Loop Transformations

Sebastian Hack
Saarland University

Compiler Construction
W2015

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Loop Transformations: Example

matmul.c

Optimization Goals

- Increase locality (caches)
- Facilitate Prefetching (contiguous access patterns)
- Vectorization (SIMD instructions, contiguity, avoid divergence)
- Parallelization (shared and non-shared memory systems)

Dependences

- True (flow) dependence (RAW = read after write)
- Anti dependence (WAR = write after read)
- Output dependence (WAW = write after write)

Anti and output dependences are called false dependences. They only arise when we consider memory cells instead of values. SSA eliminates false dependences by renaming.

```
1: a = 1;
2: b = a;
3: a = a + b;
4: c = a;
```

If S_j is dependent on S_i, we write $S_1 \ \delta \ S_2$. Sometimes we also indicate the kind of dependence.

$$S_1 \ \delta^f \ S_2 \quad S_1 \ \delta^o \ S_3 \quad S_2 \ \delta^a \ S_3 \quad \ldots$$

Dependences

- Must be preserved for correctness
- Impose order statement instances
- Compilers represent dependences on syntactic entities (CFG nodes, AST nodes, statements, etc.)
- Each syntactic entity then stands for all its instances
- For scalar variables this is ok
- For arrays (especially in loops) this is too coarse-grained

Dependences in Loops

```
for i = 1 to 3
    1: X[i] = Y[i] + 1
    2: X[i] = X[i] + X[i-1]
```

- loop-independent flow dependence from S_1 to S_2
- loop-carried flow dependence from S_2 to S_2
- loop-carried anti dependence from S_2 to S_2

Example: GEMVER kernel

```
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
        S1: A[i,j] = A[i,j]+u1[i] * v1[j]
                   + u2[i] * v2[j]

for (k=0; k < N; k++)
    for (l=0; l < N; l++)
        S2: x[k] = x[k]+beta * A[l,k] * y[l]
```

Dependences in Loops

```
for i = 1 to 3
    1: X[i] = Y[i] + 1
    2: X[i] = X[i] + X[i-1]
```

```
X[1] = Y[1] + 1
X[1] = X[1] + X[0]
X[2] = Y[2] + 1
X[2] = X[2] + X[1]
X[3] = Y[3] + 1
X[3] = X[3] + X[2]
```

How to determine dependences in loops?

- Conceptually, unroll loops entirely.
- Every instance has then one syntactic entity.
- Construct dependence graph.

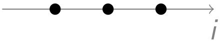
In practice, this is infeasible: Loop bounds may not be constant; even if they were, the graph would be too big.

We need a more compact representation.

Iteration Space

The iteration space of loop is the set of all iterations of that loop.

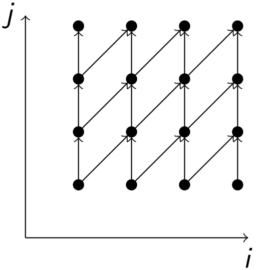
```
for i = 1 to 3
    1: X[i] = Y[i] + 1
    2: X[i] = X[i] + X[i-1]
```



In the following, we'll be interested in loop (nests) whose iteration space can be described by the integer points inside a polyhedron. Each iteration of a loop nest of depth n is then given by a n-dimensional iteration vector.

Dependence Distance Vectors

```
for i = 1 to 3
  for j = 1 to 3
    X[i,j] = X[i,j-1]
           + X[i-1,j-1]
```



Dep. vectors $(0,1),(1,1)$

- One way to represent dependences are distance vectors
- If statement instance $\vec{t}$ is dependent on instance $\vec{s}$ the distance vector for these two instances is
$$\vec{d} = \vec{t} - \vec{s}$$
- Uniform dependences are described by distance vectors that do not contain index variables.

Direction Vectors

- Used to approximate distance vectors
- Or, if dependences cannot be represented by distance vectors (non-uniform dependences)
- Vector $(\rho_1, \ldots, \rho_n)$ of "directions" $\rho_i \in \{<, \leq, =, \geq, >, *\}$
- Consider two statements s, t and all distance vectors of their instances. A direction vector ρ is legal for s and t if for all instances $\vec{s}$ and $\vec{t}$ it holds that

$$\vec{s}[k] \; \rho[k] \; \vec{t}[k] \quad \text{forall } 1 \leq k \leq n$$

- Examples
  - The distance vector $(0, 1)$ corresponds to $(=, <)$
  - The distance vector $(1, 1)$ corresponds to $(<, <)$
  - The distance vectors $\{(0, i) \mid -n \leq i \leq n\}$ correspond to $(<, *)$

Loop-Carried Dependences

```
for i = 1 to N
  for j = 1 to M
    A[i  , j  ] = A[i, j]
    B[i  , j+1] = B[i, j]
    C[i+1, j+1] = B[i, j+1]
```

- Dependence on A not loop carried
- Dependence on B carried by j loop
- Dependence on C carried by i loop

Let k be the first non-$=$ entry in the direction vector of a dependence: Dependence carried by the k-the nested loop. Dependence level is k (∞ if direction vector all $=$).

Loop Unswitching

```
for i = 1 to N
  for j = 1 to M
    if X[i] > 0
      S
    else
      T
```

```
for i = 1 to N
  if X[i] > 0
    for j = 1 to M
      S
  else
    for j = 1 to M
      T
```

- Hoist conditional as far outside as possible
- Enable other transformations

Loop Peeling

```
for i = 1 to N
  S
```

```
if N ≥ 1
  S
  for i = 2 to N
    S
```

- Align trip count to a certain number (multiple of N)
- Peeled iteration is a place where loop invariant code can be executed non-redundantly

Index Set Splitting

```
for i = 1 to N
  S
```

```
assert 1 ≤ M < N
for i = 1 to M
  S
for i = M + 1 to N
  S
```

- Create specialized variants for different cases
  e.g. vectorization (aligned and contiguous accesses)
- Can be used to remove conditionals from loops

Loop Unrolling

```
for i = 1 to N
  S
```

```
for (i = 0; i < n; i += U)
  S(i+0)
  S(i+1)
  ...
  S(i+U-1)
for (; i < N; i++)
  S(i)
```

- Create more instruction-level parallelism inside the loop
- Less specualtion on OOO processors, less branching
- Increases pressure on instruction / trace cache (code bloat)

Loop Fusion

```
for i = 1 to N
  S
for i = 1 to N
  T
```

```
for i = 1 to N
  S
  T
```

- Save loop control overhead
- Increase locality if both loops access same data
- Increase instruction-level parallelism
- Important after inlining livrary functions
- Not always legal: Dependences must be preserved

Loop Interchange

```
for i = 1 to N
  for j = 1 to M
    S
```

```
for j = 1 to M
  for i = 1 to N
    S
```

- Expose more locality
- Expose parallelism
- Legality: Preserve data dependences, direction vector $(<, >)$ forbidden

Parallelization / Vectorization

```
for i = 1 to N
    S
```

```
parallel for i = 1 to N
    S
```

- Loop must not carry dependence
- Vectorization nowadays uses SIMD code -> strip mining

Strip Mining

```
for i = 1 to N
  S
```

```
for (i = 0; i < n; i += U)
  for (j = 0; i < U; j++)
    S(i + j)
```

- strip-mine + interchange = tiling
- Vectorization is a kind of strip mining