Dependence Analysis

CSE 501
Lecture 18
June 3, 2013
Dependences

Last time we talked about dependences and how they are useful

This time we'll talk about dependence analysis; that is, given a pair of memory references, how do they depend on each other?

We'll especially focus on array references in common loops.
Representation

What's a dependence look like?

Perhaps a struct with several fields:
- source and destination
- kind (true, anti, output, maybe input)
- levels
- distance/direction vector
- flags for consistent & loop independent

Entries in the direction vector might look like
- direction
- distance
- flag for scalar
Saving space

Many dependences are confused. So let's differentiate between a confused dependence, where we know almost nothing (e.g., may alias)

```cpp
class Dependence {
  private:
    const Instruction *src, *dst;
};
```

and a full dependence, where we know many things

```cpp
class FullDependence : public Dependence {
  private:
    unsigned levels;
    bool consistent;
    Level *dv;
};
```

We'll define appropriate constructors and a number of public methods.
Methods

Here's a reasonable selection. Might add a few more, depending on the exact xforms you plan to implement.

```cpp
bool isInput() const;
bool isOutput() const;
bool isFlow() const;
bool isAnti() const;
bool isLoopIndependent() const;
bool isConfused() const;
bool isConsistent() const;
unsigned getLevels() const;
unsigned getDirection(unsigned level) const;
unsigned const *SCEV *getDistance(unsigned level) const;
bool isScalar(unsigned level) const;
```

The last three show what we need for each level.
Level

We might define Level like this

```c
struct Level {
    enum {NONE=0, LT=1, EQ=2, LE=3, GT=4, NE=5, GE=6, ALL=7};
    unsigned char direction;
    const *SCEV *distance;
    bool scalar;
};
```

where SCEV is scalar evolution, LLVM's symbolic arithmetic package. Google "chains of recurrences"
Array references, subscripts, and indices

We test for dependence between a pair of array references in some common loops.

\[ A(i, j+1, 2*k) \text{ and } A(i-1, k, j) \]

A subscript is a \textit{pair} of subscript positions in a pair of array references

\[ i \text{ and } i-1 \quad j+1 \text{ and } k \quad 2*k \text{ and } j \]

An index is the loop index for some loop surrounding a pair of references.
Conservative testing

Consider only linear subscript expressions.

Finding solutions to systems of linear Diophantine equations is, in general, NP-Complete

Most common approximation is conservative testing.
See if we can prove:

*No dependence exists between two subscripted references of the same array*

Never incorrect; may be less than optimal.
Complexity

A subscript is said to be

• ZIV if it contains no index
• SIV if it contains only one index
• MIV if it contains more than one index

For example, consider $A(5, i+1, j)$ and $A(1, i, k)$

• The first subscript is ZIV
• The second subscript is SIV
• The third subscript is MIV

Here's another, $A(5, i+1, 2)$ and $A(i, i, i+j)$

• The 1st subscript is SIV
• The 2nd subscript is SIV
• The 3rd subscript is MIV
Separability

A subscript is separable if its indices do not occur in other subscripts.

If two subscripts contain the same index, they are coupled.

For example

\[ A(i+1, j) \text{ and } A(k, j) \]

Both subscripts are separable.

\[ A(i, j, j) \text{ and } A(i, j, k) \]

The 1st subscript is separable, the 2nd and 3rd subscripts are coupled.
Coupled subscript groups

Ignoring coupling can cause imprecision in dependence testing.

Consider

\[
\begin{align*}
\text{do } & i = 1, 100 \\
& A(i+1, i) = B(i) + C \\
& D(i) = A(i, i) + E \\
\text{enddo}
\end{align*}
\]

If we consider the subscripts of A separately, there appears to be a dependence.

