Dependence Analysis and Loop Transformations

John Cavazos
University of Delaware
Lecture Overview

- Very Brief Introduction to Dependences
- Loop Transformations
The Big Picture

What are our goals?

- Simple Goal: Make execution time as small as possible

Which leads to:

- Achieve execution of many (all, in the best case) instructions in parallel
- Find independent instructions
Dependences

- We will concentrate on data dependences
- Simple example of data dependence:

  \[ S_1 \quad PI = 3.14 \]
  \[ S_2 \quad R = 5.0 \]
  \[ S_3 \quad AREA = PI \times R \times 2 \]

- Statement \( S_3 \) cannot be moved before either \( S_1 \) or \( S_2 \) without compromising correct results
Dependences

- Formally:
  There is a data dependence from statement $S_1$ to statement $S_2$ ($S_2$ depends on $S_1$) if:
  1. Both statements access the same memory location and at least one of them stores onto it, and
  2. There is a feasible run-time execution path from $S_1$ to $S_2$
Load Store Classification

- Quick review of dependences classified in terms of load-store order:
  1. True dependence (RAW hazard)
  2. Antidependence (WAR hazard)
  3. Output dependence (WAW hazard)
Dependence in Loops

Let us look at two different loops:

```
DO I = 1, N
  A(I+1) = A(I) + B(I)
ENDDO
```

```
DO I = 1, N
  A(I+2) = A(I) + B(I)
ENDDO
```

- In both cases, statement $S_1$ depends on itself
Transformations

- We call a transformation safe if the transformed program has the same "meaning" as the original program.

- But, what is the "meaning" of a program?

For our purposes:

- Two computations are equivalent if, on the same inputs:
  - They produce the same outputs in the same order.
Reordering Transformations

- Is any program transformation that changes the order of execution of the code, without adding or deleting any executions of any statements
Properties of Reordering Transformations

- A reordering transformation does not eliminate dependences.
- However, it can change the ordering of the dependence which will lead to incorrect behavior.
- A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence.
Loop Transformations

- Compilers have always focused on loops
  - Higher execution counts
  - Repeated, related operations
- Much of real work takes place in loops
Several effects to attack

- **Overhead**
  - Decrease control-structure cost per iteration

- **Locality**
  - Spatial locality $\Rightarrow$ use of co-resident data
  - Temporal locality $\Rightarrow$ reuse of same data

- **Parallelism**
  - Execute independent iterations of loop in parallel
Eliminating Overhead

Loop unrolling (the oldest trick in the book)

- To reduce overhead, replicate the loop body

\[
\begin{align*}
\text{do } i & = 1 \text{ to } 100 \text{ by } 1 \\
& a(i) = a(i) + b(i) \\
\text{end}
\end{align*}
\]

becomes

\[
\begin{align*}
\text{do } i & = 1 \text{ to } 100 \text{ by } 4 \\
a(i) & = a(i) + b(i) \\
a(i+1) & = a(i+1) + b(i+1) \\
a(i+2) & = a(i+2) + b(i+2) \\
a(i+3) & = a(i+3) + b(i+3) \\
\text{end}
\end{align*}
\]

(unroll by 4)

Sources of Improvement

- Less overhead per useful operation
- Longer basic blocks for local optimization
Eliminating Overhead

Loop unrolling with unknown bounds

- Generate guard loops

```
do i = 1 to n by 1
  a(i) = a(i) + b(i)
end
```

becomes

(roll out by 4)

```
i = 1
do while (i+3 < n)
  a(i) = a(i) + b(i)
  a(i+1) = a(i+1) + b(i+1)
  a(i+2) = a(i+2) + b(i+2)
  a(i+3) = a(i+3) + b(i+3)
  i = i + 4
end

do while (i < n)
  a(i) = a(i) + b(i)
  i = i + 1
end
```
Eliminating Overhead

