CS 526

Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526

DEPENDENCE ANALYSIS

The slides adapted from Vikram Adve and David Padua

Motivation: Vectorization

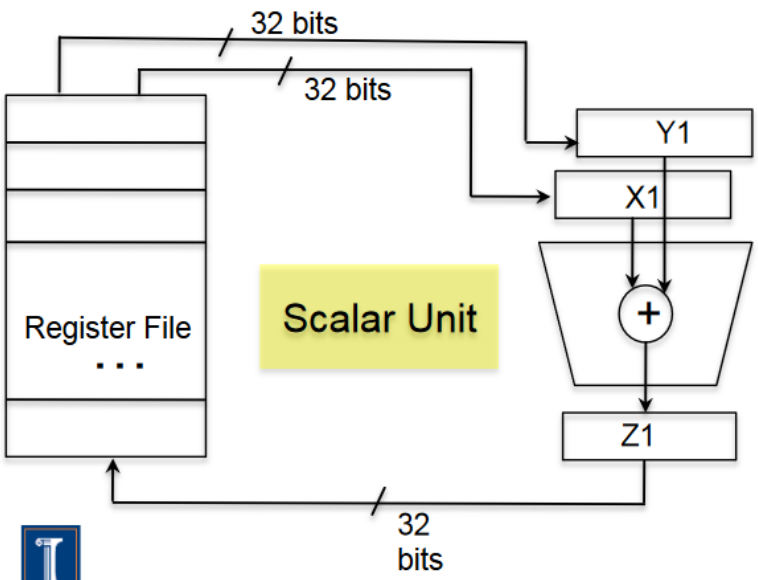
n times

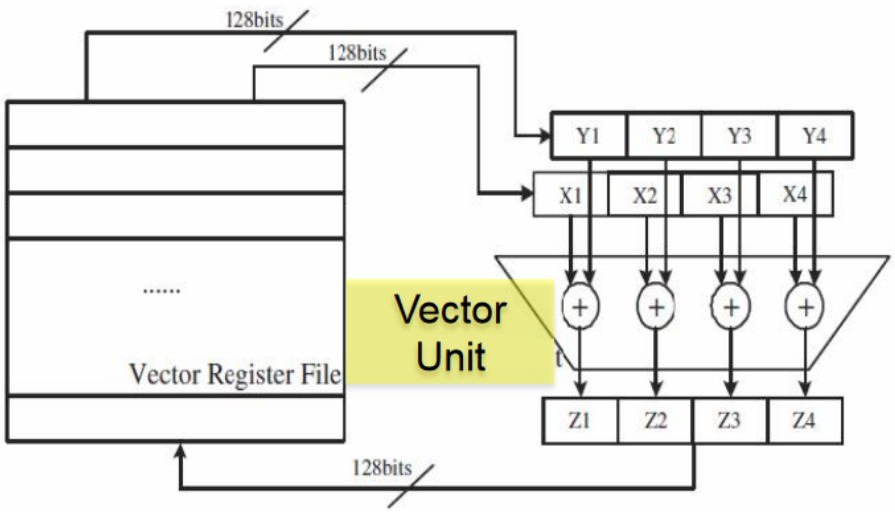
```
ld r1, addr1
ld r2, addr2
add r3, r1, r2
st r3, addr3
```

```
for (i=0; i<n; i++)
  c[i] = a[i] + b[i];
```

n/4 times

```
ldv vr1, addr1
ldv vr2, addr2
addv vr3, vr1, vr2
stv vr3, addr3
```



