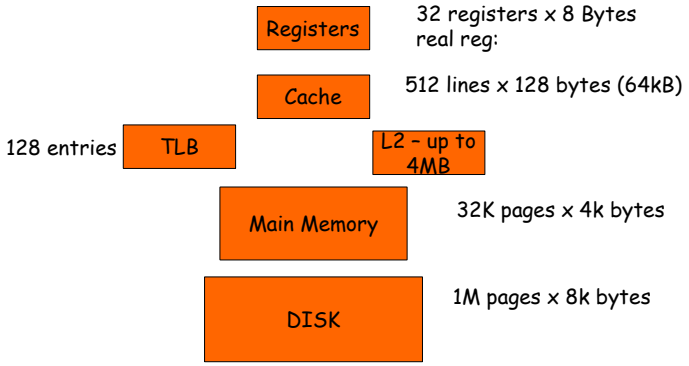
Dependence Analysis & Memory Hierarchy Optimizations

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

An Example Memory Hierarchy



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?

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 a 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

```
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];
```

- 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

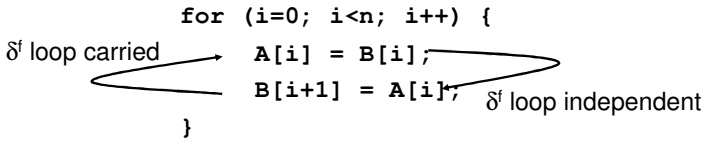
Data Dependence in Loops

- 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.

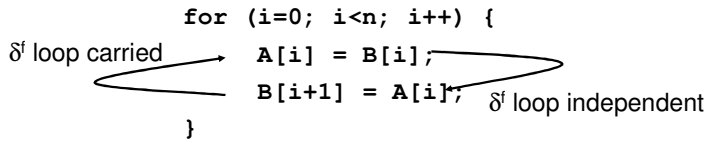
```
for (i=0; i<n; i++) {
      A[i] = B[i];
      B[i+1] = A[i];
}
```

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.



Unroll Loop to Find Dependencies




```
A[0] = B[0];   }
B[1] = A[0];   } i=0
A[1] = B[1];   }
B[2] = A[1];   } i=1
A[2] = B[2];   }
B[3] = A[2];   } i=2
   ⋮
```

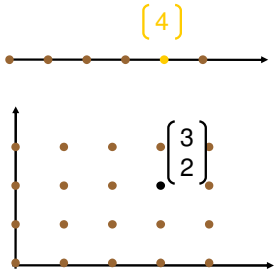
Distance/Direction of the dependence is also important.

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



Distance Vector

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

```
A[0] = B[0];   }
B[1] = A[0];   } i=0
A[1] = B[1];   }
B[2] = A[1];   } i=1
A[2] = B[2];   }
B[3] = A[2];   } i=2
   ⋮
```

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

$\mathbf{d} = \mathbf{I}_t - \mathbf{I}_s$

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

$\mathbf{I}_s + \mathbf{d} = \mathbf{I}_t$

Example of Distance Vectors

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

$A_{0,2}$= =$A_{0,2}$ $B_{0,3}$= =$B_{0,2}$ $C_{1,2}$= =$C_{0,3}$	$A_{1,2}$= =$A_{1,2}$ $B_{1,3}$= =$B_{1,2}$ $C_{2,2}$= =$C_{1,3}$	$A_{2,2}$= =$A_{2,2}$ $B_{2,3}$= =$B_{2,2}$ $C_{3,2}$= =$C_{2,3}$
$A_{0,1}$= =$A_{0,1}$ $B_{0,2}$= =$B_{0,1}$ $C_{1,1}$= =$C_{0,2}$	$A_{1,1}$= =$A_{1,1}$ $B_{1,2}$= =$B_{1,1}$ $C_{2,1}$= =$C_{1,2}$	$A_{2,1}$= =$A_{2,1}$ $B_{2,2}$= =$B_{2,1}$ $C_{3,1}$= =$C_{2,2}$
$A_{0,0}$= =$A_{0,0}$ $B_{0,1}$= =$B_{0,0}$ $C_{1,0}$= =$C_{0,1}$	$A_{1,0}$= =$A_{1,0}$ $B_{1,1}$= =$B_{1,0}$ $C_{2,0}$= =$C_{1,1}$	$A_{2,0}$= =$A_{2,0}$ $B_{2,1}$= =$B_{2,0}$ $C_{3,0}$= =$C_{2,1}$