One other use for loop unrolling

- Eliminate copies at the end of a loop

\[
\begin{align*}
t_1 &= b(0) \\
do \ i &= 1 \ \text{to} \ 100 \\
t_2 &= b(i) \\
a(i) &= a(i) + t_1 + t_2 \\
t_1 &= t_2 \\
end
\end{align*}
\]

becomes

\[
\begin{align*}
t_1 &= b(0) \\
do \ i &= 1 \ \text{to} \ 100 \ \text{by} \ 2 \\
t_2 &= b(i) \\
a(i) &= a(i) + t_1 + t_2 \\
t_1 &= b(i+1) \\
a(i+1) &= a(i+1) + t_2 + t_1 \\
end
\end{align*}
\]
Loop Unswitching

- Hoist invariant control-flow out of loop nest
- Replicate the loop & specialize it
- No tests, branches in loop body
- Longer segments of straight-line code
Loop Unswitching

If test then
  loop
    statements
    if test then
      then part
    else
      else part
    endif
    more statements
  endloop
else
  loop
    statements
    then part
    more statements
  endloop
endif

*
Loop Unswitching

\[
\text{do } i = 1 \text{ to } 100 \\
\quad a(i) = a(i) + b(i) \\
\quad \text{if (expression) then} \\
\quad \quad d(i) = 0 \\
\text{end}
\]

becomes

\[
\text{if (expression) then} \\
\quad \text{do } i = 1 \text{ to } 100 \\
\quad \quad a(i) = a(i) + b(i) \\
\quad \quad d(i) = 0 \\
\text{end} \\
\text{else} \\
\quad \text{do } i = 1 \text{ to } 100 \\
\quad \quad a(i) = a(i) + b(i) \\
\text{end}
\]
Loop Fusion

- Two loops over same iteration space ⇒ one loop
- Safe if does not change the values used or defined by any statement in either loop (i.e., does not violate deps)

\[
\begin{align*}
\text{do } i &= 1 \text{ to } n \\
c(i) &= a(i) + b(i) \\
\text{end} & \quad \text{(fuse)} \\
\text{do } j &= 1 \text{ to } n \\
d(j) &= a(j) \times e(j) \\
\text{end}
\end{align*}
\]

For big arrays, \( a(i) \) may not be in the cache

\( a(i) \) will be found in the cache
Loop Fusion Advantages

- Enhance temporal locality
- Reduce control overhead
- Longer blocks for local optimization & scheduling
- Can convert inter-loop reuse to intra-loop reuse
Loop Fusion of Parallel Loops

- Parallel loop fusion legal if dependences loop independent
  - Source and target of flow dependence map to same loop iteration
Loop distribution (fission)

- Single loop with independent statements $\Rightarrow$ multiple loops
- Starts by constructing statement level dependence graph
- Safe to perform distribution if:
  - No cycles in the dependence graph
  - Statements forming cycle in dependence graph put in same loop
Loop distribution (fission)

\[
\begin{align*}
\text{Reads b, c, e, f, h, & k} & \quad \left\{ \begin{align*}
\text{do } i &= 1 \text{ to } n \\
\text{a(i) = b(i) + c(i)} \\
\text{d(i) = e(i) * f(i)} \\
\text{g(i) = h(i) - k(i)} \\
\text{end}
\end{align*} \right. \\
\text{becomes (fission)} & \quad \left\{ \begin{align*}
\text{do } i &= 1 \text{ to } n \\
\text{a(i) = b(i) + c(i)} \\
\text{d(i) = e(i) * f(i)} \\
\text{g(i) = h(i) - k(i)} \\
\text{end}
\end{align*} \right.
\end{align*}
\]

\[
\text{Reads b & c} \quad \rightarrow \quad \text{Reads e & f} \quad \rightarrow \quad \text{Reads h & k} \quad \rightarrow \quad \text{Reads a, d, & g} \quad \rightarrow \quad \text{Writes a} \quad \rightarrow \quad \text{Writes d} \quad \rightarrow \quad \text{Writes g}
\]
Loop distribution (fission)