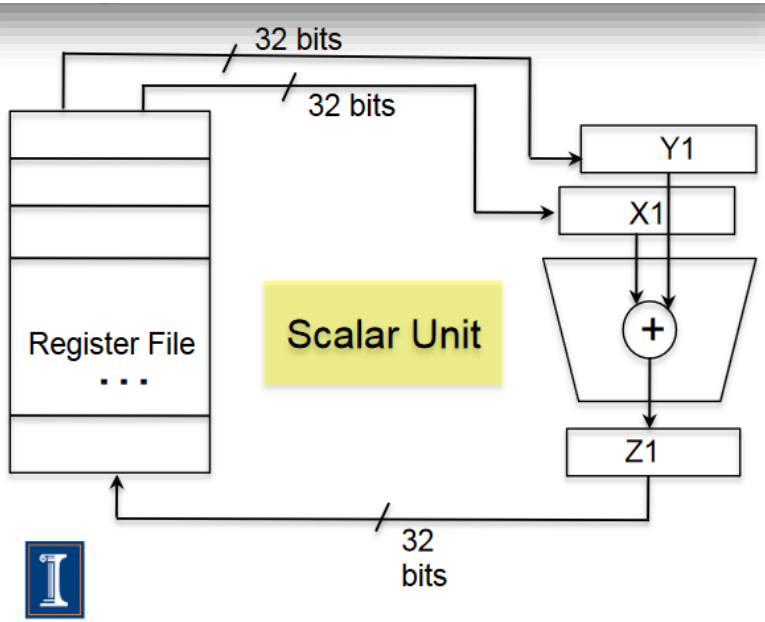


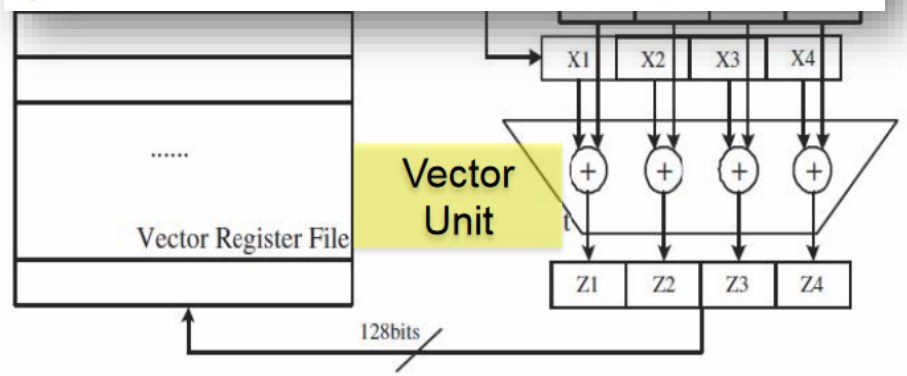
*Slide from Maria Garzaran and David Padua

Motivation: Vectorization

```
void vec_eltwise_product(vec_t* a, vec_t* b,
                         vec_t* c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```

```
void vec_eltwise_product_avx(vec_t* a, vec_t* b,
                             vec_t* c) {
    size_t i;
    __m256 va;
    __m256 vb;
    __m256 vc;
    for (i = 0; i < a->size; i += 8) {
        va = _mm256_loadu_ps(&a->data[i]);
        vb = _mm256_loadu_ps(&b->data[i]);
        vc = _mm256_mul_ps(va, vb);
        _mm256_storeu_ps(&c->data[i], vc);
    }
}
```





*Slide from Maria Garzaran and David Padua
** AVX code from Intel's Software&Services Group talk

Data Dependence

A **data dependence** from statement **S1** to statement **S2** exists if

1. there is a ***feasible execution path*** from S1 to S2, and
2. an instance of S1 ***references the same memory location*** as an instance of S2 in some execution of the program, and
3. at ***least one of the references is a store***.

Kinds of Data Dependence

Direct Dependence

X =…
…= X + …

Anti-dependence

… = X
X = …

Output Dependence

X = …
X = …

Dependence Graph

A **dependence graph** is a graph with:

- Each ***node represents a statement***, and
- Each ***directed edge*** from S1 to S2, if there is a **data dependence** between S1 and S2 (where the instance of S2 follows the instance of S1 in the relevant execution).
  - S1 is known as a ***source*** node
  - S2 is known as a ***sink*** node

Kinds of Data Dependence

Dependence Graph Edges

Direct Dependence

S1: X =…
S2: … = X + …

$$S_1 \longrightarrow S_2$$

Anti-dependence

S1: … = X
S2: X = …

$$S_1 \longrightarrow\!\!\!\!\!\!+\;\; S_2$$

Output Dependence

S1: X = …
S2: X = …

$$S_1 \longrightarrow\!\!\!\!\!\!\circ\;\; S_2$$

Dependence Graph for Loops

(***Repeat***) A **dependence graph** is a graph with:

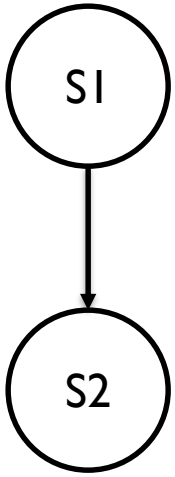
- one node per statement, and
- a directed edge from S1 to S2 if there is a data dependence between S1 and S2 (where the instance of S2 follows the instance of S1 in the relevant execution).

For loops: dependence graph is a ***summary of unrolled dependencies*** for different iterations

- Some (detailed) information may be lost

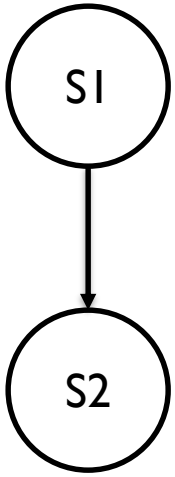
Dependence in Loops

