Compiler-Based Autotuning Technology

Lecture 3: A Closer Look at Polyhedral Compiler Technology

Mary Hall

July, 2011

* This work has been partially sponsored by DOE SciDAC as part of the Performance Engineering Research Institute (PERI), DOE Office of Science, the National Science Foundation, DARPA and Intel Corporation.

THE UNIVERSITY OF UTAH

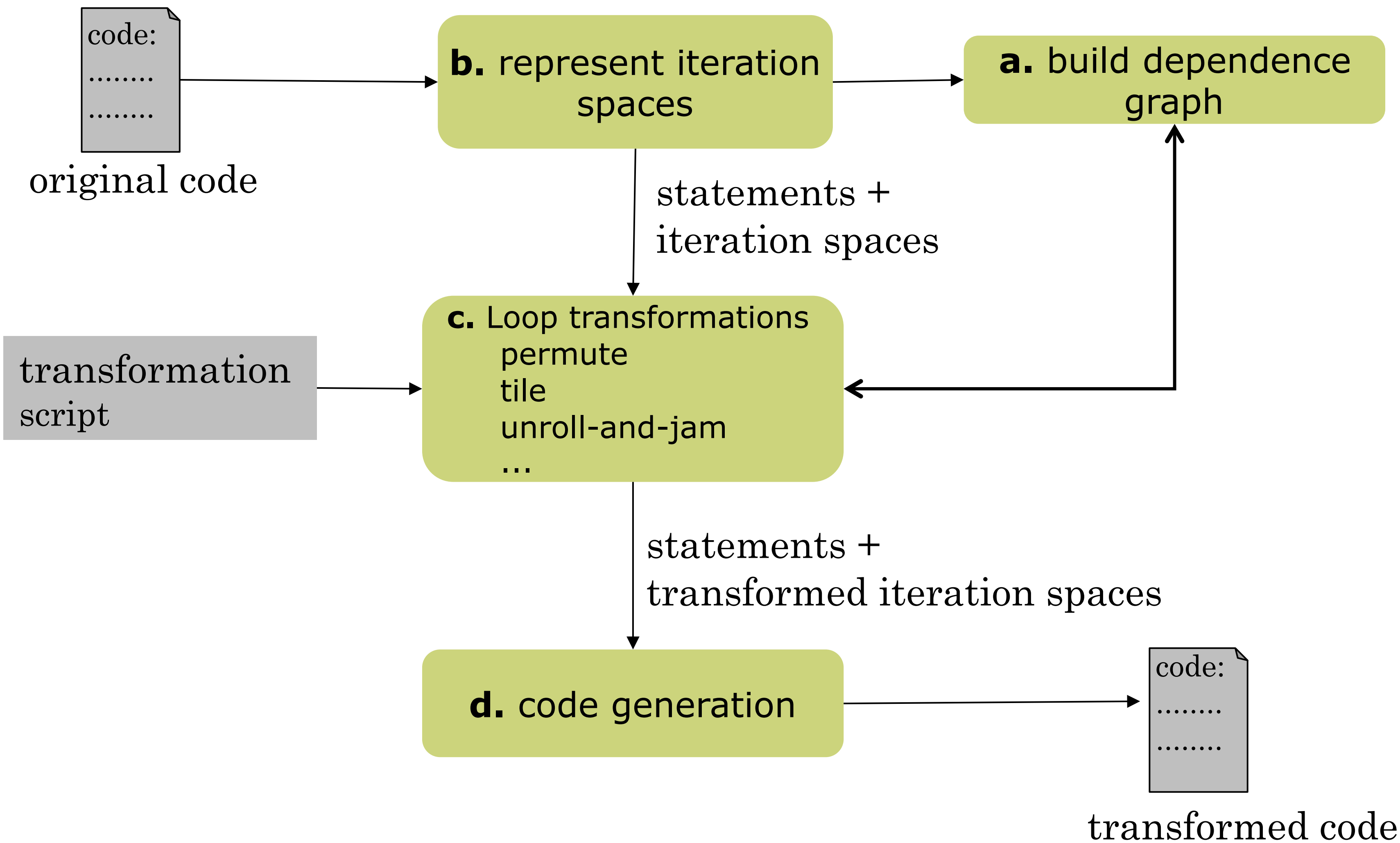
Polyhedral Compiler Technology

- Definition:
  - Represent iteration spaces of loop nests as sets of integer-valued points in regions of spaces
  - A set S is a polyhedron if it can be represented by a system of inequalities Ax <= b
- Advantages:
  - Mathematical representation provides elegant and robust representation for manipulation and code generation
  - Suitable for loop nest computations, where subscripts and loop bounds are affine
- Systems dating back to early 1990s, but renewed interest and production implementations in recent years
  - Graphite (gcc), Polly (LLVM), R-Stream (Reservoir), Omega, CLooG, PLUTO, ISL, piplib, PPL, LooPo,…

