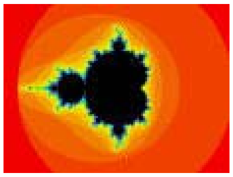
Fractal Symbolic Analysis



Keshav Pingali

Joint work with
Vijay Menon, Nikolai Mateev
Cornell University

Context

- Restructuring compilers
- Program Transformations
  - Legality of Transformation
    - Must preserve semantics of original program
  - Generation of Transformation
    - Enhance temporal/spatial locality
    - Increase parallelism
- Focus of talk: legality

Legality of transformations

- Standard approach: dependence analysis
  - Sufficient but not necessary condition for legality
  - Not powerful enough to handle LU + pivoting
- Powerful approach: symbolic analysis
  - Intractable for most programs
- Our approach: fractal symbolic analysis
  - Combines power with tractability
  - Solves problem with restructuring LU + pivoting !

Overview of Talk

- Background on Legality
  - Dependence Analysis
  - Symbolic Execution
  - Two running examples
- Fractal Symbolic Analysis
- Automatic Blocking of LU with pivoting
- Summary and Open Issues

Dependence Analysis

- Considers only memory locations touched by statements
  - Independent operations may be reordered

```
S1;    <=>    S2;
S2            S1
```

  - S1 does not read/write location written by S2 and vice versa
- Extends to loop transformations
  - interchange
  - distribution
  - tiling

Symbolic Analysis

- Dependence Analysis is inexact:

```
s1: a = b         ?      s3: a = 2*a
s2: b = 2*b      <=>     s2: b = 2*b
s3: a = 2*a              s1: a = b
```

- Symbolic execution shows equality:
  - $a_{out} = 2*b_{in}$
  - $b_{out} = 2*b_{in}$
- Powerful and general technique
- But, intractable for recurrent loops....

Example #1: Reduction

- Loop interchange for spatial locality

```
do i = 1,N                   do j = 1,N
  do j = 1,N        <=>        do i = 1,N
    k = k + A(i,j)               k = k + A(i,j)
```

  - dependence must be violated
  - unbounded recurrence complicates symbolic approach
- Possible solution:
  - pattern match reduction
  - disregard reduction dependences
- SGI MIPSpro performs loop interchange above

Reduction, Cont.

- Pattern matching is fragile. Consider:

```
do i = 1,N
  do j = 1,N
    t1 = k
    t2 = t1 + A(i,j)
    k = t2
```

- SGI MIPSpro fails to interchange above
- Symbolic execution?
  - body is equivalent to previous example
  - how to handle recurrent loop?

Example #2: Pivoting

- Loop distribution (assume $p(j) \geq j$)

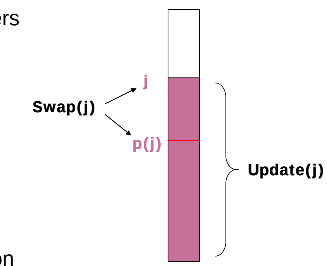
```
        for j = 1:n                                  for j = 1:n
B1(j):    tmp = a(j)                         B1(j):    tmp = a(j)
          a(j) = a(p(j))            ⇔                  a(j) = a(p(j))
          a(p(j)) = tmp                                a(p(j)) = tmp

B2(j):    for i = j+1:n                              for j = 1:n
            a(i) = a(i)/a(j)                 B2(j):    for i = j+1:n
                                                         a(i) = a(i)/a(j)
```

- Dependence analysis: too conservative
- Symbolic comparison: ???

Pivoting, Cont.

