Loop Transformations & Dependence Analysis

Performance Optimization & Parallelism

Fall 2018

Motivation & Background

- Loops are where programs spend their time!
  - "90-10" rule of thumb
  - Programs spend 90% of their execution time in 10% of the code
- Performance analysis & optimization often involves loops
  - Either the code within a loop body
  - Or the structure of the loops themselves
- This topic covers:
  - A bit of formality - loop dependence analysis
  - How to use loop dependence analysis to evaluate various loop transformation techniques
- A word about compilers
  - Optimizing compilers often analyze & attempt loop transformations
  - Sometimes compilers cannot optimize as highly as programmers
  - Good to understand what types of optimizations are possible

Data Dependencies

- A key to evaluating loop transformations
  - Would a transformation maintain code correctness?
  - For sequential code
  - For parallel loops (we'll see this again later in the semester)
- 3 types of dependences
  - Instruction A comes before instruction B in program order
  - True Dependence
    - Input of instruction B is produced as an output of instruction A
  - Anti Dependence
    - Output of instruction B is an input of instruction A
  - Output Dependence
    - Output of instruction B is an output of instruction A

Example

```
S1: x = 2;

S2: y = x;

S3: y = x + z;

S4: z = 6;
```

- Dependences:
  - S1 =>T S2
  - S1 =>T S3
  - S3 =>A S4
  - S2 =>O S3

Loop-independent vs. Loop-carried

- Loop-independent dependence
  - Dependence exists within an iteration
  - If loop is removed, the dependence still exists
- Loop-carried dependence
  - Dependence exists across iterations
  - If loop is removed, the dependence no longer exists
- Example

```
for (i=1; i<n; i++) {
    S1: a[i] = a[i-1] + 1;
    S2: b[i] = a[i];
}
```

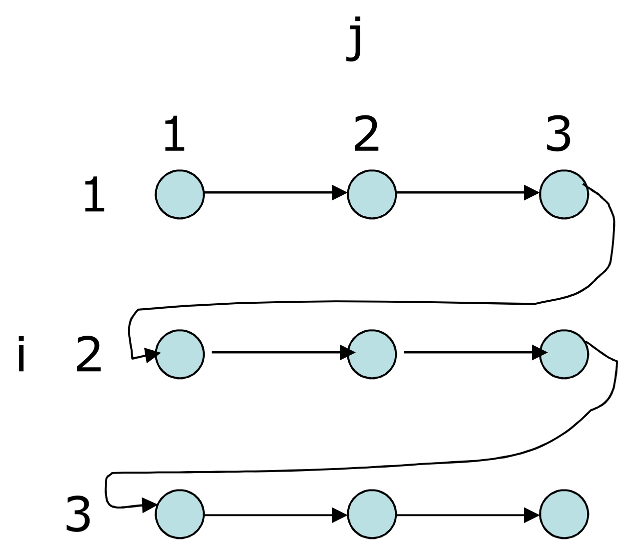
S1[i] =>t S1[i+1] (loop-carried)

S1[i] =>t S2[i] (loop-independent)

Iteration-space Traversal Graph (ITG)

- ITG shows graphically the order of traversal in the iteration space (happens-before relationship)
- Node = a point in the iteration space
- Directed Edge = the next point that will be encountered after the current point is traversed
- Example -

```
for (i=1; i<4; i++)
  for (j=1; j<4; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

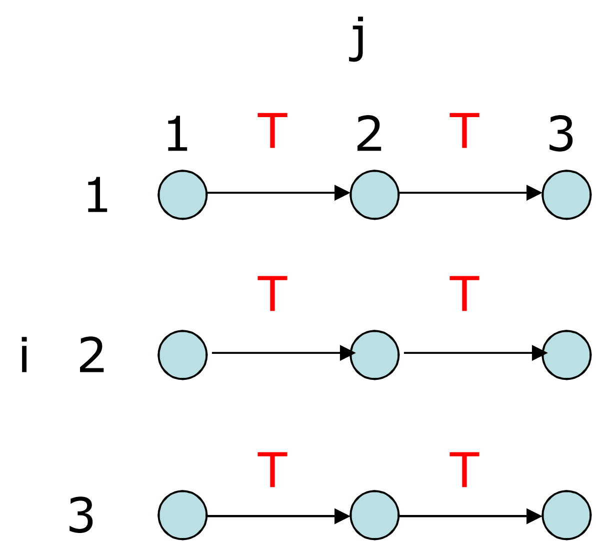


Loop-carried Dependence Graph

- LDG shows the true/anti/output dependence relationship graphically
- Node = a point in the iteration space
- Directed Edge = the dependence
- Example -