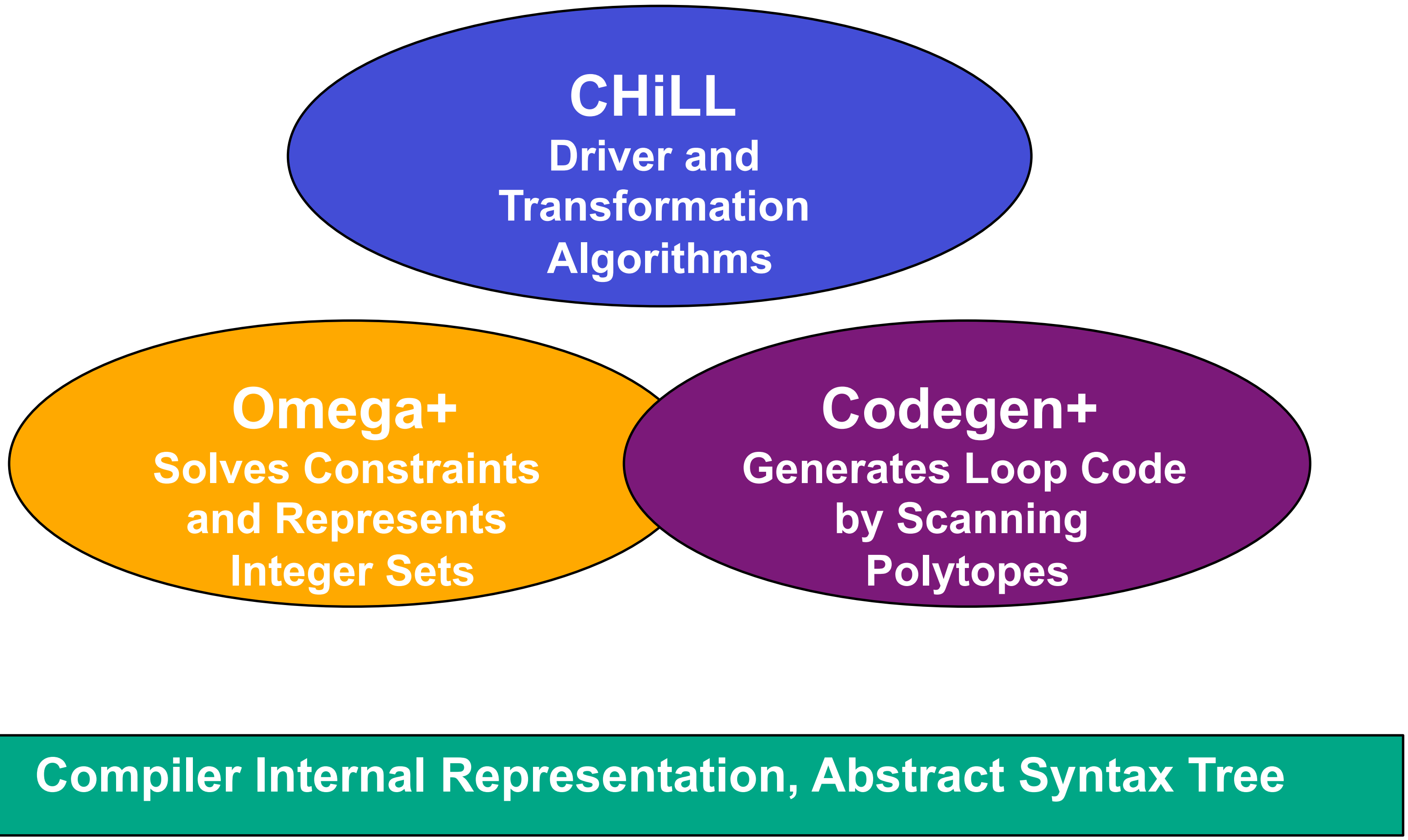
Outline for Today's Lecture

1. Abstractions
   a. Dependence graph
   b. Iteration space representation
   c. Code transformations rewrite iteration spaces
   d. Scanning polyhedra for code generation
2. More transformations: tiling, unroll-and-jam
3. Advanced concepts for imperfect loop nests
   a. Sequencing statements
   b. Aligning iteration spaces
   c. Code generation for imperfect loop nests
4. Extended example: LU without pivoting