- Distribution reorders
  - swaps
  - updates



- Effect of distribution
  - Before: swaps & updates interleaved
  - After: all swaps followed by all updates

Overview of Talk

- Background on Legality
- Fractal Symbolic Analysis
  - High Level Algorithm
  - Examples
  - Guarded Symbolic Expressions
- Automatic Blocking of LU with pivoting
- Summary and Open Issues

High Level Algorithm

- Compare a program P_1 and its transformed version P_2 via comparison of simplified programs

P_1	=?=	P_2
↓	⇑	↓
P_1'	=?=	P_2'
↓	⇑	↓
:	:	:
↓	⇑	↓
P_1^n	=?=	P_2^n

- Equality of simpler programs ⇒ equality of complex programs
  - sufficient, but not necessary condition
- Simplify until symbolic execution is tractable
  - e.g., comparing basic blocks

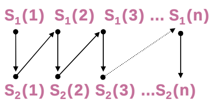
Example

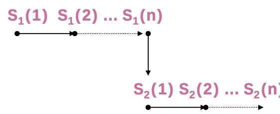
- Prove or disprove equivalence of:

```
for i = 1 : n
  S1(i);
  S2(i);
```

⟺

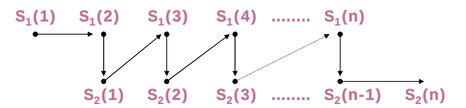
```
for i = 1 : n
  S1(i);
for i = 1 : n
  S2(i);
```

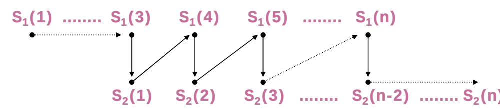




Inductive Approach

- Think of transformation as incremental process



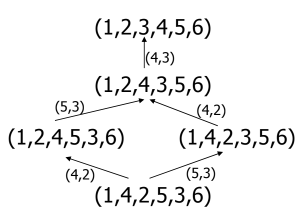


- If reordering at each step is legal, overall transformation is legal!

Theorem

1. Any permutation can be generated by sequences of adjacent transpositions.
2. The reordered pairs of a permutation generate such a sequence of adjacent transpositions.

(1,4,2,5,3,6) -> (1,2,3,4,5,6) reordered pairs: (4,2),(4,3),(5,3)



11/3/2010 15

Incremental proof of legality

- Loop distribution: show that
  - $\forall l,m.(1 \le m < l \le n)$
    S2(m); S1(l) = S1(l);S2(m)
- Similar conditions:
  - Statement reordering
  - Loop interchange
  - Loop reversal
  - Loop tiling
- Proving incremental steps may be easier than proving that entire transformation is correct.

Example #1: Loop Interchange

- Prove equivalence of:

```
        do i = 1,N                    do j = 1,N
         do j = 1,N         <=>        do i = 1,N
          t1 = k                        t1 = k
S(i,j):   t2 = t1 + A(i,j)    S(i,j):   t2 = t1 + A(i,j)
          k = t2                        k = t2
```

- Sufficient condition:
  - Prove
    - $S(i_1,j_1); S(i_2,j_2) = S(i_2,j_2); S(i_1,j_1)$
    - where $i_1 < i_2$ and $j_1 > j_2$

Simplified Test

- Prove equivalence:

t1 = k t2 = t1 + $A(i_1,j_1)$ k = t2 t1 = k t2 = t1 + $A(i_2,j_2)$ k = t2	⟺	t1 = k t2 = t1 + $A(i_2,j_2)$ k = t2 t1 = k t2 = t1 + $A(i_1,j_1)$ k = t2

Use symbolic analysis:

$k_{out} = k_{in} + A(i_1,j_1) + A(i_2,j_2)$ $\quad$ $k_{out} = k_{in} + A(i_2,j_2) + A(i_1,j_1)$

- Proves legality of interchange
  - Sufficient, but not necessary condition

Example #2: Loop Distribution

- Given $p(j) \geq j$, prove:

```
       for j = 1:n                      for j = 1:n
        tmp = a(j)                       tmp = a(j)
B1(j):  a(j) = a(p(j))           B1(j):  a(j) = a(p(j))
        a(p(j)) = tmp                    a(p(j)) = tmp
                          <=>
                                         for j = 1:n
B2(j):  for i = j+1:n            B2(j):  for i = j+1:n
         a(i) = a(i)/a(j)                 a(i) = a(i)/a(j)
```

- Dependence analysis: too conservative
- Symbolic comparison: intractable

Simplified Test

- Given $p(l) \geq l \wedge l > m$ prove:

```
B2(m):  for i = m+1:n            B1(l):  tmp = a(l)
         a(i) = a(i)/a(m)                a(l) = a(p(l))
                                         a(p(l)) = tmp
                          <=>
        tmp = a(l)
B1(l):  a(l) = a(p(l))           B2(m):  for i = m+1:n
        a(p(l)) = tmp                     a(i) = a(i)/a(m)
```

- Further simplification?