```
for (i=1; i<4; i++)
  for (j=1; j<4; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

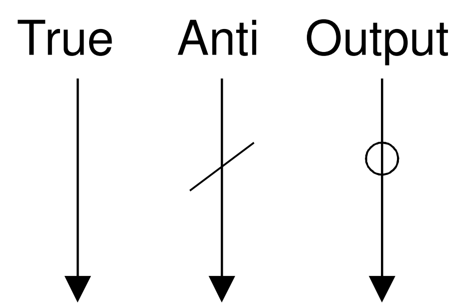
S3[i,j] $\rightarrow$T S3[i,j+1]



Additional Examples

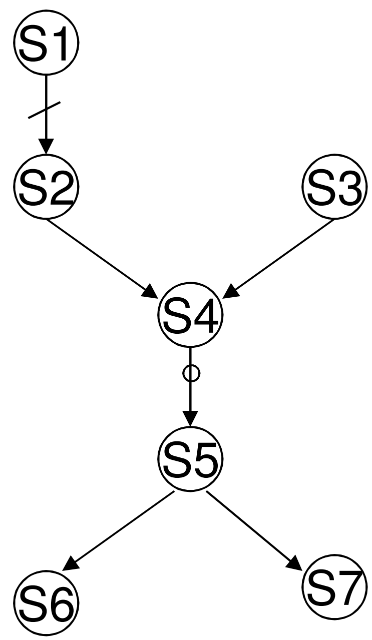
```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];

for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

- Draw the ITG
- List all the dependence relationships
- Draw the LDG



Dependences and Performance

- Parallelism within sequential code
  - ILP – instruction level parallelism



  - S1, S2 can execute in parallel with S3
  - S6 can execute in parallel with S7

Dependency Removal

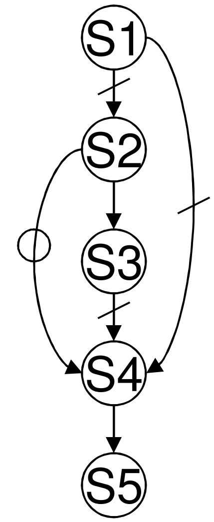
- True Dependences
  - May be fundamental to the algorithm & code
- Anti- and Output-dependences may be removed
  - Variable renaming
  - Scalar expansion
  - Node splitting

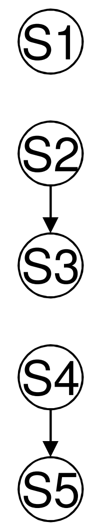
Variable Renaming

```
S1:   A = X + B
S2:   X = Y + 1
S3:   C = X + B
S4:   X = Z + B
S5:   D = X + 1
```

Variable renaming

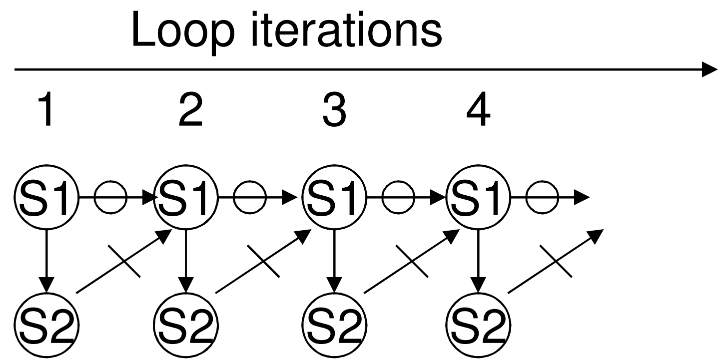
```
S1:   A = X + B
S2:   X1 = Y + 1
S3:   C = X1 + B
S4:   X2 = Z + B
S5:   D = X2 + 1
```





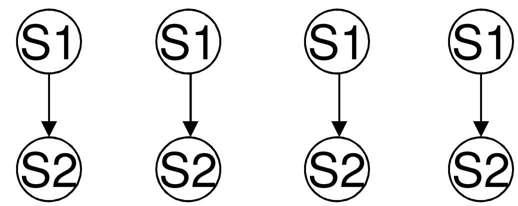
Scalar Expansion

```
for (i=1; i<=n; i++) {
    S1: a = b[i] + 1;
    S2: c[i] = a + d[i];
}
```



Scalar expansion