If we consider them together, there's clearly no dependence.
Overview

Given a pair of array references,
• Partition the subscripts into separable and coupled groups
Overview

Given a pair of array references,
- Partition the subscripts into separable and coupled groups
- Classify each subscript as ZIV, SIV, or MIV
Overview

Given a pair of array references,

- Partition the subscripts into separable and coupled groups
- Classify each subscript as ZIV, SIV, or MIV
- For each separable subscript, apply single subscript test. If any test proves independence, we're done.
Overview

Given a pair of array references,

- Partition the subscripts into separable and coupled groups
- Classify each subscript as ZIV, SIV, or MIV
- For each separable subscript, apply single subscript test. If any test proves independence, we're done.
- For each coupled group, apply multiple subscript test, e.g. Delta Test
Overview

Given a pair of array references,
• Partition the subscripts into separable and coupled groups
• Classify each subscript as ZIV, SIV, or MIV
• For each separable subscript, apply single subscript test. If any test proves independence, we're done.
• For each coupled group, apply multiple subscript test, e.g. Delta Test
• If we still haven't proven independence, merge all direction vectors computed in earlier steps into a single set.
Single subscript tests

It's worthwhile to handle special cases carefully.

- ZIV test
- SIV test
  - Strong SIV test
  - Weak SIV test
  - Weak-zero SIV
  - Weak-crossing SIV
ZIV test

Consider

    do j = 1, 100
        A(e1) = A(e2) + B
    enddo

$e_1$ and $e_2$ are constants or loop invariants

There are three possible results:
- If $e_1 - e_2 \neq 0$, then no dependence exists
- If $e_1 - e_2 = 0$, then a dependence certainly exists
- If we're unsure about the relationship between $e_1$ and $e_2$, then a dependence may exist.
SIV tests

SIV subscripts are generally of the form

\[ a^i+c1 \text{ and } b^i+c1 \]

where \( a, b, c1, \) and \( c2 \) are loop invariant (and possibly constant).

While we can sometimes prove some facts about this general case, we can often prove more in (common) special cases.
A geometric interpretation

There are a couple of plausible geometric interpretations of the SIV test. I like this one, especially for creating examples.

Suppose we're looking at $f(i)$ and $g(i)$.

For there to be a dependence, $f(i)$ must equal $g(i')$ for some integer values of $i$ and $i'$ in the interval $[L, U]$

Plot $f(i) = g(i')$
A geometric interpretation

There are a couple of plausible geometric interpretations of the SIV test. I like this one, especially for creating examples.

Suppose we're looking at \( f(i) \) and \( g(i') \).

For there to be a dependence, \( f(i) \) must equal \( g(i') \) for some integer values of \( i \) and \( i' \) in the interval \([L, U]\)

Plot \( f(i) = g(i') \)
A geometric interpretation

There are a couple of plausible geometric interpretations of the SIV test. I like this one, especially for creating examples.

Suppose we're looking at \( f(i) \) and \( g(i') \).

For there to be a dependence, \( f(i) \) must equal \( g(i') \) for some integer values of \( i \) and \( i' \) in the interval \([L, U]\)

Plot \( f(i) = g(i') \)
A geometric interpretation

There are a couple of plausible geometric interpretations of the SIV test. I like this one, especially for creating examples.

Suppose we're looking at $f(i)$ and $g(i)$.

For there to be a dependence, $f(i)$ must equal $g(i')$ for some integer values of $i$ and $i'$ in the interval $[L, U]$.

Plot $f(i) = g(i')$.
A geometric interpretation

Suppose we're looking at the subscript $2i-1$ and $i+3$. 
A geometric interpretation

Suppose we're looking at the subscript $2i-1$ and $i+3$.

We'll plot the line defined by

$$2i - 1 = i' + 3$$

or

$$2i - 4 = i'$$
A geometric interpretation

Suppose we're looking at the subscript $2i - 1$ and $i + 3$.

We'll plot the line defined by

\[ 2i - 1 = i' + 3 \]

or

\[ 2i - 4 = i' \]

Then we ask:
- Does the line hit any intersections on the graph paper (places where $i$ and $i'$ are both integers)?
A geometric interpretation

Suppose we're looking at the subscript $2i-1$ and $i+3$.