Another Step

- Given $p(l) \geq l \wedge l > m \wedge i > m$ prove:

```
a(i) = a(i)/a(m)              tmp = a(l)
                              a(l) = a(p(l))
tmp = a(l)           ⇔        a(p(l)) = tmp
a(l) = a(p(l))
a(p(l)) = tmp                 a(i) = a(i)/a(m)
```

- Programs not equivalent
- Over-simplification!

Observation

- Underlying symbolic technique is important
- More powerful symbolic analysis
  - less simplification
  - more accurate test

Our Symbolic Analyzer

- Restriction
  - non-recurrent loops (no dependences)
  - affine indices/loop bounds (finite array regions)
- Restricted Programs
  - Easily identified
  - Can symbolically summarize effect

Conditional Symbolic Expressions

- Summarize effect of statement on data
- For single statement:

  a(p(l)) = tmp

  - $a_{out}(k) = \begin{cases} (k = p(l)) \Rightarrow tmp_{in} \\ \text{else} \Rightarrow a_{in}(k) \end{cases}$
- For multiple statements, recurse

Back to Example #2

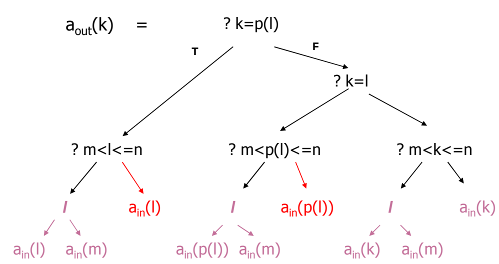
- 1 level of simplification meets restriction:

```
B2(m):  for i = m+1:n              B1(l):  tmp = a(l)
          a(i) = a(i)/a(m)                 a(l) = a(p(l))
                                           a(p(l)) = tmp
                            ⟺
B1(l):  tmp = a(l)                 B2(m):  for i = m+1:n
        a(l) = a(p(l))                       a(i) = a(i)/a(m)
        a(p(l)) = tmp
```

where $p(l) \geq l \wedge l > m$

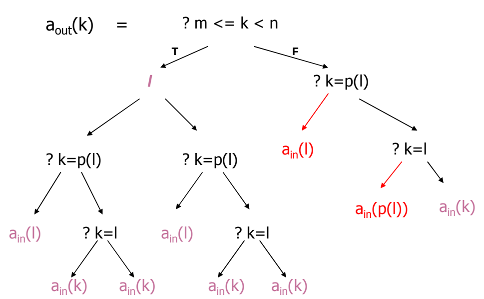
- No further simplification necessary

Conditional Expression Tree#1

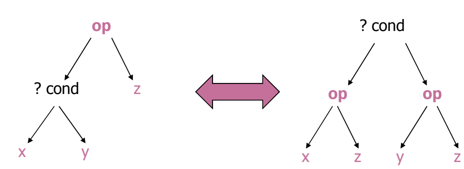


Expression live output variables in terms of input variables

Conditional Expression Tree#2



Normalization



Rotate conditions to top

Guarded Symbolic Expressions

- Convert to GSE:
  - $array_{out}(\vec{j}) = \begin{cases} guard_1(\vec{j}) \Rightarrow expr_1(\vec{j}) \\ \vdots \\ guard_n(\vec{j}) \Rightarrow expr_n(\vec{j}) \end{cases}$
- guards
  - affine contraints on
    - loop vars
    - symbolic constants
  - describe regions of array
- exprs
  - unconditional symbolic expressions
  - describe values in an array region