j

i

Example of Distance Vectors

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

$A_{0,2}$= =$A_{0,2}$ $B_{0,3}$= =$B_{0,2}$ $C_{1,2}$= =$C_{0,3}$	$A_{1,2}$= =$A_{1,2}$ $B_{1,3}$= =$B_{1,2}$ $C_{2,2}$= =$C_{1,3}$	$A_{2,2}$= =$A_{2,2}$ $B_{2,3}$= =$B_{2,2}$ $C_{3,2}$= =$C_{2,3}$
$A_{0,1}$= =$A_{0,1}$ $B_{0,2}$= =$B_{0,1}$ $C_{1,1}$= =$C_{0,2}$	$A_{1,1}$= =$A_{1,1}$ $B_{1,2}$= =$B_{1,1}$ $C_{2,1}$= =$C_{1,2}$	$A_{2,1}$= =$A_{2,1}$ $B_{2,2}$= =$B_{2,1}$ $C_{3,1}$= =$C_{2,2}$
$A_{0,0}$= =$A_{0,0}$ $B_{0,1}$= =$B_{0,0}$ $C_{1,0}$= =$C_{0,1}$	$A_{1,0}$= =$A_{1,0}$ $B_{1,1}$= =$B_{1,0}$ $C_{2,0}$= =$C_{1,1}$	$A_{2,0}$= =$A_{2,0}$ $B_{2,1}$= =$B_{2,0}$ $C_{3,0}$= =$C_{2,1}$

j

i

A yields: $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ B yields: $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ C yields: $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$

Direction Vectors

Less exact than distance vectors

- can't analyze exactly, or
- summary of multiple distance vectors

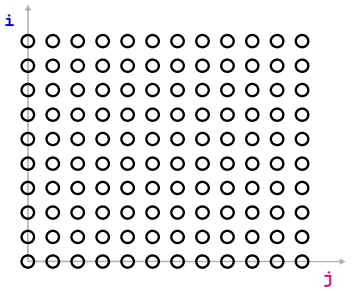
[0,0]	[1,inf]	[-inf,-1]	[-inf,inf]
=	+	-	+/-
=	<	>	*

Example:

- {<1,-1>, <1,0>, <1,1>} => <1,*>

Handy Representation: "Iteration Space"

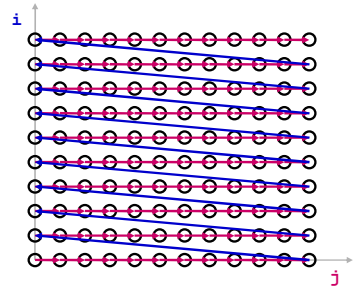
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



- each position represents an iteration

Visitation Order in Iteration Space

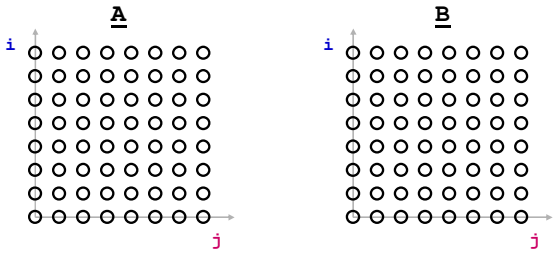
```
for i = 0 to N-1
   for j = 0 to N-1
     A[i][j] = B[j][i];
```