```
for (i=1; i<=n; i++) {
    S1: a[i] = b[i] + 1;
    S2: c[i] = a[i] + d[i];
}
```

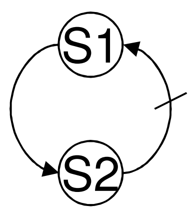


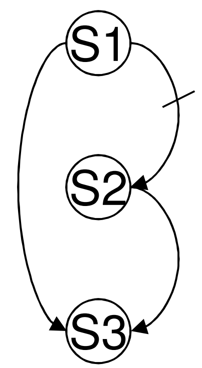
Node Splitting

- Loop-carried data-dependence cycles can sometimes be eliminated by copying data

```
for (i=0; i<=n; i++) {
    S1: a[i] = b[i] + c[i];
    S2: d[i] = (a[i] + a[i+1])/2;
}
```

```
for (i=0; i<=n; i++) {
    S1: temp[i] = a[i+1];
    S2: a[i] = b[i] + c[i];
    S3: d[i] = (a[i] + temp[i])/2;
}
```





Loop Optimizations

- Loop Invariant Hoisting
- Loop Unrolling
- Loop Fusion
- Loop Fission (Loop Distribution)
- Loop Peeling
- Loop Unswitching
- Loop Interchange
- Loop Reversal
- Loop Unroll and Jam
- Loop Strip Mining
- Loop Tiling (Strip Mine + Interchange)

Loop Invariant Hoisting

- Pull non-loop-dependent calculations out of loop

```
for (i=0; i<N; i++) {
    pi = circum/diameter;
    measurement += pi * array[i];
}
```

Loop invariant hoisting

```
pi = circum/diameter;
for (i=0; i<N; i++) {
    measurement += pi * array[i];
}
```

- Optimizing compiler would likely take care of this
  - But not always!
  - E.g. “volatile” variables used in C/C++ programs

Loop Unrolling

- 2 steps:
  - Combine multiple instances of the loop body
  - Make corresponding reduction to the loop iteration count

Original Loop

```
sum = 0
for (i=0; i<N; i++) {
    sum += array[i];
}
```

Full Unrolling

```
N = 8
sum = 0
sum += array[0];
sum += array[1];
sum += array[2];
sum += array[3];
sum += array[4];
sum += array[5];
sum += array[6];
sum += array[7];
```

Partial Unrolling

```
sum = 0
for (i=0; i<N; i+=4) {
    sum += array[i];
    sum += array[i+1];
    sum += array[i+2];
    sum += array[i+3];
}
```

Loop Unrolling

Assumptions:
- 0xBEEF is base address of array
- Array is 0x10000 bytes in size
- Register r1 tracks index into array

Original Code

```
sum = 0
for (i=0; i<N; i++) {
    sum += array[i];
}
```

```
Label:
  ld   r3 <- 0xBEEF(r1)
  add  r2 <- r3, r2
  addi r1 <- r1, 4
  cmp  0x10000, r1
  bne  Label
```

Unrolled Code

```
sum = 0
for (i=0; i<N; i+=4) {
    sum += array[i];
    sum += array[i+1];
    sum += array[i+2];
    sum += array[i+3];
}
```

```
Label:
  ld   r3 <- 0xBEEF(r1)
  ld   r4 <- 0xBEEF(r1+4)
  ld   r5 <- 0xBEEF(r1+8)
  ld   r6 <- 0xBEEF(r1+12)
  add  r2 <- r3, r2
  add  r2 <- r4, r2
  add  r2 <- r5, r2
  add  r2 <- r6, r2
  addi r1 <- r1, 16
  cmp  0x10000, r1
  bne  Label
```

Loop Unrolling

- Benefits
  - Expose additional ILP (instruction level parallelism)
  - Reduce instruction count (fewer loop management instructions)
- Drawbacks
  - Potentially use more registers
    - May cause *register spilling*
      - Save registers to memory with stores
      - Restore registers from memory later with loads
  - Increases code size => use more of the instruction cache

Loop Fusion

- Merge adjacent loops into one loop
  - Separate loops should have the same iteration sequence