1. Guide to Abstractions



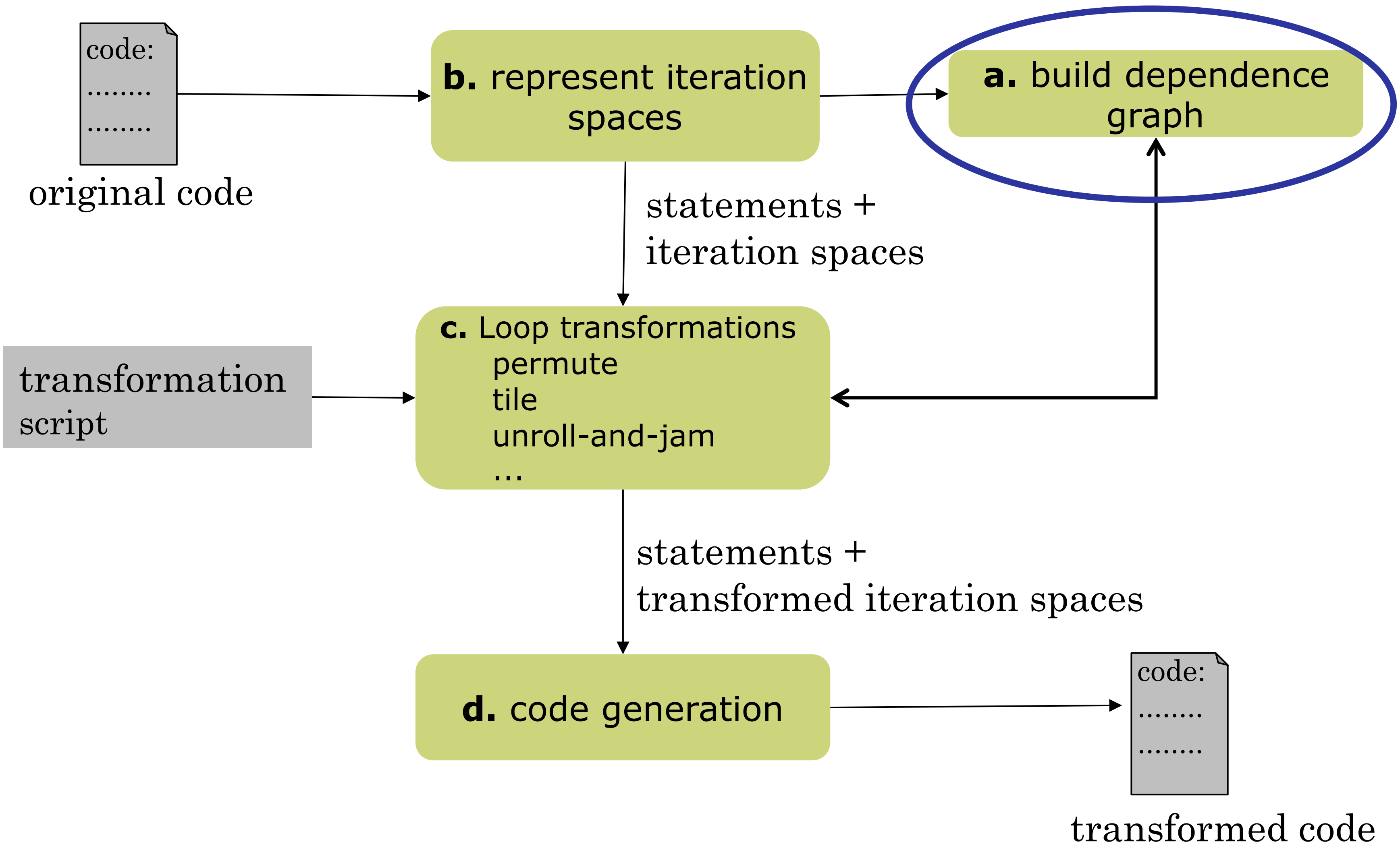
1. Guide to Implementation



1. Example: Matrix-Vector Multiply

```
for (i=0; i<100; i++)
   for (j=0; j<50; j++)
      a[i] = a[i] + c[j][i]*b[j];
```

1a. Guide to Abstractions: Dependence Graph



1a. Data Dependence

- **Definition:**
  A ***data dependence*** is an ordering on a pair of memory operations that must be preserved to maintain correctness.

  Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the references is a write.
  A data dependence can either be between two distinct program statements or two different dynamic executions of the same program statement.

- Two important uses of data dependence information (among others):
  Parallelization: no data dependence between two computations ➔ parallel execution safe
  Locality optimization: absence of data dependences & presence of reuse ➔ reorder memory accesses for better data locality

THE UNIVERSITY OF UTAH

1a. Data Dependence of Scalar Variables

True (flow) dependence

```
a        =
         = a
```

Anti-dependence

```
         = a
a        =
```

Output dependence

```
a        =
a        =
```

Input dependence (for locality)

```
         = a
         = a
```

Definition:

Data dependence exists from a reference instance I to I' iff

- either i or i' is a write operation
- I and I' refer to the same variable
- I executes before I'

1a. Fundamental Theorem of Dependence

- **Theorem 2.2 from Allen/Kennedy:**
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.

Result: Use data dependence analysis to determine whether dependences are preserved by transformations, including parallelization.

Reference: "Optimizing Compilers for Modern Architectures: A Dependence-Based Approach", Allen and Kennedy, 2002, Ch. 2.

1a. Data Dependence of Array Variables Equivalence to Integer Programming

- Determine if F(I) = G(I'), where I and I' are iteration vectors, with constraints I,I' >= L, U>= I, I'
- **Example:**

```
for (i=1; i<=100; i++)
    A[i] = A[i-1];
```

- **Inequalities:**

  1 <= iw <= 100, ir = iw - 1, ir <= 100

  integer vector I, *AI <= b*

- **Integer Programing is NP-complete**
  - O(size of the coefficients)
  - $O(n^n)$

1a. Calculating Data Dependences using Omega+ Calculator

- **Example:**

```
for (i=2; i<=100; i++)
   A[i] = A[i-1];
```

- Define relation iw = i, and iw = ir-1
  in the iteration space 2 <= i <= 100.

```
R :=  {[iw] -> [ir] :
      2 <= iw, ir <= 100            /* iteration space */
      && iw < ir                    /* looking for loop-carried true dep */
      && iw = ir-1};                /* can they be the same? */
```

R := {[iw] -> [ir] : 2 <= iw, ir <= 100 && iw < ir && iw = ir – 1};

Result: {[iw] -> [iw+1] : 2 <= iw <= 99}

1a. Dependences in Matrix-Vector Multiply

```
for (i=0; i<100; i++)
  for (j=0; j<50; j++)
     a[i] = a[i] + c[j][i]*b[j];
```

1a. Dependences in Matrix-Vector Multiply

```
for (i=0; i<100; i++)
  for (j=0; j<50; j++)
     a[i] = a[i] + c[j][i]*b[j];
```