We'll plot the line defined by

$$2i - 1 = i' + 3$$

or

$$2i - 4 = i'$$

Then we ask:
- Does the line hit any intersections on the graph paper (places where $i$ and $i'$ are both integers)?
- Are any of those points inside the U-L bounding box?
A geometric interpretation

Suppose we're looking at the subscript $2i - 1$ and $i + 3$.

We'll plot the line defined by

$$2i - 1 = i' + 3$$

or

$$2i - 4 = i'$$

Then we ask:

- Does the line hit any intersections on the graph paper (places where $i$ and $i'$ are both integers)?
- Are those points inside the U-L bounding box?
- What's the direction vector?
Strong SIV test

Strong SIV subscripts are have the form

\[ a^i + c_1 \text{ and } a^i + c_2 \]

where \( a, c_1, \) and \( c_2 \) are loop invariant (and possibly constant).

For example,

\[ i+1 \text{ and } i \]

\[ 4i+2 \text{ and } 4i+4 \]

An important test.
Very common and the only case where we can compute a distance.
Strong SIV test

If there's a dependence,

\[ a^*i + c1 = a^*i' + c2 \]

so the distance

\[ d = i' - i = (c1 - c2)/a \]

But there's no dependence if

- \( a \) doesn't divide \( c1 - c2 \), or
- \( |d| > U - L \)
Weak SIV tests

Weak SIV subscripts are less restricted. They're of the form

\[ a*i + c1 \text{ and } b*i + c2 \]

For example,
\[ i+1 \text{ and } 5 \]
\[ 2*i+1 \text{ and } i+5 \]
\[ 2*i+1 \text{ and } -2*i \]

Wolfe gives an approach to finding direction vectors in the general case; but it's worth paying additional attention to certain special cases. (The special tests are faster and can sometimes work in the presence of symbolics)

See https://sites.google.com/site/parallelizationforllvm/weak-siv-test for my version of Wolfe's exact test.
Weak-zero SIV test

The special case of Weak SIV, where one of the coefficients of the index is zero.

For example,

\[ 2i + 1 \text{ and } 3 \]

If there's a dependence,

\[ ai + c_1 = c_2 \]

therefore,

\[ i = \frac{c_2 - c_1}{a} \]

The test requires simply that \( i \) be an integer and within the loop bounds.
Loop peeling

Here's a case that comes up sometimes

```c
for (i = 0; i < N; i++) {
1)   t = A[0];
2)   A[i] = t + c;
}
```

There's a loop-carried flow dependence from (2) to (1) with distance vector <= and a loop-independent anti dependence from (1) to (2). We can find both with the Weak-zero SV test.

Furthermore, we can peel the first iteration and delete the loop-carried dependence, like this

```c
A[0] = A[0] + c;
for (i = 1, i < N; i++) {
   t = A[0];
   A[i] = t;
}
```
Weak-crossing SIV test

A special case of Weak SIV where the coefficients of the index are equal in magnitude but opposite in sign. For example,

\[ 2i + 1 \text{ and } 3 - 2i \]

Generally, it's \( a*i + c1 \) and \( c2 - a*i \)

If \( a \) doesn't divide \( c2 - c1 \), there's no dependence.
Otherwise, solve

\[ i = (c2 - c1) / (2a) \]

• If \( i \) is not in the interval \([L, U]\), then there's no dependence.
• If \( i \) is an integer, the direction is *
• If \( i \) is not an integer, the direction is <> (that is, there's no loop-independent dependence)
Loop splitting

This case comes up occasionally

```
   do i = 1, n
       A(i) = A(n-i+1) + c
   enddo
```

The loop-carried dependence prevents parallelization; but (!), we can *split* the loop at the crossing iteration, thusly

```
   do i = 1, (n+1)/2
       A(i) = A(n-i+1) + c
   enddo
   do i = (n+1)/2+1, n
       A(i) = A(n-i+1) + c
   enddo
```

Breaks the dependence, allowing us to parallelize both loops.
MIV tests

If we have a subscript with multiple induction variables, things get much more interesting.