```
for (i=1; i<N; i++) {
    S1: a[i] = b[i];
}

for (i=1; i<N; i++) {
    S2: c[i] = b[i] * a[i-1];
}
```

Loop fusion

```
for (i=1; i<N; i++) {
    S1: a[i] = b[i];
    S2: c[i] = b[i] * a[i-1];
}
```

Loop Fusion

- Loop Fusion is safe *iff*
  - No forward data dependence between the nests becomes a backward loop-carried data dependence (i.e. reverse the dep.)
    - Change from true dep. from A to B to anti dep. from A to B
    - Or change from anti dep. from A to B to true dep. from A to B
  - E.g. what if previous code example were altered to –

```
for (i=1; i<N; i++) {
    S1: a[i] = b[i];
}
for (i=1; i<N; i++) {
    S2: c[i] = b[i] * a[i+1];
}
```

- Benefits
  - Reduce overhead of loop management instructions
    - E.g. loop counter, loop branch
  - Increase granularity of work done in a loop
  - Improve data locality (more on this when we discuss cache perf)

Loop Fission

- Split statements within a loop body into multiple loops

```
for (i=0; i<N; i++) {
    S1: a[i] = b[i];
    S2: c[i] = c[i-1] + 1;
}
```

Loop fission

```
for (i=0; i<N; i++) {
    S1: a[i] = b[i];
}

for (i=0; i<N; i++) {
    S2: c[i] = c[i-1] + 1;
}
```

Loop Fission

- Loop Fission is safe *iff*
  - Statements involved in a cycle of loop-carried data dependences remain in the same loop AND
  - If there exists a data dependence between 2 statements placed in different loops, it must be a forward, true dependence
- In-class example…
- Benefits
  - Simplify loop body to enable other transformations
  - Improve data locality
    - Less data referenced during execution of each loop body
  - Separate memory reference streams to improve data prefetch

Loop Peeling

- Special case of Loop Splitting
- Remove first and/or last iterations of a loop body to separate code outside the loop

```
for (i=0; i<N; i++) {
    S1: a[i] = b[i] + c[i];
}
```

→ Loop peeling

```
if (N > 0) {
    a[0] = b[0] + c[0];
}

for (i=1; i<N; i++) {
    a[i] = b[i] + c[i];
}
```

Loop Peeling

- Always legal, if # of loop body executions is unchanged
- Benefits
  - Enforce a memory alignment on array references
    - E.g. to prevent misaligned memory accesses, or for vectorization
  - Sometimes special handling needed for first or last iterations
- Drawbacks
  - Additional runtime checks may be needed depending on possible values of the loop iteration count

Loop Unswitching

- Move a conditional expression outside of a loop, and replicate loop body inside of each conditional block

```
for (i=0; i<N; i++) {
    if (watermark < 2.5) {
        a[i] = a[i]*4;
    } else {
        a[i] = a[i]*2;
    }
}
```

Loop unswitching

```
if (watermark < 2.5) {
    for (i=0; i<N; i++)
        a[i] = a[i]*4;
} else {
    for (i=0; i<N; i++)
        a[i] = a[i]*2;
}
```

Loop Unswitching

- Loop Unswitching is safe *iff*
  - Evaluated conditional expression cannot change during loop execution

- Benefits
  - Reduce instruction count to execute loop iterations
  - Eliminate conditional branches from within each loop iteration

Loop Interchange

- Switch the positions of one loop that is tightly nested within another loop

```
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        S1: a[i][j] = 0;
    }
}
```

Loop interchange

```
for (j=0; j<N; j++) {
    for (i=0; i<M; i++) {
        S1: a[i][j] = 0;
    }
}
```

Loop Interchange

- Loop Interchange is safe if outermost loop does not carry any data dependence from one statement instance executed for *i* and *j* to another statement instance executed for *i'* and *j'* where ($i < i'$ and $j > j'$) OR ($i > i'$ and $j < j'$)

```
for (i=1; i<3; i++) {
    for (j=0; j<3; j++) {
        a[i][j] = a[i-1][j+1];
    }
}
```

```
for (i=1; i<3; i++) {
    for (j=0; j<3; j++) {
        a[i][j] = a[i-1][j-1];
    }
}
```

Loop Interchange

- Benefits
  - Can enable parallelization of outer and/or inner loops
    - More on this later in the semester
  - Can improve data reuse (i.e. benefit from hardware cache)
    - Languages natively support a memory layout for 2D arrays
      - Row-major order: C/C++, Python
      - Column-major order: Fortran, OpenGL, MATLAB

2-D Array

1,1	1,2	1,3
2,1	2,2	2,3

Row-major storage

Address	Value
0	1,1
1	1,2
2	1,3
3	2,1
4	2,2
5	2,3

Column-major storage

Address	Value
0	1,1
1	2,1
2	1,2
3	2,2
4	1,3
5	3,3

Loop Reversal

- Reverse the order of the loop iteration