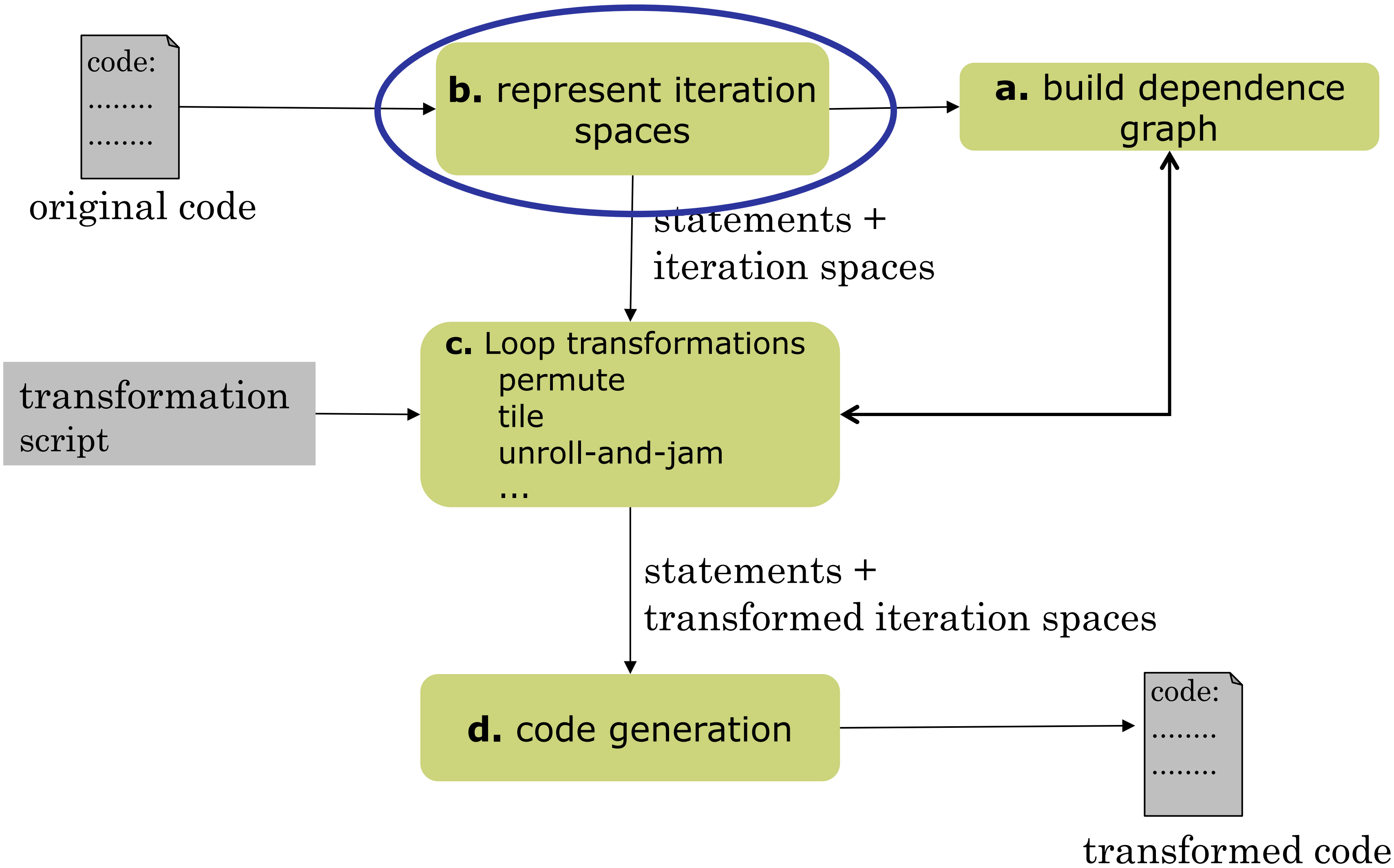
- b and c are read only: *no dependence*
- Each I=[i,j] iteration accesses the same a[i] for all 50 values of j: *dependence "carried" by j loop*

1a. How Dependences are Used in CHiLL

- Dependence graph analyzed to determine safety of code transformations and determine correctness
- After each transformation, the dependence graph is updated to maintain consistency
- An annotation allows the user to indicate that certain dependences can be ignored by the system (related to $IVDEP in vectorizing compilers)

In remainder of course, we will not discuss dependences, but their careful handling is essential to guarantee correctness

1b. Guide to Abstractions: Iteration Spaces



1b. Represent Loop Nest Iteration Space

```
for (i=0; i<100; i++)
  for (j=0; j<50; j++)
    a[i] = a[i] + c[j][i]*b[j];
```

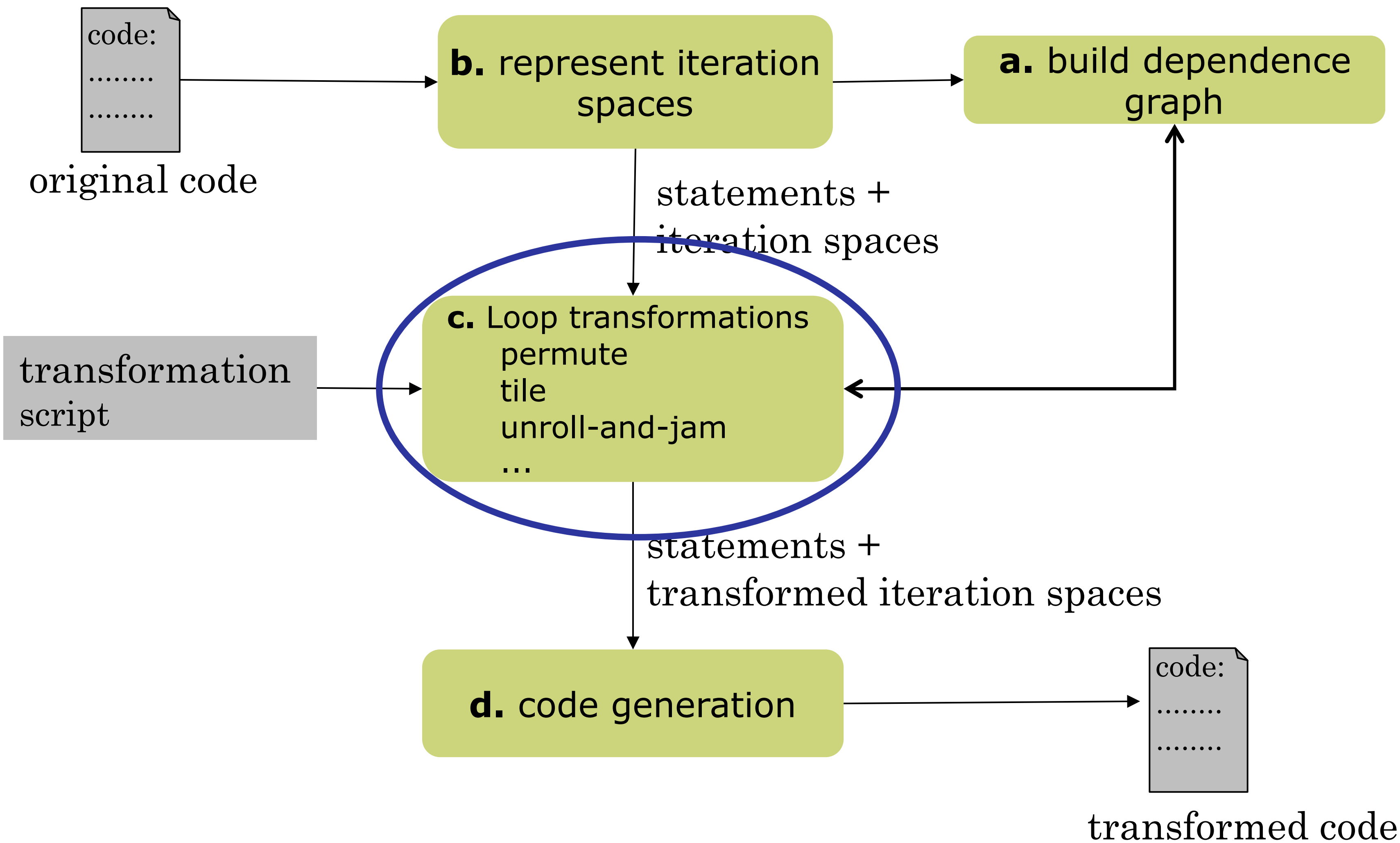
Iteration space defined by:

$I := \{[I_1,\ldots,I_n] : LB_1 <= I_1 < UB_1 \ \&\& \ \ldots \ LB_n <= I_n < UB_n\};$

In this case:

I1 := {[i,j] : 0 <= i < 100 && 0 <= j < 50};

1c. Guide to Abstractions: Transformations



THE UNIVERSITY OF UTAH

1c. Transformations Manipulate Iteration Space

```
for (i=0; i<100; i++)
  for (j=0; j<50; j++)
     a[i] = a[i] + c[j][i]*b[j];
```

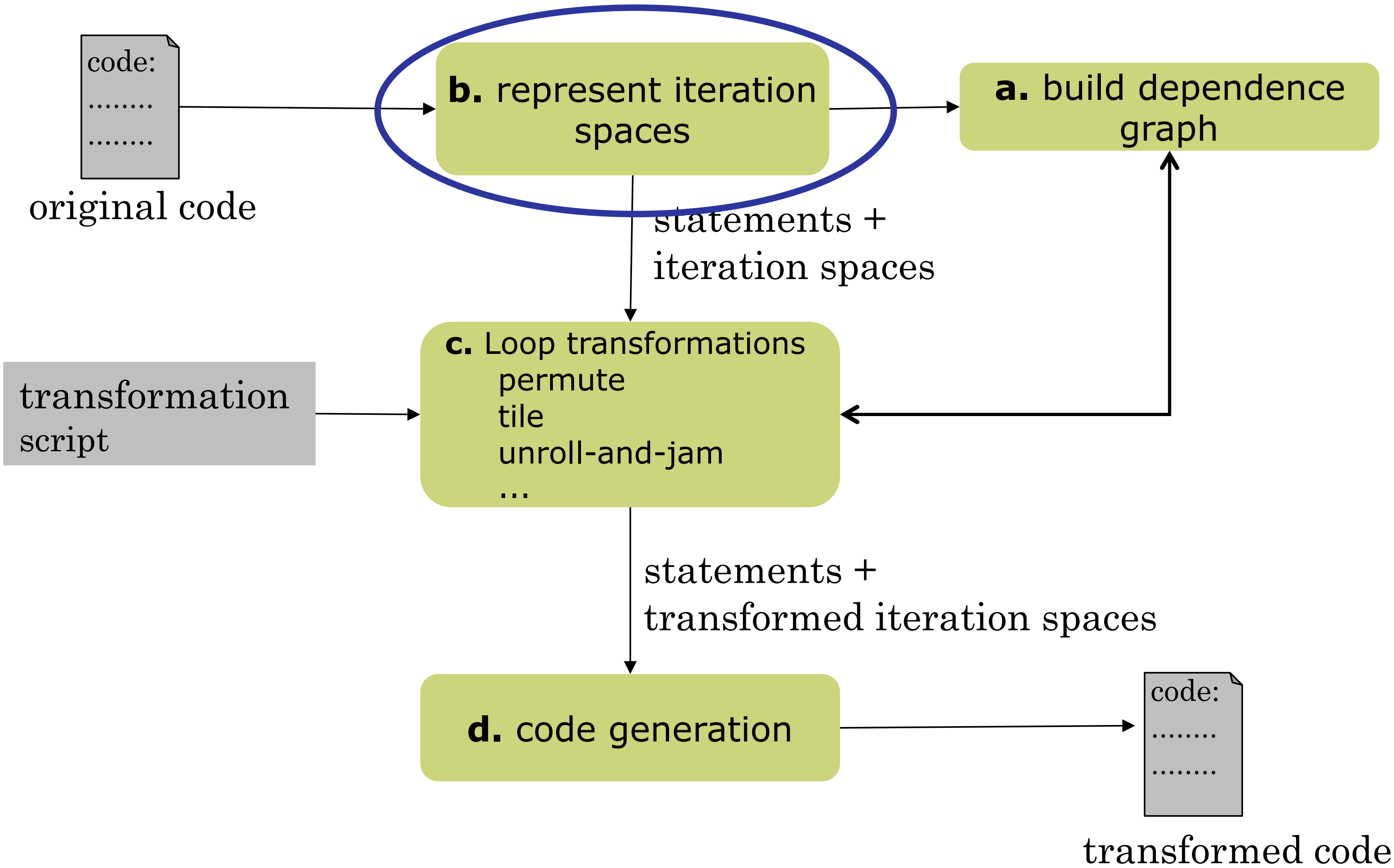
Initial iteration space:

I1 := {[i,j] : 0 <= i < 100 && 0 <= j < 50};

Permutation:

P := {[i,j] -> [j,i]};

1d. Guide to Abstractions: Code Generation



1d. Scan Polyhedra to Convert Iteration Spaces Back to Loops for Code Generation

```
for (i=0; i<100; i++)
   for (j=0; j<50; j++)
      a[i] = a[i] + c[j][i]*b[j];
```

Initial iteration space:

```
I1 := {[i,j] : 0 <= i < 100 && 0 <= j < 50};
```

Permutation:

```
P := {[i,j] -> [j,i]};
```

Generate code:

```
codegen P:I1;
```

Output of codegen I1

```
for(t1 = 0; t1 <= 99; t1++) {
  for(t2 = 0; t2 <= 49; t2++) {
    s1(t1,t2);
  }
}
```