```
        def X(), Y(), a(), i;
        do i = 1 to N
S1:        X(i) = a(i) + 2
S2:        Y(i) = X(i) + 1
        enddo
```



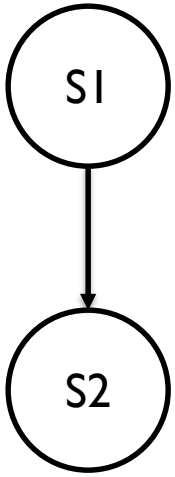
Dependence in Loops

```
        def X(), Y(), a(), i;
        do i = 1 to N
S1:        X(i+1) = a(i) + 2
S2:        Y(i) = X(i) + 1
        enddo
```



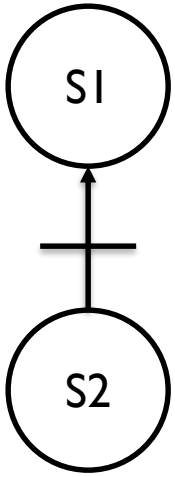
Dependence in Loops

```
        def X(), Y(), a(), i;
        do i = 2 to N
S1:        X(i) = a(i) + 2
S2:        Y(i) = X(i-1) + 1
        enddo
```



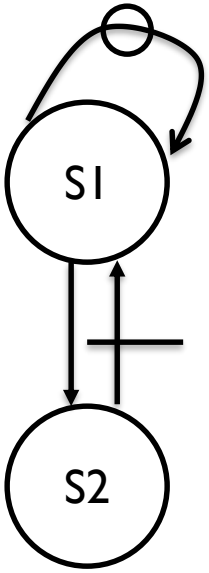
Dependence in Loops

```
        def X(), Y(), a(), i;
        do i = 1 to N
S1:         X(i) = a(i) + 2
S2:         Y(i) = X(i+1) + 1
        enddo
```



Dependence in Loops

```
        def X(), Y(), a(), t, i;
        do i = 1 to N
S1:         t = a(i) + 2
S2:         Y(i) = t + 1
        enddo
```



Reordering Transformation

Reordering Transformation: **merely changes the order** of execution of computations in a program, **without adding or deleting** executions of any computations.

Preserving Dependence: a reordering transformation preserves a dependence if it **preserves the relative execution order** of the ***source*** and ***sink*** statements of the dependence.

Reordering Transformation

Definition. Legal Transformation preserves the meaning of that program, i.e., **all externally visible outputs are identical to the original program**, and in identical order.

- We consider two programs equivalent (i.e., the transformation preserving the program meaning) if on the same inputs both the original and transformed programs, after being executed, produce the same outputs.

Theorem. A ***reordering*** transformation that preserves all data dependences in a program is a ***legal*** transformation.

Proof of Theorem 1

(by contradiction)

Loop-free program:

Let $S_1, \ldots S_n$ be the original execution order, and $i_1 \ldots i_n$ a permutation of the statement indices in the reordered program. If <u>we reorder code without violating dependencies</u>, but the output changed, then at least one statement would need to produce a different output. Since the statement is the same as in the original program, then its error must have propagated from the inputs. But in that case, there must have been a previous statement that violated (flow, anti, or output) dependence. Contradiction!

Loops:

The previous argument directly extends, by unrolling (and the index of the loop iteration represents the part of the permutation index).

Conditionals:

If there are conditional statements, the theorem must include ***<u>control</u> dependences*** in addition to data dependences.

(We will come back to this point next week)

Dependence in Loop Nests

Goal: Supporting transformations of a given loop nest (Assume perfect loop nest here)

Canonical Loop Nest: A loop nest is in canonical form if both lower bound and step of each loop are +1.