Comparing GSE's

- Comparison 2 GSE's:
  - Prove union of guards in each GSE are equivalent
    - GSE's must cover same regions
  - When guards of 2 GSE's intersect,
    - Prove corresponding expressions equivalent
- Tools:
  - Integer Programming (e.g., Omega Library)
  - Symbolic Math Engine (e.g., Maple)

Back to Example

- For both program blocks:
- $a_{out}(k) = \begin{cases} k \le m & \Rightarrow a_{in}(k) \\ k = l & \Rightarrow a_{in}(p(l))/a_{in}(m) \\ k = p(l) & \Rightarrow a_{in}(l)/a_{in}(m) \\ \text{else} & \Rightarrow a_{in}(k)/a_{in}(m) \end{cases}$
- Note:
  - 16 pair wise intersections / 4 non-empty
  - Expressions are syntactically identical
  - No floating point computation reordered!

Loop distribution is legal in our example

```
        for j = 1:n
B1(j):    tmp = a(j)
          a(j) = a(p(j))
          a(p(j)) = tmp
B2(j):    for i = j+1:n
            a(i) = a(i)/a(j)
```

⇔

```
        for j = 1:n
B1(j):    tmp = a(j)
          a(j) = a(p(j))
          a(p(j)) = tmp
        for j = 1:n
B2(j):    for i = j+1:n
            a(i) = a(i)/a(j)
```

Overview of Talk

- Background on Legality
- Fractal Symbolic Analysis
- Automatic Blocking of LU with pivoting
  - Blocking LU
  - Legality Issue
  - Application of FSA
- Summary and Open Issues

LU Factorization with Partial Pivoting

- Key algorithm for solving systems of linear equations:
  - To solve **A x = b** for x:
    - => Factor A into L U
      - L is lower triangular
      - U is upper triangular
    - => Solve **L y = b** for y
      - Forward substitution
    - => Solve **U x = y** for x
      - Backward substitution
- Note:
  - Partial pivoting required for stability
  - Data cache key to performance

Blocking LU without Pivoting

```
do j = 1, N
 do i = j+1, N
  A(i,j) /= A(j,j)
 do k = j+1, N
  do i = j+1, N
   A(i,k) -= A(i,j)*A(j,k)
```

- Must be blocked to exploit reuse in update
- Compiler transformations (Carr & Kennedy 1992)
  - strip-mining
  - index-set-splitting
  - loop distribution
  - tiling

LU with Partial Pivoting

```
do j = 1,N
```

Select pivot row:

```
  p(j) = j;
  do i = j+1,N
    if (A(i,j)>A(p(j),j))
      p(j) = i;
```

Swap pivot row with current:

```
  do k = 1,N
    tmp = A(j,k);
    A(j,k) = A(p(j),k);
    A(p(j),k) = tmp;
```

Scale column (to store L):

```
  do i = j+1,N
    A(i,j) = A(i,j)/A(j,j);
```

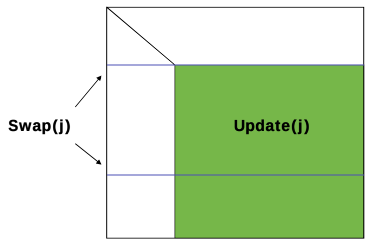
Update(to compute partial U):

```
  do k = j+1,N
    do i = j+1,N
      A(i,k) = A(i,k) - A(i,j)*A(j,k);
```

x	x	x	x	x	x
0	x	x	x	x	x
0	0	5	x	x	x
0	0	3	x	x	x
0	0	7	x	x	x
0	0	2	x	x	x

- Same opts. $\Rightarrow$ legal blocked code

Caveat: Proving Legality



- Blocking is legal, but
  - Reorders swaps and updates
  - Violates dependences

Legality: Loop Distribution

```
do jB = 1,N,B
  do j = jB,jB+B-1
    p(j) = j;
    do i = j+1,N
      if (A(i,j)>A(p(j),j))
        p(j) = i;

    do k = 1,N
      tmp = A(j,k);
      A(j,k) = A(p(j),k);
      A(p(j),k) = tmp;

    do i = j+1,N
      A(i,j) = A(i,j)/A(j,j);

    do k = j+1,jB+B-1
      do i = j+1,N
        A(i,k) = A(i,k) - A(i,j)*A(j,k);

    do k = jB+B,N
      do i = j+1,N
        A(i,k) = A(i,k) - A(i,j)*A(j,k);
```