Output of codegen P:I1

```
for(t1 = 0; t1 <= 49; t1++) {
  for(t2 = 0; t2 <= 99; t2++) {
    s1(t2,t1);
  }
}
```

2. More Transformations: Tiling

```
for (i=0; i<100; i++)
  for (j=0; j<50; j++)
    a[i] = a[i] + c[j][i]*b[j];
```

Initial iteration space:

I1 := {[i,j] : 0 <= i < 100 && 0 <= j < 50};

Tiling (i loop, tile size = 4):

T:={[i,j]->[ii,i,j] : exists (a : ii = 4a &&
a >= 0 && ii <= i < ii + 4)};

Generate code:

codegen T:I1;

Output of codegen I1

```
for(t1 = 0; t1 <= 99; t1++) {
 for(t2 = 0; t2 <= 49; t2++) {
  s1(t1,t2);
 }
}
```

Output of codegen T:I1

```
for(t1 = 0; t1 <= 96; t1 += 4) {
 for(t2 = t1; t2 <= t1+3; t2++) {
  for(t3 = 0; t3 <= 49; t3++) {
   s1(t2,t3);
  }
 }
}
```

2. More Transformations: Unroll, Unroll-and-Jam

```
for (i=0; i<100; i++)
    for (j=0; j<=i; j++)
        c[i][j] += val;
```

Initial iteration space:

I1 := {[i,j] : 0 <= i < 100 && 0 <= j <= i};

Unrolling (i loop, unroll factor = 2):

s0: c[i][j]+= val; s1: c[i+1][j]+=val;

r0:={[i,j]: exists (a: i=2a && 0<=i<100 && 0<=j<=i)};

r1:={[i,j]: exists (a: i=2a && 0<=i<100 && 0<=j<=i+1)};

Generate code:

codegen r0,r1;

Output of codegen r0, r1;

```
for(t1 = 0; t1 <= 98; t1 += 2) {
  for(t2 = 0; t2 <= t1; t2++) {
    s1(t1,t2);
    s2(t1,t2);
  }
  s2(t1,t1+1);
}
```

3. Advanced Concepts: Imperfect Loop Nests

```
          for (i=0; i<100; i++)
s0:           a[i] = 0;
              for (j=0; j<50; j++)
s1:               a[i] = a[i] + c[j][i]*b[j];
```

- Suppose each vector element is initialized to 0.
- How do we represent imperfect iteration spaces?

3a. Advanced Concepts: Sequencing in Imperfect Loop Nests

```
      for (i=0; i<100; i++)
s0:       a[i] = 0;
          for (j=0; j<50; j++)
s1:           a[i] = a[i] + c[j][i]*b[j];
```

- We add an auxiliary loop to sequence subloops in an imperfect nest.

I(s0) := {[0,i,0,j] : 0 <= i < 100 && **j = 0**};

I(s1) := {[0,i,**1**,j] : 0 <= i < 100 && 0 <= j < 50};

3b. Advanced Concepts: Aligning Imperfect Loop Nests to a Common Iteration Space

Alignment example:

```
        for (i=0; i<n; i++) {
s0:        sum[i] = 0;
           for (j=0; j<i-1; j++)
s1:           sum[i] = sum[i] + a[j][i] + b[j];
s2:        b[i] = b[i] – sum[i];
        }
```

I(s0) := {[0,i,0,j] : 0 <= i < n && j = 0};
I(s1) := {[0,i,1,j] : 0 <= i < 100 && 0 <= j < i-1};
I(s1) := {[0,I,2,j] : 0 <= i < 100 && j = i-2};

Alternative alignment for s2 (j=n-2) leads to less efficient code.

3b. Advanced Concepts: Code Generation of Imperfect Loop Nests

Iteration spaces:

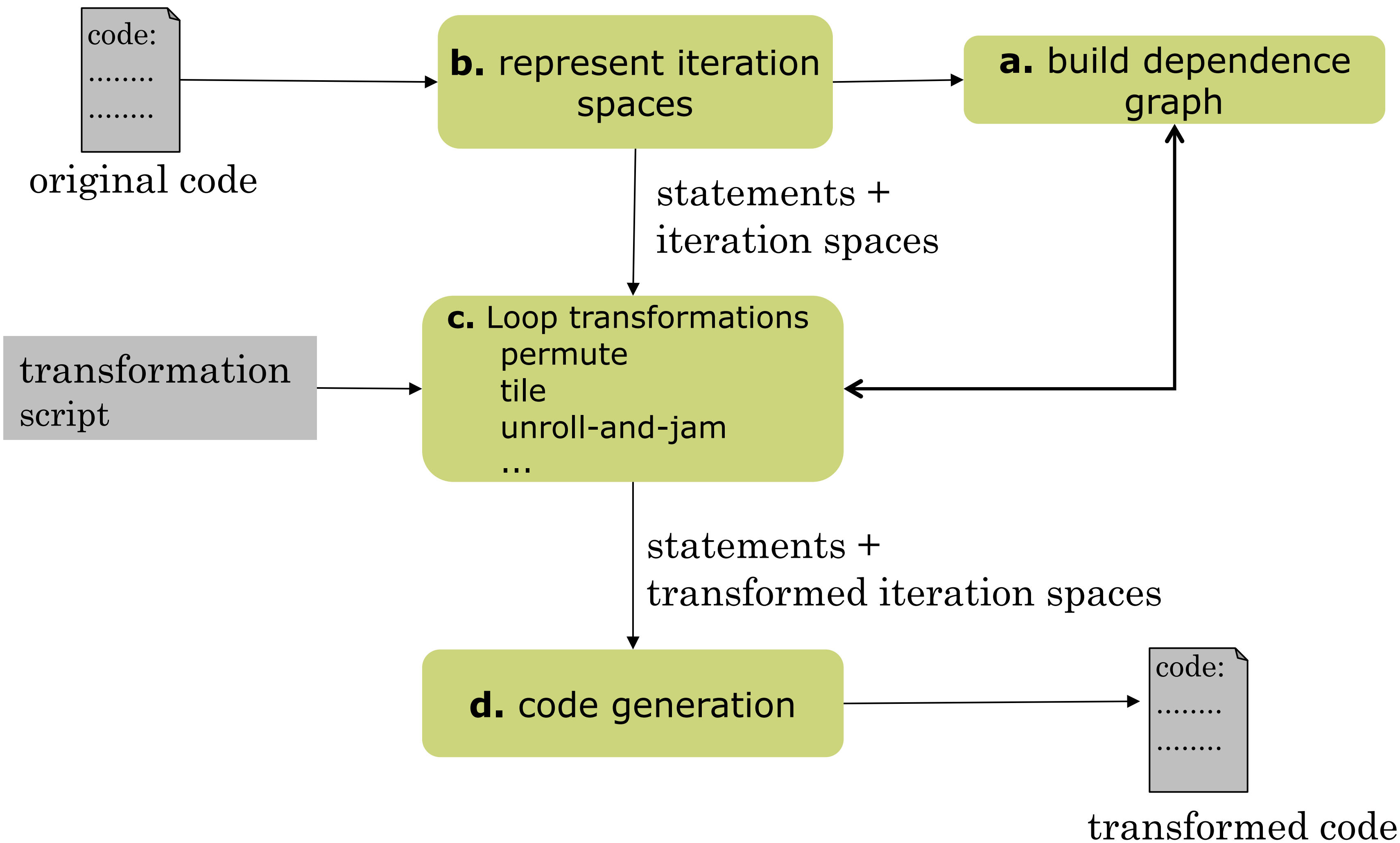
r1:={[0,i,0,j] : 0<=i<100 && j=0};
r2:={[0,i,1,j] : 0<=i<100 && 1<=j<50};
r3:={[0,i,1,j] : 0 <= i, j < 50};