```
do i1 = 1 to n1
   do i2 = 1 to n2
      . . .
       do ik = 1 to nk
         statements
       enddo
      . . .
   enddo
enddo
```

Rectangular Loop Nest: The value of n1 to nk does not change during the execution

Dependence in Loop Nests

```
do i1 = 1 to n1
   do i2 = 1 to n2
      . . .
       do ik = 1 to nk
         statements
       enddo
      . . .
   enddo
enddo
```

Iteration space

The *iteration space* of the loop nest is a set of points in a k-dimensional integer space (i.e., a polyhedron):

$$L = \{[i_1, \ldots, i_n] : 1 \leq i_1 \leq n_1 \wedge \ldots \wedge 1 \leq i_k \leq n_k\}$$

Each element $[i_1, \ldots, i_n]$ is an **iteration vector**

Dependence in Loop Nests

Lexicographic Order: for iteration vectors $[i_1, \ldots, i_n]$ and $[j_1, \ldots, j_n]$:

$[i_1, \ldots, i_n] < [j_1, \ldots, j_n]$ iff there is a subscript k, such that $i_1 = j_1, \ldots i_{k-1} = j_{k-1}$ but $i_k < j_k$

If $\mathrm{I} = [i_1, \ldots, i_n] < [j_1, \ldots, j_n] = J$ we say that the iteration I preceds the iteration J

Dependence in Loop Nests

```
I=(v1',v2', …,vk')
J=(v1'',v2'',…, vk'')
```

```
do v1 = 1 to n1
   do v2 = 1 to n2
      . . .
       do vk = 1 to nk
          X(f1(I), …, fk(I)) = ...
          ... = X(g1(J), …, gk(J))
       enddo
      . . .
   enddo
enddo
```

Direct (Flow) Dependence in Loops

We say that S1→S2 (S1δS2) iff there exist $I, J \in L$ and $I \leq J$ where

1. There is a feasible path from instance I of statement S1 to instance J of statement S2,

```
S1:   X(f1(I), …, fk(I)) = ...
      ...
S2:   ... = X(g1(J), …, gk(J))
```

2. **$f_s(I) = g_s(J)$,** $\forall\ 1 \leq s \leq k$

   The statement S1 in iteration I writes and S2 in iteration J reads from the same memory location M

Antidependence in Loops

We say that S1$\not\leftarrow$S2 (S1δ^{-1}S2) iff there exist $I, J \in L$ and $I \leq J$:

1. There is a feasible path from instance I of statement S1 to instance J of statement S2,

```
S1:    ... = X(f1(I), …, fk(I))
       ...
S2:    X(g1(J), …, gk(J)) = ...
```

2. **$f_s(I) = g_s(J)$,** $\forall\ 1 \leq s \leq k$

   The statement S1 in iteration I reads and S2 in iteration J writes to the same memory location M

Output Dependence in Loops

We say that S1↬S2 (S1δ^0S2) iff there exist $I, J \in L$ and $I \leq J$:

1. There is a feasible path from instance I of statement S1 to instance J of statement S2,

```
S1: X(f1(I), …, fk(I)) = ...
    ...
S2: X(g1(J), …, gk(J)) = ...
```

2. **$f_s(I) = g_s(J)$,** $\forall\ 1 \leq s \leq k$

   The statement S1 in iteration I and S2 in iteration J both write to the same memory location M

Dependence Distance

Dependence Distance: If there is a dependence from statement S1 on iteration $\vec{i}$ and statement S2 on iteration $\vec{j}$ then the corresponding dependence distance vector is

$$d_{\vec{\boldsymbol{i}},\vec{\boldsymbol{j}}} = [j_1 - i_1, \ldots j_k - i_k]$$

Note: Computing distance vectors is harder than testing dependence

Dependence Distance

Direction Vector: For a distance vector of the form $d_{\vec{\boldsymbol{i}},\vec{\boldsymbol{j}}} = [\overrightarrow{j_1} - \overrightarrow{i_1}, \ldots \overrightarrow{j_k} - \overrightarrow{i_k}]$ the corresponding direction vector is $\delta_{\vec{\boldsymbol{i}},\vec{\boldsymbol{j}}} = [\delta_1, \ldots, \delta_k, \ldots \delta_m]$, where

$$\delta_k = \begin{cases} -, & if\ j_k - i_k < 0 \\ +, & if j_k - i_k > 0 \\ =, & if j_k - i_k = 0 \\ *, & if\ \text{sign} +, -, = \end{cases}$$

Note: **I** < **J** iff the leftmost non-'=' entry in δ(**I**, **J**) is '+'.

- We use the property of lexicographical ordering

Loop-Carried Dependence

Statement S2 has a loop carried dependence on statement S1 iff S1 references location M on iteration **I**, S2 references M on iteration **J** and **d(I,J)>0** .