Note that the special case

\[ c_1 + a_1 \cdot i + a_2 \cdot j \]

and \( c_2 \) can be handled with Wolfe's exact test.

Otherwise, we use a combination of the GCD test backed up by Banerjee's Inequalities.

Both conservative tests, but in practice, they work well together.
GCD test

Given an MIV subscript

\[ c_0 + c_1i_1 + c_2i_2 + \ldots + c_Ni_N \quad \text{and} \quad d_0 + d_1i_1 + d_2i_2 + \ldots + d_Bi_N \]

we begin by creating an equation (since they must be equal for a dependence to exist)

\[ c_0 + c_1i_1 + \ldots + c_Ni_N = d_0 + d_1i_1' + \ldots + d_Ni_N' \]

and reorganizing

\[ c_0 - d_0 = d_1i_1' + \ldots + d_Ni_N' - c_1i_1 - \ldots - c_Ni_N \]
GCD test

Given the linear diophantine equation

\[ c_0 - d_0 = d_1 i_1' + \ldots + d_N i_N' - c_1 i_1 - \ldots - c_N i_N \]

we can make use of an ancient result that says:

If the gcd of \( d_1, d_2, \ldots, d_N, c_1, c_2, \ldots, c_N \) doesn't divide \( c_0 - d_0 \), then the equation has no solution in the integers.

Consider an example,

\[ 2i + 4j + 6 = 4i' - 10j' + 5 \]

\[ 1 = 4i' - 10j' - 2i - 4j \]

The gcd\((4, 10, -2, -4) = 2\), and since 2 doesn't divide 1, there can be no integer values of \( i, j, i', \) and \( j' \) that satisfy the equation. So no dependence.
GCD test

We can even use the GCD test to help with non-linear subscripts. Consider


Forming an equation, we see that

$1 = 2*B[i'] - 2*i*i$

Focusing on the constants, we know that the gcd(2, 2) = 2 and that 2 doesn't divide 1, so there can be no dependence.

The GCD test is simple and inexpensive, but not as useful as we might hope, since the gcd of several values tends to be 1.
Banerjee's inequality

Aka, the Extreme Value Test

If we can't disprove an MIV dependence using the GCD test, we try Banerjee's Inequality.

It's truly marvelous, but too big to fit into my remaining time.

The short version:
We examine the maximum and minimum possible values for each subscript achieved by substituting the upper and lower bounds for each index variable.

Coupled subscripts

Until now, we've been focusing on individual subscripts; involved finding solutions for a single linear Diophantine equation.

When we're faced with coupled subscripts, we have to find solutions for systems of Diophantine equations.

The paper talks about the Delta test.

I'll spend some time introducing a more powerful (and potentially much more expensive) approach called Fourier-Motzkin variable elimination.
Coupled subscripts

Consider this example

\[
\begin{align*}
&\text{do } i = 1, 10 \\
&\quad \text{do } j = 5, 20 \\
&\quad \quad A(i, 2\times j) = A(i+j, j-1) + C \\
&\quad \text{enddo} \\
&\text{enddo}
\end{align*}
\]

For there to be a loop-carried true dependence, we need

\[
\begin{align*}
&i = i' + j' \\
&2\times j = j' - 1 \\
&i, i' \geq 1 \\
&i, i' \leq 10 \\
&j, j' \geq 5 \\
&j, j' \leq 20
\end{align*}
\]
Systems of inequalities

We can rewrite these in a canonical form

\[
\begin{align*}
  i &= i' + j' \\
  2j &= j' - 1 \\
  i, i' &\geq 1 \\
  i, i' &\leq 10 \\
  j, j' &\geq 5 \\
  j, j' &\leq 20 \\
\end{align*}
\]

and test for integer feasibility.

If there's no feasible solution in the integers, then there's no dependence.