- Code generation optimizes the combining of iteration spaces to derive efficient results in the presence of imperfect loop nests

Output of codegen r1, r2, r3;

```
for(t2 = 0; t2 <= 99; t2++) {
  s1(0,t2,0,0);
  if (t2 <= 49) {
    for(t4 = 0; t4 <= 49; t4++) {
      s2(0,t2,1,t4);
      s3(0,t2,1,t4);
    }
  }
  if (t2 >= 50) {
    for(t4 = 0; t4 <= 49; t4++) {
      s2(0,t2,1,t4);
    }
  }
}
```

4. LU Decomposition: Abstractions



THE UNIVERSITY OF UTAH

4. LU Decomposition: Abstractions

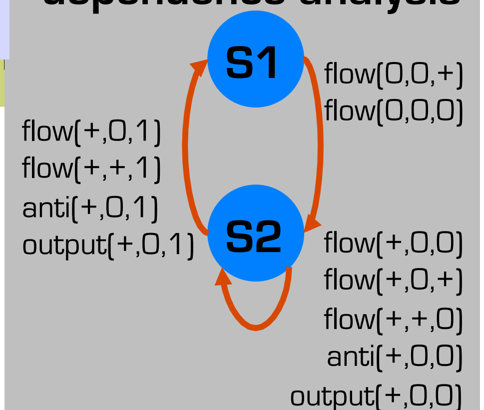
$I(s_1)$: {[k,i,j] | $1 \le k \le N-1$ && $k+1 \le i \le N$ && $j=k+1$}
$I(s_2)$: {[k,i,j] | $1 \le k \le N-1$ && $k+1 \le i,j \le N$ }

code:
....

```
    DO K=1,N-1
     DO I=K+1,N
s1     A(I,K)=A(I,K)/A(K,K)
     DO I=K+1,N
      DO J=K+1,N
s2      A(I,J)=A(I,J)-A(I,K)*A(K,J)
```

ements +
ation spaces

dependence analysis



script

unroll-and-jam

Tile and permute loops

t1 := { [k,i,j] -> [0, jj, 0, kk, 0, j, 0, i, 0, k, 0] : jj=2+16β && kk = 1+128α && j-15, 2 <= jj <=j && kk-127, 1 <= kk <= k}

t2 := { [k,i,j] -> [0, jj, 0, kk, 0, j, 0, i, 1, k, 0] : jj=2+16β && kk = 1+128α && j-15, 2 <= jj <=j && kk-127, 1 <= kk <= k}

ces

code:
....
....

transformed code

4. CHiLL Transformation Script for LU

```
    DO K=1,N-1
      DO I=K+1,N
s1      A(I,K)=A(I,K)/A(K,K)
      DO I=K+1,N
        DO J=K+1,N
s2        A(I,J)=A(I,J)-A(I,K)*A(K,J)
```

```
permute([1,2,3])
tile(1,3,Tj,1)
split(1,2,L2 ≤ L1-2)
permute(3,2,[2,4,3])
permute(1,2,[3,4,2])
split(1,2,L2 ≥L1-1)
tile(4,2,Ti1,2)
split(4,3,L5 ≤ L2-1)
tile(4,5,Tk1,3)
tile(4,5,Tj1,4)
datacopy([[4,1]],4,false,1)
datacopy([[4,2]],5)
unroll(4,5,Ui1)
unroll(4,6,Uj1)
datacopy([[5,1,]],3,false,1)
tile(1,4,Tk2,2)
tile(1,3,Ti2,3)
tile(1,5,Tj2,4)
datacopy([[1,1]],4,false,1)
datacopy([[1,2]],5)
unroll(1,5,Ui2)
unroll (1,6,Uj2)
```

TRSM: tile(4,5,Tk1,3) … datacopy([[5,1,]],3,false,1)

GEMM: tile(1,4,Tk2,2) … unroll (1,6,Uj2)

separate perfect and imperfect loop nests (points to split(1,2,L2 ≤ L1-2))

separate non-overlapping read and write accesses (points to split(1,2,L2 ≥L1-1))