```
do i = 1 to N
    A(i+1) = B(i)
    B(i+1) = A(i)
enddo
```

Level of loop-carried dependence is the leftmost non-"=" sign in the direction vector

- Forward dependence: S1 appears before S2 in the loop body
- Backward dependence: S2 appears before S1 in the loop body

Loop-Independent Dependence

Statement S2 has a loop independent dependence on statement S1 iff S1 references location M on iteration **I**, S2 references M on iteration **J** and **d(I,J)=0** .

```
do i = 1 to N
    A(i+1) = B(i)
    B(i+1) = A(i+1)
enddo
```

Determines the order in which the code is executed within the nest of loops (compare to loop carried dependence!)

Dependence in Loops

```
      do i = 2 to N
S1:      X(i-1) = X(i) + 1
      enddo
```

```
      do i = 1 to N
S1:      X(i+1) = X(i) + 1
      enddo
```

Dependence in Loops

```
      do j = 1 to 10
        do i = 1 to 100
S1:       X(i,j) = W(i,j) + 1
S2:       Y(i,j) = X(100-i,j)
      enddo
```

Dependence in Loops

```
    for i = 1 to N
      for j = 1 to M
        for k = 1 to 100
S1:         X(i,j,k+1) = X(i,j,k) + 1
        endfor
      endfor
    endfor
```

Direction Vector: =,=,+
Level: 3

Loop-Carried Dependence

Recall: Statement S2 has a loop carried dependence on statement S1 iff S1 references location M on iteration **I**, S2 references M on iteration **J** and **d(I,J)>0** .

So, in the direction vector for any dependence, the leftmost non-'=' entry must be '+' (if any non-'=' entry is present).

Equivalently: the distance vector d(I,J) ≥ 0.

Transformations and Direction Vectors

Theorem: Consider a transformation T on a loop nest that does not reorder statements within a loop body.

- Only changes how the program iterates the loops

Such a transformation is legal if, after applying the corresponding transformation to the direction vectors of each dependence, none of them have a leftmost non-'=' entry that is '-'. Equivalently, distance cannot be d<0.

- ***Equivalently:*** none of the dependences have had the order of their source and sink reversed.

Transformations and Direction Vectors

Theorem: Any transformation that does not change the order of loops, and does not reorder the iterations of the level-k loop preserves all level-k dependences in that loop.

Theorem: Any transformation that reorders the iterations of the level-k loop and makes no other changes is legal if the loop carries no dependences.

For discussion, see Allen and Kennedy book.

Dependence Testing

Dependence testing requires finding a solution to

$\{ f_s(\mathbf{I}) = g_s(\mathbf{J}), \forall 1 \leq s \leq n\}$

under the inequality constraints $\mathbf{I}, \mathbf{J} \in \mathrm{L}$