⇔

```
do jB = 1,N,B

  do j = jB,jB+B-1
    p(j) = j;
    do i = j+1,N
      if (A(i,j)>A(p(j),j))
        p(j) = i;
    do k = 1,N
      tmp = A(j,k);
      A(j,k) = A(p(j),k);
      A(p(j),k) = tmp;
    do i = j+1,N
      A(i,j) = A(i,j)/A(j,j);
    do k = j+1,jB+B-1
      do i = j+1,N
        A(i,k) = A(i,k) - A(i,j)*A(j,k);

  do j = jB,jB+B-1
    do k = jB+B,N
      do i = j+1,N
        A(i,k) = A(i,k) - A(i,j)*A(j,k);
```

Dependent swaps and updates are reordered

Simplified Programs

```
B1(l):
  pl = l;
  do i = l+1,N
    if (A(i,l)>A(pl,l))
      pl = i;
  do k = 1,N
    tmp = A(l,k);
    A(l,k) = A(pl,k);
    A(pl,k) = tmp;
  do i = l+1,N
    A(i,l) = A(i,l)/A(l,l);
  do k = l+1,jB+B-1
    do i = l+1,N
      A(i,k) = A(i,k) - A(i,l)*A(l,k);
B2(m):
  do k = jB+B,N
    do i = m+1,N
      A(i,k) = A(i,k) - A(i,m)*A(m,k);
```

⇔

```
B2(m):
  do k = jB+B,N
    do i = m+1,N
      A(i,k) = A(i,k) - A(i,m)*A(m,k);
B1(l):
  pl = l;
  do i = l+1,N
    if (A(i,l)>A(pl,l))
      pl = i;
  do k = 1,N
    tmp = A(l,k);
    A(l,k) = A(pl,k);
    A(pl,k) = tmp;
  do i = l+1,N
    A(i,l) = A(i,l)/A(l,l);
  do k = l+1,jB+B-1
    do i = l+1,N
      A(i,k) = A(i,k) - A(i,l)*A(l,k);
```

- Prove equivalence where `jB ≤ m < l ≤ pl, jB+B-1`

Another step of simplification

```
S3(l):
    do k = 1,N
      tmp = A(l,k);
      A(l,k) = A(pl,k);
      A(pl,k) = tmp;

B2(m):
    do k = jB+B,N
      do i = m+1,N
        A(i,k) = A(i,k) - A(i,m)*A(m,k);
```

⇔

```
B2(m):
    do k = jB+B,N
      do i = m+1,N
        A(i,k) = A(i,k) - A(i,m)*A(m,k);

S3(l):
    do k = 1,N
      tmp = A(l,k);
      A(l,k) = A(pl,k);
      A(pl,k) = tmp;
```

- Prove equivalence where `jB ≤ m < l ≤ pl, jB+B-1`

Nonempty Intersecting Regions

```
# # Test whether A(l,y) - A(l,m) * A(m,y) = A(l,y) - A(l,m) * A(m,y)
{[pl,y]: 1 <= jB <= m < l <= jE < y <= N && l <= pl <= N}

# # Test whether A(pl,y) - A(pl,m) * A(m,y) = A(pl,y) - A(pl,m) * A(m,y)
{[l,y]: 1 <= jB <= m < l <= jE < y <= N && l < pl <= N}

# # Test whether A(x,y) - A(x,m) * A(m,y) = A(x,y) - A(x,m) * A(m,y)
{[x,y]: 1 <= jB <= m < x < l <= jE < y <= N && l <= pl <= N} union
 {[x,y]: 1 <= jB <= m < l <= jE < y <= N && l < x < pl <= N} union
 {[x,y]: 1 <= jB <= m < l <= pl < x <= N && l <= jE < y <= N}

# # Test whether A(l,y) = A(l,y)
{[pl,y]: 1 <= jB <= m < l <= pl,jE <= N && 1 <= y <= jE}

# # Test whether A(pl,y) = A(pl,y)
{[l,y]: 1 <= jB <= m < l < pl <= N && l, y <= jE <= N && 1 <= y}

# # Test whether A(x,y) = A(x,y)
{[x,y]: 1 <= jB <= m < l <= pl,jE <= N && 1 <= x < l && 1 <= y <= jE} union
 {[x,y]: 1 <= jB <= m < l < x < pl <= N && y, l <= jE <= N && 1 <= y} union
 {[x,y]: 1 <= jB <= m < l <= pl < x <= N && y, l <= jE <= N && 1 <= y} union
 {[x,y]: 1 <= jB <= m < l <= jE < y <= N && l <= pl <= N && 1 <= x <= m}
```