- Note: iteration space is not data space

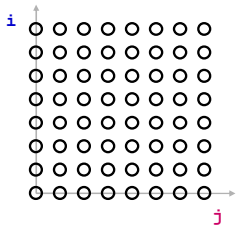
When Do Cache Misses Occur?

```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] = 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 the Cache Behavior of Array Accesses

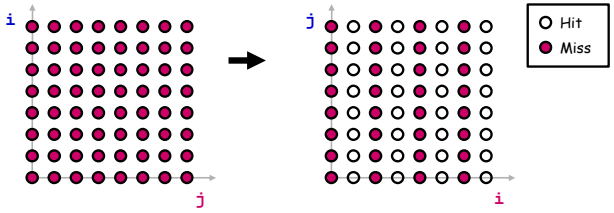
- We need to answer the following questions:
  - when do cache misses occur?
    - use "locality analysis"
  - can we change the order of the iterations (or possibly data layout) to produce better behavior?
    - evaluate the cost of various alternatives
  - does the new ordering/layout still produce correct results?
    - use "dependence analysis"

Examples of Loop Transformations

- Loop Interchange
- Cache Blocking
- Skewing
- Loop Reversal
- …

Loop Interchange

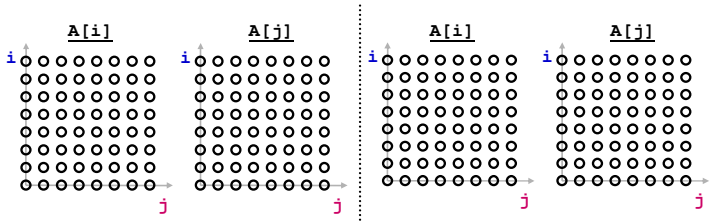
```
for i = 0 to N-1                for j = 0 to N-1
    for j = 0 to N-1                for i = 0 to N-1
        A[j][i] = i*j;                  A[j][i] = i*j;
```



- *(assuming N is large relative to cache size)*

Cache Blocking (aka "Tiling")

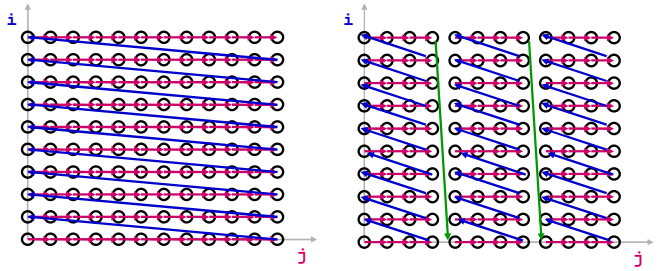
```
for i = 0 to N-1                for JJ = 0 to N-1 by B
    for j = 0 to N-1                for i = 0 to N-1
        f(A[i],A[j]);                   for j = JJ to max(N-1,JJ+B-1)
                                            f(A[i],A[j]);
```



now we can exploit temporal locality

Impact on Visitation Order in Iteration Space

```
for i = 0 to N-1                for JJ = 0 to N-1 by B
    for j = 0 to N-1                for i = 0 to N-1
        f(A[i],A[j]);                   for j = JJ to max(N-1,JJ+B-1)
                                            f(A[i],A[j]);
```



Cache Blocking 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];
```

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

Predicting Cache Behavior through "Locality Analysis"

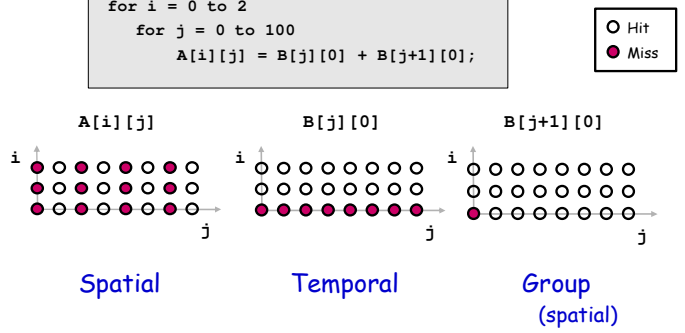
- Definitions:
  - Reuse:
    - accessing a location that has been accessed in the past
  - Locality:
    - accessing a location that is now found in the cache
- Key Insights
  - Locality only occurs when there is reuse!
  - BUT, reuse does not necessarily result in locality.
    - why not?

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 ➡ locality

Types of Data Reuse/Locality

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



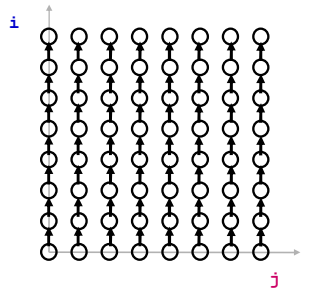
But...is the transform legal?

- 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?

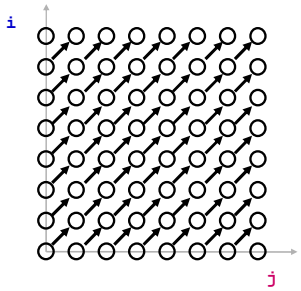
```
for i = 0 to N-1
    for j = 0 to N-1
      A[i+1][j] += A[i][j];
```



But...is the transform legal?

- Loop reversal ok?
- Loop interchange ok?