```
do i1 = L_1 to U_1
   do i2 = L_2 to U_2
      . . .
       do ik = L_k to U_k
         statements
       enddo
      . . .
   enddo
enddo
```

Complexity: undecidable in general

- Indirection arrays (e.g. X[Y[i]])
- Indirection arrays may only be known at runtime, without a specific application knowledge
- General alias analysis
- Non-linear subscript expressions

Dependence Testing

Assume linear subscript expressions, e.g., each f_s and g_s is

$$c_0 + c_1 i_1 + \ldots + c_n i_n,$$

where $i_1 \ldots i_n$ are loop index variables and c's are constants.

So we now have a system of equations

$$a_{10} + a_{11} i_1 + \ldots + a_{1n} i_n = b_{10} + b_{11} j_1 + \ldots + b_{1n} j_n$$

$$\ldots.$$

$$a_{k0} + a_{k1} i_1 + \ldots + a_{kn} i_n = b_{k0} + b_{k1} j_1 + \ldots + b_{kn} j_n$$

And for all I: $L_1 \leq i_1 \leq U_1 \ldots. L_n \leq i_n \leq U_n$ and same for J

Instance of integer programming

$\Rightarrow$ NP-complete in general

Simplifications

Two major simplifications in practice:

- Subscript expressions are usually simple: most often $\mathbf{i_k}$ or $\mathbf{a_1 i_k + a_0}$
- Be **conservative**:
  Check if a dependence **may exist**.

Simplifications

ZIV, SIV, MIV A subscript expression containing **z**ero, **s**ingle, or **m**ultiple index variable respectively:

E.g., A[3], A[2 * i1 + n], A[2 * i1 + 3 * i2 + 5]

Separable Subscripts : A subscript position is said to be **separable** if the index variables used in that subscript position are not used in any other subscript position.

E.g., A[i+1, j, k] and A[i, j, k]

Coupled Subscripts : Two subscript positions are said to be coupled if the same index variable is used in both positions.

E.g., A[i+1, i, k] and A[i, j+i, k]

Exact Solutions for SIV

A pair of subscripts with index variable i_j are **Strong SIV** if the subscript expressions are the form $a\ i_j + b_1$ and $a\ i_j + b_2$

- The loop iterates between one and n_j.

Dependence exists *iff* either of these hold:

1. $a = 0$ and $b_1 = b_2$, or
2. $|\ d_j\ | \leq n_j - 1$, where $d_j = (b1 - b2)/a$

Assumes: n_j , a, b_1, b_2 *are known*

Exact Solutions for Weak SIV

The set of subscripts with index variable i_j are **Weak SIV** if the subscripts are of the form $a_1\ i_j + b_1$ and $a_2\ i_j + b_2$

Each such subscript position j gives an equation of the form:

$$a_1\ i_j = a_2\ i_j + b_2 - b_1$$

Approach for each index variable i_j :

1. Solve up to r simultaneous equations in 2 unknowns.
2. Check if solutions satisfy 2 inequalities from the previous slide

Exact Solutions for Weak SIV

Special case: one of a_1 or a_2 is zero: **Weak-Zero SIV**
(solution is similar to strong SIV)

General problem: Find if $a_1 i_1 + a_0 = b_1 i_2 + b_0$

(Lemma) An extended GCD property:
For any pair of values (x, y), the Euclidian GCD algorithm can also compute a triplet (g, n_x, n_y) such that

$$g = n_x x + n_y y = \gcd(x, y)$$

Exact Solutions for Weak SIV

Theorem. Let (g, n_a, n_b) be such a triplet for pair $(a_1, -b_1)$.
Let x_k and y_k be given by:

$$
\begin{aligned}
x_k &= n_a \left(\frac{b_0 - a_0}{g} \right) + k \frac{b_1}{g} \\
y_k &= n_b \left(\frac{b_0 - a_0}{g} \right) + k \frac{a_1}{g}
\end{aligned}
$$

Then (x_k, y_k) is a solution of $a_1 i_1 + a_0 = b_1 i_2 + b_0$ for an integral value of k.
Furthermore, for any solution (x,y) there is a k such that $x = x_k$ and $y = y_k$

Solution strategy:

1. Compute x_0, y_0 using the above equations
2. Then find all values of k for which $x_0 + k\, b_1/g$ falls within loop bounds, and similarly for y_k.
3. For dependence to exist, the solution (x_k, y_k) must be within the region bounded by loop bounds

GCD Test

Simplifications

1. ignore loop bounds!
2. only test if a solution is *possible* (GCD property)
3. test each subscript position separately

GCD Property for Single Variable

Let $f(i) = a_1 i + a_0$ and $g(i) = b_1 i + b_0$

$f(i_1) = g(i_2) \Rightarrow a_1 i_1 + a_0 = b_1 i_2 + b_0$.

GCD Property: If there is a solution to the previous equation, then $g = \gcd(a_1, b_1)$ divides $a_0 - b_0$.

Proof: Let $a_1 = n_1 g$, $b_1 = m_1 g$. Then $g \times (n_1 i_1 - m_1 i_2) = a_0 - b_0$, and the term in parenthesis must be an integer.

GCD Test for Multiple Indices

Let $f(\mathbf{I}) = a_k i_k + \ldots + a_1 i_1 + a_0$ and
$g(\mathbf{I}) = b_k i_k + \ldots + b_1 i_1 + b_0$.

GCD Property: If there is a solution to the equation

$a_k i_{k1} + \ldots + a_0 = b_k i_{k2} + \ldots + b_0$, then

$g = \gcd(a_1, \ldots, a_k, b_1, \ldots, b_k)$ divides $(a_0 - b_0)$.

More tests: E.g., Banerjee test, Lamport test, Delta test…

Solving Complicated Indices

E.g. `A[x+2y-1, 2y, z, w+z, v, 1].`

Simplify the problem by identifying common special cases:

1. Separate subscript positions into coupled groups
2. Label each subscript as ZIV, SIV, or MIV
3. For each separable subscript, apply appropriate test (ZIV, SIV, or MIV). Yields direction vectors.
4. For each coupled group, apply a coupled subscript test; e.g., GCD test or Delta test or …
5. *If no test yields independence, a dependence exists*:
6. Concatenate direction vectors from different groups

6. [10 points]: We studied several tests for independence (ZIV, SIV, MIV, GCD). Which test would you use to test for a possible dependence in the following loops? Apply the test of your choice and report if there are dependences. Assume that the array boundaries are correct. (Use the space to the right for work).