(1) for \( l = 1 \) to \( N \) do
(3) \( B[l] = C[l-1]*X+C \)
(4) \( C[l] = 1/B[l] \)
(5) \( D[l] = \sqrt{C[l]} \)
(6) endfor

Has the following dependence graph
Loop distribution (fission)

(1) for I = 1 to N do
(3) B[I] = C[I-1]*X+C
(4) C[I] = 1/B[I]
(5) D[I] = sqrt(C[I])
(6) endfor

(1) for I = 1 to N do
(3) endfor
(4) for
(5) B[I] = C[I-1]*X+C
(6) C[I] = 1/B[I]
(7) endfor
(8) for
(9) D[I] = sqrt(C[I])
(10) endfor
Loop Fission Advantages

- Enables other transformations
  - E.g., Vectorization
- Resulting loops have smaller cache footprints
  - More reuse hits in the cache
Loop Interchange

- Swap inner & outer loops to rearrange iteration space
- Improves reuse by using more elements per cache line
- Goal is to get as much reuse into inner loop as possible

\[
\begin{align*}
&\text{do } i = 1 \text{ to } 50 \\
&\quad \text{do } j = 1 \text{ to } 100 \\
&\quad \quad a(i,j) = b(i,j) \times c(i,j) \\
&\text{end} \\
&\text{end}
\end{align*}
\]

\[
\begin{align*}
&\text{do } j = 1 \text{ to } 100 \\
&\quad \text{do } i = 1 \text{ to } 50 \\
&\quad \quad a(i,j) = b(i,j) \times c(i,j) \\
&\text{end} \\
&\text{end}
\end{align*}
\]

becomes

\[
\begin{align*}
&\text{do } j = 1 \text{ to } 100 \\
&\quad \text{do } i = 1 \text{ to } 50 \\
&\quad \quad a(i,j) = b(i,j) \times c(i,j) \\
&\text{end} \\
&\text{end}
\end{align*}
\]
Loop Interchange Effect

- If one loop carries all dependence relations
  - Swap to outermost loop and all inner loops executed in parallel
- If outer loops iterates many times and inner only a few
  - Swap outer and inner loops to reduce startup overhead
- Improves reuse by using more elements per cache line
- Goal is to get as much reuse into inner loop as possible
Reordering Loops for Locality

In row-major order, the opposite loop ordering causes the same effects.

In Fortran's column-major order, \( a(4,4) \) would lay out as:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1,1</td>
<td>2,1</td>
<td>3,1</td>
<td>4,1</td>
</tr>
<tr>
<td>1,2</td>
<td>2,2</td>
<td>3,2</td>
<td>4,2</td>
</tr>
<tr>
<td>1,3</td>
<td>2,3</td>
<td>3,3</td>
<td>4,3</td>
</tr>
<tr>
<td>1,4</td>
<td>2,4</td>
<td>3,4</td>
<td>4,4</td>
</tr>
</tbody>
</table>

After interchange, direction of Iteration is changed:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1,1</td>
<td>2,1</td>
<td>3,1</td>
<td>4,1</td>
</tr>
<tr>
<td>1,2</td>
<td>2,2</td>
<td>3,2</td>
<td>4,2</td>
</tr>
<tr>
<td>1,3</td>
<td>2,3</td>
<td>3,3</td>
<td>4,3</td>
</tr>
<tr>
<td>1,4</td>
<td>2,4</td>
<td>3,4</td>
<td>4,4</td>
</tr>
</tbody>
</table>

As little as 1 used element per line Runs down cache line
Loop permutation

- Interchange is degenerate case
  - Two perfectly nested loops
- More general problem is called permutation

Safety

- Permutation is safe iff no data dependences are reversed
  - The flow of data from definitions to uses is preserved
Loop Permutation Effects