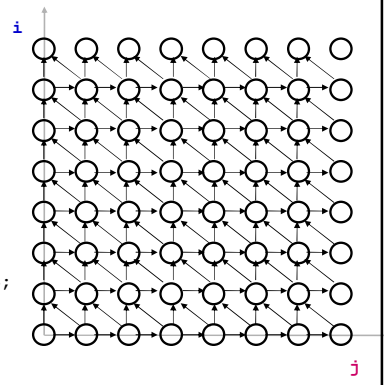
```
for i = 0 to N-1
    for j = 0 to N-1
      A[i+1][j+1] += A[i][j];
```



But...is the transform legal?

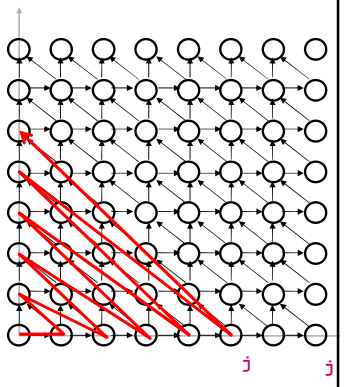
- What other visit order is legal here?

```
for i = 0 to TS
  for j = 0 to N-2
    A[j+1] =
     (A[j] + A[j+1] + A[j+2])/3;
```



But...is the transform legal?

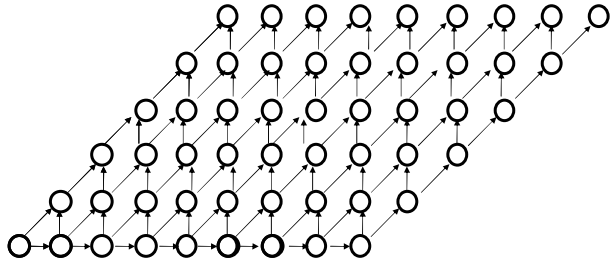
- What other visit order is legal here?




```
for i = 0 to TS
   for j = 0 to N-2
     A[j+1] =
       (A[j] + A[j+1] + A[j+2])/3;
```

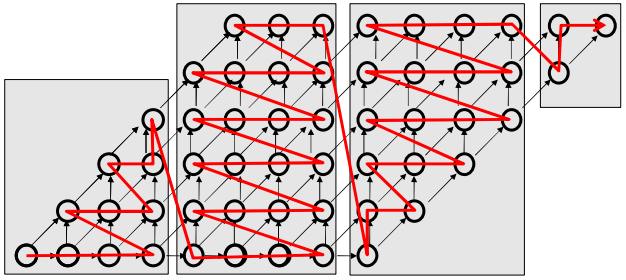

But...is the transform legal?

- Skewing...



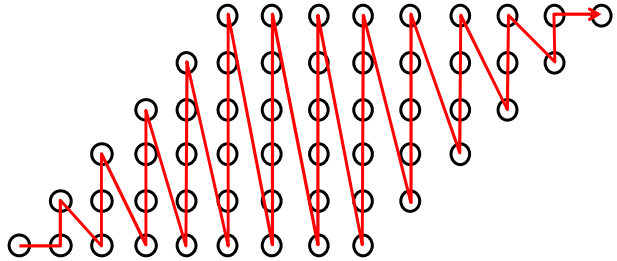
But...is the transform legal?

- 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.

Reversal $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ Interchange $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ Skew $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$

"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, ……

Garpcc 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

Garpcc 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

```
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[][] with scalar in inner loop.
- Reduces memory references by $2(N^3-N^2)$

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


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 t_0 to $t_{p(e)}$
- In body of loop:
  - Replace A[i] with t_0
  - Replace A[i+j] with t_j
- At end of innermost loop body add assignments $t_{p(e)} = t_{p(e)-1}$; ... ; t_1 <- t_0
- Init temps by peeling off p(e) iterations

Example: MM

```
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];
```

p=<0,1>

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

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


Scalar Replacement: Loop Body

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

p=<1,0>

p=<0,1>

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

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


Scalar Replacement: Init

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

2) after replacement

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

3) If we aren't sure of trip count

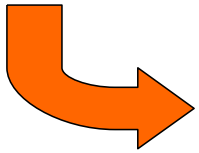
```
if (n>=0) {
    t0 = b[0]
    t1 = t0 + f
    b[1] = t1
    a[0] = 2 * t0 +
c[0]
}
```

1) Peel of p(e) iterations of loop

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

Finished

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



```
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
}
```