- ```
  for (i=0; i<100; i++)
    for (j=0; j<100; i++)
      b[i]=b[i-1]+a[j];
  ```

- ```
  for (i=0; i<n; i++)
    for (j=0; j<n; i++)
       b[3*i-2] = b[2*i+5]+a[j];
  ```

- ```
  for (i=0; i<n; i++)
    for (j=0; j<n; i++)
       b[6*i+2*j+2] = b[2*i+4*j+4]+a[i];
  ```
  ```

Control-Flow Analysis

Consider now a program with conditionals:

```
for j = 1 to n {
  A[j] = A[j] * C[j]   // S1
  if (A[j] > k)
     B[j] = B[j] + D[j]   // S2
  else
     B[j] = B[j] – 1.0f
}
```

Control flow dependency exists between S1 and S2
(B[j] will be assigned the value only if A[j] has some value)

Control-Flow Analysis

We can convert the control dependency into a data dependency. Key steps:

- Consider ***guarded statements*** (if (bool_var) Stmt) and
- Transform the program to ***extract*** complicated expressions from the conditionals

```
for j = 1 to n {
  A[j] = A[j] * C[j]   // S1
  m = A[j] > k
  if (m) B[j] = B[j] + D[j]
  if (!m) B[j] = B[j] – 1.0f
}
```

Control-Flow Analysis (Forward)

```
for j = 1 to n {
  A[j] = A[j] * C[j]   // S1
  m = A[j] > k
  if (m) B[j] = B[j] + D[j]
  if (!m) B[j] = B[j] – 1.0f
}
```

The transformed program preserves all dependencies

This code can be readily vectorized:

- Compute the mask vector m[1…n]
- Compute the then branch result by filtering on m
- Compute the else branch result by filtering on m

E.g., SSE has operations that admit the mask.

Control-Flow Analysis (Exit)

```
for j = 1 to n {
  A[j] = A[j] * C[j]
  if (A[j] > k) break;
  B[j] = B[j] + D[j]
}
```

This is harder to transform with guarded form:

- If the condition is true once, exiting the loop is the same as if it fuly executed
- The condition depends on all iterations so far.
- Sketch of a solution. What is missing?

```
for j = 1 to n {
  if (m) break;
  A[j] = A[j] * C[j]
  m = m || A[j] > k
  if (m) break; // ?
  B[j] = B[j] + D[j]

}
```

```
for j = 1 to n {
  m1 = m2
  if (!m1) A[j] = A[j] * C[j]
  if (!m1) m2 = m2 || A[j] > k
  if (!m2) B[j] = B[j] + D[j]
}
```

Control-Flow Analysis (Backward)

```
for j = 1 to n {
      if (A[j] < m) continue;
  S1: k = k + 1
      A[j] = B[k] + D[j]
      if (A[j] > k) goto S1;
}
```

Appears when there is an inner loop like structure

- Applying just the forward analysis would yield potentially wrong code when combined with forward analysis
- It is transformed in conjunction with the related forward branches
- Simple heuristic: identify all code affected by a backward branch untouched and treat as a black-box. However, inefficient; for a more powerful analysis see e.g., *Conversion of Control Dependence to Data Depemdence; J.R. Allen and Ken Kennedy; POPL 1983*