CHiLL Script Source: Chun Chen

THE UNIVERSITY OF UTAH

4. Automatically-Generated LU Code

```
REAL*8 P1(32,32),P2(32,64),P3(32,32),P4(32,64)
OVER1=0
OVER2=0
DO T2=2,N,64
 IF (66<=T2)
  DO T4=2,T2-32,32
   DO T6=1,T4-1,32
    DO T8=T6,MIN(T4-1,T6+31)
     DO T10=T4,MIN(T2-2,T4+31)
      P1(T8-T6+1,T10-T4+1)=A(T10,T8)
    DO T8=T2,MIN(T2+63,N)
     DO T10=T6,MIN(T6+31,T4-1)
      P2(T10-T6+1,T8-T2+1)=A(T10,T8)
    DO T8=T4,MIN(T2-2,T4+31)
     OVER1=MOD(-1+N,4)
     DO T10=T2,MIN(N-OVER1,T2+60),4
      DO T12=T6,MIN(T6+31,T4-1)
       A(T8,T10)=A(T8,T10)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10-T2+1)
       A(T8,T10+1)=A(T8,T10+1)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+1-T2+1)
       A(T8,T10+2)=A(T8,T10+2)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+2-T2+1)
       A(T8,T10+3)=A(T8,T10+3)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+3-T2+1)
     DO T10=MAX(N-OVER1+1,T2),MIN(T2+63,N)
      DO T12=T6,MIN(T4-1,T6+31)
       A(T8,T10)=A(T8,T10)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10-T2+1)
   DO T6=T4+1,MIN(T4+31,T2-2)
    DO T8=T2,MIN(N,T2+63)
     DO T10=T4,T6-1
      A(T6,T8)=A(T6,T8)-A(T6,T10)*A(T10,T8)
```

TRSM

data copy

unroll by 4

unroll cleanup

4. Automatically-Generated LU Code

```
IF (66<=T2)
 DO T4=1,T2-33,32
  DO T6=T2-1,N,32
   DO T8=T4,T4+31
    DO T10=T6,MIN(N,T6+31)
     P3(T8-T4+1,T10-T6+1)=A(T10,T8)
   DO T8=T2,MIN(T2+63,N)
    DO T10=T4,T4+31
     P4(T10-T4+1,T8-T2+1)=A(T10,T8)
   DO T8=T6,MIN(T6+31,N)
    OVER2=MOD(-1+N,4)
    DO T10=T2,MIN(N-OVER2,T2+60),4
     DO T12=T4,T4+31
      A(T8,T10)=A(T8,T10)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10-T2+1)
      A(T8,T10+1)=A(T8,T10+1)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10+1-T2+1)
      A(T8,T10+2)=A(T8,T10+2)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10+2-T2+1)
      A(T8,T10+3)=A(T8,T10+3)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10+3-T2+1)
    DO T10=MAX(T2,N-OVER2+1),MIN(T2+63,N)
     DO T12=T4,T4+31
      A(T8,T10)=A(T8,T10)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10-T2+1)
DO T4=T2-1,MIN(N-1,T2+62)
 DO T8=T4+1,N
  A(T8,T4)=A(T8,T4)/A(T4,T4)
 DO T6=T4+1,MIN(T2+63,N)
  DO T8=T4+1,N
   A(T8,T6)=A(T8,T6)-A(T8,T4)*A(T4,T6)
```

GEMM (brace: DO T4=1,T2-33,32 through the first unroll-cleanup statement)

Mini-LU (brace: DO T4=T2-1,MIN(N-1,T2+62) to end)

data copy (points to DO T8=T4,T4+31 and DO T8=T2,MIN(T2+63,N))

unroll by 4 (points to DO T10=T2,MIN(N-OVER2,T2+60),4)

unroll cleanup (points to DO T10=MAX(T2,N-OVER2+1),MIN(T2+63,N))

Summary of Lecture

- Polyhedral compiler frameworks becoming more common
  - Mathematical manipulation of iteration spaces for transformations and code generation
  - Mostly applicable to affine domain
- Key concepts/abstractions
  - Dependence graph
  - Iteration spaces
  - Transformations rewrite iteration spaces
  - Code generation scans resulting iteration spaces to convert back to loops
- CHiLL-specific concepts
  - Auxiliary loops and alignment represent imperfect loop nests
  - Transformation and code generation algorithms manipulate this expanded iteration space

References

Other polyhedral and related compiler frameworks.

CLooG: N. Vasilache, C. Bastoul, A. Cohen, “Polyhedral Code Generation in the Real World,” Compiler Construction, A. Mycroft and A. Zeller ed., Lecture Notes in Computer Science, Springer Berlin / Heidelberg Publisher, pp. 185-201, Volume: 3923, 2006.

Graphite: J. Sjödin, S. Pop, H. Jagasia, T. Grosser, A. Pop, “Design of Graphite and the Polyhedral Compilation Package”. GCC Summit, 2009.

LooPo: M. Griebl and C. Lengauer. “The Loop Parallelizer LooPo – Announcement”. In David Sehr, editor, Languages and Compilers for Parallel Computing (LCPC ‘96), number 1239, Lecture Notes in Computer Science, pp. 603-604, Springer-Verlag, 1997.

F. Quillere, S. Rajopadhye, D. Wilde, “Generation of Efficient Nested Loops from Polyhedra”. International Journal of Parallel Processing (IJPP), Volume 28, Number 5, pp. 469-498, Oct 2000.

Omega: The Omega Calculator and Library, version 1.1.0 Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, Dave Wonnacott , Nov. 1996. http://www.cs.umd.edu/projects/omega/.

PLUTO: U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "PLUTO: A Practical and Fully Automatic Polyhedral Program Optimization System," Proc. ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), June 2008.

WraPIT: S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. International Journal of Parallel Programming, 34(3):261-317, June 2006.

THE UNIVERSITY OF UTAH