- Change order of access & order of computation
- Move accesses closer in time $\Rightarrow$ increase temporal locality
- Move computations farther apart $\Rightarrow$ cover pipeline latencies
Strip Mining

- Splits a loop into two loops

\[
\text{do } j = 1 \text{ to } 100 \\
\text{do } i = 1 \text{ to } 50 \\
a(i,j) = b(i,j) \times c(i,j) \\
\text{end}
\]

becomes

\[
\text{do } j = 1 \text{ to } 100 \\
\text{do } ii = 1 \text{ to } 50 \text{ by } 8 \\
\text{do } i = ii \text{ to } \min(ii+7,50) \\
a(i,j) = b(i,j) \times c(i,j) \\
\text{end} \\
\text{end} \\
\text{end}
\]

Note: This is always safe, but used by itself not profitable!
Strip Mining Effects

- May slow down the code (extra loop)
- Enables vectorization
Loop Tiling (blocking)

\[
\begin{align*}
& \text{do } t = 1, T \\
& \quad \text{do } i = 1, n \\
& \quad \quad \text{do } j = 1, n \\
& \quad \quad \quad \ldots \ a(i,j) \ \ldots \\
& \quad \quad \text{end do} \\
& \quad \text{end do} \\
& \text{end do}
\end{align*}
\]

Want to exploit temporal locality in loop nest.
Loop Tiling (blocking)

```
do ic = 1, n, B
  do jc = 1, n, B
    do t = 1, T
      do i = ic, min(n,ic+B-1), 1
        do j = jc, min(n, jc+B-1), 1
          ... a(i,j) ...
        end do
      end do
    end do
  end do
end do
```

B: Block Size
Loop Tiling (blocking)

```plaintext
do ic = 1, n, B
  do jc = 1, n, B
    do t = 1, T
      do i = ic, min(n,ic+B-1), 1
        do j = jc, min(n, jc+B-1), 1
          ... a(i,j) ...
        end do
      end do
    end do
  end do
end do
```

B: Block Size
Loop Tiling (blocking)

```
do ic = 1, n, B
  do jc = 1, n, B
    do t = 1, T
      do i = ic, min(n,ic+B-1), 1
        do j = jc, min(n, jc+B-1), 1
          ... a(i,j) ...
        end do
      end do
    end do
  end do
end do
```

B: Block Size
Loop Tiling (blocking)

do ic = 1, n, B
    do jc = 1, n, B
        do t = 1, T
            do i = ic, min(n, ic+B-1), 1
                do j = jc, min(n, jc+B-1), 1
                    ... a(i,j) ...
                end do
            end do
        end do
    end do
end do

B: Block Size
When is this legal?
Loop Tiling Effects

- Reduces volume of data between reuses
  - Works on one “tile” at a time (tile size is B by B)
- Choice of tile size is crucial
Scalar Replacement

- Allocators never keep c(i) in a register
- We can trick the allocator by rewriting the references

The plan
- Locate patterns of consistent reuse
- Make loads and stores use temporary scalar variable
- Replace references with temporary’s name
Scalar Replacement

\[
\begin{align*}
d& i = 1 \text{ to } n \\
& \quad \begin{align*}
& \quad \text{do } j = 1 \text{ to } n \\
& \quad \quad a(i) = a(i) + b(j) \\
& \quad \end{align*} \\
& \quad \text{end}
\end{align*}
\]

\[
\begin{align*}
d& i = 1 \text{ to } n \\
& \quad t = a(i) \\
& \quad \begin{align*}
& \quad \text{do } j = 1 \text{ to } n \\
& \quad \quad t = t + b(j) \\
& \quad \end{align*} \\
& \quad a(i) = t \\
& \quad \text{end}
\end{align*}
\]

becomes

Almost any register allocator can get \( t \) into a register
Scalar Replacement Effects

- Decreases number of loads and stores
- Keeps reused values in names that can be allocated to registers
- In essence, this exposes the reuse of a(i) to subsequent passes