```
for (i=0; i<N; i++) {
    a[i] = b[i] + 1;
    c[i] = a[i] / 2;
}

for (j=0; j<N; j++) {
    d[j] = d[j] + 1/c[j+1];
}
```

Loop reversal

```
for (i=N-1; i>=0; i--) {
    a[i] = b[i] + 1;
    c[i] = a[i] / 2;
}

for (j=N-1; j>=0; j--) {
    d[j] = d[j] + 1/c[j+1];
}
```

Can enable other transformations

```
for (i=N-1; i>=0; i--) {
    a[i] = b[i] + 1;
    c[i] = a[i] / 2;
    d[i] = d[i] + 1/c[i+1];
}
```

Loop Reversal

- Loop Reversal is safe *iff*
  - There are no loop-carried dependences

- Benefits
  - Can enable other loop transformations by altering dependences
  - Some ISAs contain efficient loop count instructions that count in a single direction
    - e.g. PowerPC ‘bdnz’ – branch and decrement if non-zero)

Loop Unroll and Jam

- Partially unroll one or more loops higher in the loop nest than the innermost loop, and then fuse (jam) resulting loops back together

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        a[i][j] = b[j][i];
    }
}
```

Unroll and Jam

```
for (i=0; i<N; i=i+2) {
    for (j=0; j<N; j++) {
        a[i][j] = b[j][i]
        a[i+1][j] = b[j][i+1];
    }
}
```

Loop Unroll and Jam

- (similar to loop unrolling benefits & drawbacks)
- Benefits
  - Expose additional ILP (instruction level parallelism)
  - Can improve reference locality
    - As in example on previous chart
  - Reduce instruction count (fewer loop management instructions)
- Drawbacks
  - Potentially use more registers
    - May cause *register spilling*
      - Save registers to memory with stores
      - Restore registers from memory later with loads
  - Increases code size => use more of the instruction cache

Loop Strip Mining

- Transforms singly-nested loop into doubly-nested loop
  - Outer loop steps through index in blocks of some size
  - Inner loop iterates through each block

```
for (i=0; i<N; i++) {
    a[i] = b[i] + 1;
    d[i] = b[i] - 1;
}
```

Strip Mining

```
for (j=0; j<N; j=j+32) {
    for (i=j; i<(j+32); i++) {
        a[i] = b[i] + 1;
        d[i] = b[i] - 1;
    }
}
```

Example assumes N is an even multiple of 32

Loop Strip Mining

- Strip Mining is always safe
- Block size used by outer loop is based on some attribute of the target machine
  - Cache size
  - Vector width
- Benefits
  - Can make vectorization easier
    - We'll discuss vectorization a bit more later in the semester

Loop Tiling

- Combination of Strip Mine + Interchange
- Changes traversal order of multiply-nested loops so that iteration space is traversed on a tile basis

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        c[i] = a[i][j] * b[i];
    }
}
```

Loop Tiling

```
for (i=0; i<N; i=i+2) {
    for (j=0; j<N; j=j+2) {
        for (ii=i; ii<(i+2); ii++) {
            for (jj=j; jj<(j+2); jj++)
                c[ii] = a[ii][jj] * b[ii];
        }
    }
}
```

* Loop Tiling is always safe

* Benefits similar to strip mining

* Also can improve cache reuse

Function In-lining

- Not a loop optimization
- But similar benefits as some of the loop transformations

```
void add(int *a, int *b, int *c, int i)
{
  c[i] = a[i] + b[i];
}

int main (int argc, char *argv[])
{
  int *a, *b, *c;
  //allocate & init a, b, c
  for (i=0; i<N; i++) {
    add(a, b, c, i);
  }
}
```

```
void add(int *a, int *b, int *c)
{
  *c = *a + *b;
}

int main (int argc, char *argv[])
{
  int *a, *b, *c;
  //allocate & init a, b, c
  for (i=0; i<N; i++) {
    c[i] = a[i] + b[i]
  }
}
```

Function In-lining

- Benefits
  - Reduce number of executed instructions
  - Avoid function call & return instructions
    - Which are unconditional branch instructions
  - May allow the compiler to better optimize the code
    - In and around the code of the function body
- Costs
  - Increases size of the code, which impacts the L1 instruction cache