for j = 0 to 100
        A[i][j] = B[j][0] + B[j+1][0];
```
But...is the transform legal?

- Distance/direction vectors give a partial order among points in the iteration space
- A loop transform changes the order in which 'points' are visited
- The new visit order must respect the dependence partial order!

But...is the transform legal?

- Loop reversal ok?
- Loop interchange ok?

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

But...is the transform legal?

- What other visit order is legal here?

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

```plaintext
for i = 0 to TS
  for j = 0 to N-2
```
But...is the transform legal?

- What other visit order is legal here?

\[
\begin{align*}
\text{for } i &= 0 \text{ to } TS \\
\text{for } j &= 0 \text{ to } N-2 \\
\end{align*}
\]

But...is the transform legal?

- Skewing...

But...is the transform legal?

- Skewing...now we can block

But...is the transform legal?

- Skewing...now we can loop interchange
### Unimodular transformations

- Express loop transformation as a matrix multiplication
- Check if any dependence is violated by multiplying the distance vector by the matrix - if the resulting vector is still lexicographically positive, then the involved iterations are visited in an order that respects the dependence.

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reversal</td>
<td>\begin{pmatrix} 1 &amp; 0 \ 0 &amp; -1 \end{pmatrix}</td>
</tr>
<tr>
<td>Interchange</td>
<td>\begin{pmatrix} 0 &amp; 1 \ 1 &amp; 0 \end{pmatrix}</td>
</tr>
<tr>
<td>Skew</td>
<td>\begin{pmatrix} 1 &amp; 1 \ 0 &amp; 1 \end{pmatrix}</td>
</tr>
</tbody>
</table>

“A Data Locality Optimizing Algorithm”, M.E. Wolf and M. Lam

### Other uses?

- Of course - many!
- Removing intra- and inter-loop dependence edges
  - i.e. token edges in Pegasus
- Expose more instruction level parallelism
- Enable streaming, vectorization,......

### Garpcce demo

- Dependence analysis uses:
  - more scheduling flexibility
  - determine when it’s legal to use memory queues
- SUIF’s dependence library
  - many tests; if any can prove independence, then the accesses are independent

### Garpcce demo

- What I would want:
  - Loop interchange & reversal to enable queue use in the inner loop
Scalar Replacement

- Replaces subscripted array references with scalars.
- AKA: register pipelining
- Benefits:
  - Improved D$ performance
  - Register allocation made possible
  - Easier to software pipeline

Example: MM

```c
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

- replace C[i][j] with scalar in inner loop.
- Reduces memory references by 2(N^3 - N^2)

Scalar Replacement data structures

- Lets consider loops without conditionals
- Define the period of a loop carried dependence for edge e, p(e), as the CONSTANT number of iterations between the references at tail and head. (If not constant we can’t do it).
- Build a partial dependence graph including
  - flow (R after W) and
  - input dependencies (R after R)
And the dependencies
- have a constant period
- are:
  - loop independent or
  - carried by innermost loop
Scalar Replacement Alg

- For a period of p(e) cycles, use p(e)+1 temporaries t₀ to tₚₑ
- In body of loop:
  - Replace A[i] with t₀
  - Replace A[i+j] with t_j
- At end of innermost loop body add assignments tₚₑ = tₚₑ₋₁; ... ; t₁ = t₀
- Init temps by peeling off p(e) iterations

Scalar Replacement: Loop Body

```c
for (i=0; i<n; i++) {
    b[i+1] = b[i] + f
    a[i] = 2 * b[i] + c[i]
}
```

- We need two temporaries: t₀, t₁
- Replace b[i] with t₀ and b[i+1] with t₁
- Insert copies at bottom of loop

Scalar Replacement: Init

```c
for (i=0; i<n; i++) {
    t₁ = t₀ + f
    b[i+1] = t₁
    a[i] = 2 * t₀ + c[i]
    t₀ = t₁
}
```

Example: MM

```c
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            A[i][k]*B[k][j];
```

- replace C[i][j] with scalar in inner loop.
- Reduces memory references by 2(N³-N²)

Scalar Replacement: Init

```c
if (n>=0) {
    t₀ = b[0]
    t₁ = t₀ + f
    b[1] = t₁
    a[0] = 2 * t₀ + c[0]
}
```

2) after replacement

```c
t₀ = b[0]
t₁ = t₀ + f
b[1] = t₁
a[0] = 2 * t₀ + c[0]
```

3) If we aren’t sure of trip count

```c
if (n>=0) {
    t₀ = b[0]
    t₁ = t₀ + f
    b[1] = t₁
    a[0] = 2 * t₀ + c[0]
}
```
if (n>=0) {
    t0 = b[0]
    t1 = t0 + f
    b[1] = t1
    a[0] = 2 * t0 + c[0]
}
for (i=1; i<n; i++) {
    t1 = t0 + f
    b[i+1] = t1
    a[i] = 2 * t0 + c[i]
    t0 = t1
}

for (i=0; i<n; i++) {
    b[i+1] = b[i] + f
    a[i] = 2 * b[i] + c[i]
}