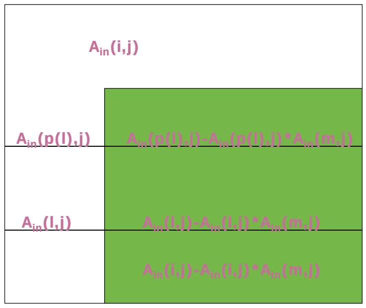
Note: no floating point computation reordered!

Conditional Tree Expressions

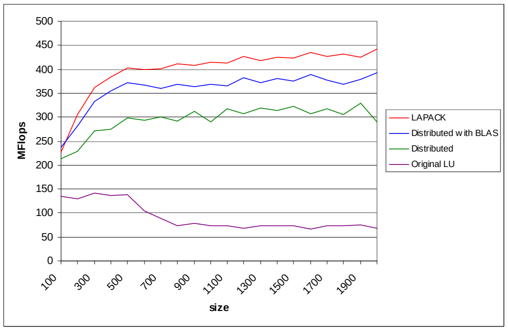
```
Region("[x,y]: 1<=jB<=m<l<=pl<=N && l<=jE<=N && 1<=x,y<=N",
 Cond("x=pl && (exists [k2]:1<=k2<=N && y=k2)",
  Cond("(exists [k4,i4]:jE+1<=k4<=N && m+1<=i4<=N && l=i4 && y=k4)",
   Op("-",
    Leaf("A(l,y)"),
     Op("*",
      Leaf("A(l,m)"),
      Leaf("A(m,y)"))),
     Leaf("A(l,y)")),
   Cond("x=l && (exists [k2]:1<=k2<=N && y=k2)",
    Cond("(exists [k4,i4]:jE+1<=k4<=N && m+1<=i4<=N && pl=i4 && y=k4)",
     Op("-",
      Leaf("A(pl,y)"),
      Op("*",
       Leaf("A(pl,m)"),
       Leaf("A(m,y)"))),
     Leaf("A(pl,y)")),
    Cond("(exists [k4,i4]:jE+1<=k4<=N && m+1<=i4<=N && x=i4 && y=k4)",
     Op("-",
      Leaf("A(x,y)"),
      Op("*",
       Leaf("A(x,m)"),
       Leaf("A(m,y)"))),
     Leaf("A(x,y)")))));;
```

Six Regions

- $A_{out}(i,j)$ =



LU Performance



- **300 MHz SGI Octane with 2MB L2 Cache**

Summary of Fractal Symbolic Analysis

- Tractable approach to using symbolic analysis to prove legality of program transformations
- Enables tradeoff between
  - tractability (dependence analysis)
  - accuracy (symbolic comparison)
- Encapsulates symbolic information a compiler is permitted to use
- Prototype implemented in OCAML
- Solves problem of restructuring LU + pivoting

Related Work

- Haghighat and Polychronopoulos(1996)
  - Symbolic analysis for induction variable recognition
- Fahringer and Scholz (1997)
  - Symbolic dependence testing
- Rinard (1997)
  - Commutativity analysis for parallelization

11/3/2010 46

Open Issues

- Synthesis of Transformations
  - e.g., dependence vectors ⇒ transformations
- Better underlying symbolic analysis
- Performance: how do we apply this to large programs?