Yields a conservative solution.
Variable elimination

By combining pairs of inequalities where a particular index has coefficients with different signs, we can eliminate the index. For example

\[
\begin{align*}
i - i' - j' & \geq 0 \\
-1 + i - i' + j' & \geq 0 \\
1 + 2j - j' & \geq 0 \\
-1 - 2j + j' & \geq 0 \\
-1 + i & \geq 0 \\
-1 + i' & \geq 0 \\
-5 + j & \geq 0 \\
-5 + j' & \geq 0 \\
10 - i & \geq 0 \\
10 - i' & \geq 0 \\
20 - j & \geq 0 \\
20 - j' & \geq 0
\end{align*}
\]
Variable elimination

By combining pairs of inequalities where a particular index has coefficients with different signs, we can eliminate the index. For example

\[
\begin{align*}
  i - i' - j' & \geq 0 \\
  -i + i' + j' & \geq 0 \\
  1 + 2j - j' & \geq 0 \\
  -1 - 2j + j' & \geq 0 \\
  -1 + i & \geq 0 \\
  -1 + i' & \geq 0 \\
  -5 + j & \geq 0 \\
  -5 + j' & \geq 0 \\
  10 - i & \geq 0 \\
  10 - i' & \geq 0 \\
  20 - j & \geq 0 \\
  20 - j' & \geq 0
\end{align*}
\]
Variable elimination

By combining pairs of inequalities where a particular index has coefficients with different signs, we can eliminate the index. For example

\[
\begin{align*}
  &i - i' - j' \geq 0 \\
+ &-1 - 2j + j' \geq 0 \\
=> &-1 + i - i' - 2j \geq 0 \\
+ &-1 + i' \geq 0 \\
=> &-2 + i - 2j \geq 0 \\
+ &10 - i \geq 0 \\
=> &8 - 2j \geq 0
\end{align*}
\]
Variable elimination

By combining pairs of inequalities where a particular index has coefficients with different signs, we can eliminate the index. For example:

\[
\begin{align*}
  i - i' - j' & \geq 0 \\
  -i + i' + j' & \geq 0 \\
  1 + 2j - j' & \geq 0 \\
  -1 - 2j + j' & \geq 0 \\
  -1 + i & \geq 0 \\
  -1 + i' & \geq 0 \\
  -5 + j & \geq 0 \\
  -5 + j' & \geq 0 \\
  10 - i & \geq 0 \\
  10 - i' & \geq 0 \\
  20 - j & \geq 0 \\
  20 - j' & \geq 0 
\end{align*}
\]
Variable elimination

By combining pairs of inequalities where a particular index has coefficients with different signs, we can eliminate the index. For example

\[
\begin{align*}
  i - i' &\geq 0 \\
  -i + i' + j' &\geq 0 \\
  1 + 2j - j' &\geq 0 \\
  -1 - 2j + j' &\geq 0 \\
  -1 + i &\geq 0 \\
  -1 + i' &\geq 0 \\
  -5 + j &\geq 0 \\
  -5 + j' &\geq 0 \\
  10 - i &\geq 0 \\
  10 - i' &\geq 0 \\
  20 - j &\geq 0 \\
  20 - j' &\geq 0
\end{align*}
\]

Oops!

No feasible solution, so no dependence.
Variable elimination

Exploring all the combinations of inequalities is expensive, $O(2^n)$ in the number of inequalities, so this approach is a last resort.

For all that expense, it's still a conservative solution.

The Omega test builds on FM variable elimination to achieve a precise solution.

While it requires exponential time in the worst case, Pugh argues that it's polynomial time for common cases.

I'd use the various special cases first, possibly falling back on the exponential cases if the maximal precision is desired.
Conclusion

There's a ton of interesting work in this area. The paper was a nice summary of the state of the art, 20 years ago. Since then, the Omega test has closed off a lot of the effort and the polyhedral work has further distracted.

Non-linear subscripts remain difficult (but see the Range Test).

There's been some work to provide feedback to programmers about the results of dependence analysis, but how useful is it?