Compiler Optimizations

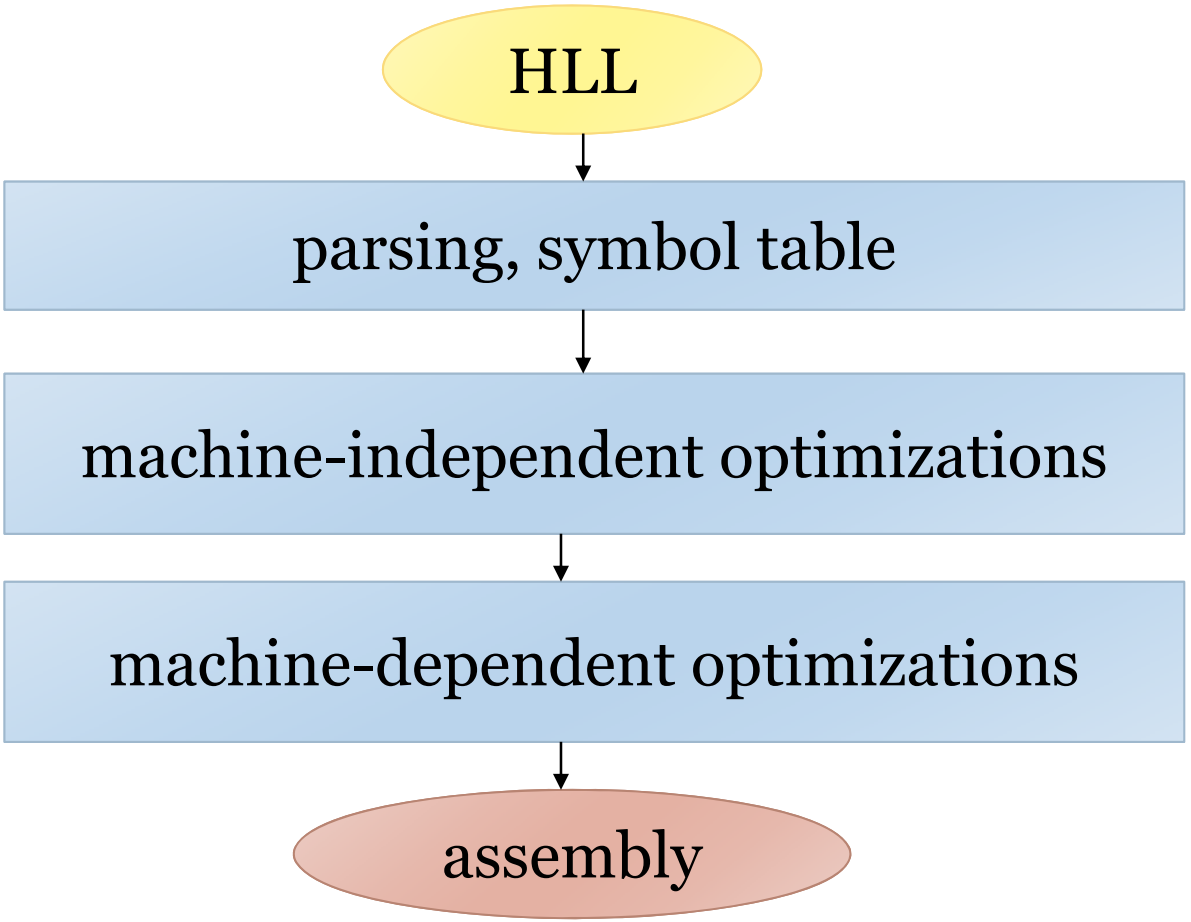
Jin-Soo Kim (*jinsookim@skku.edu*)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Compiler Basics

Compilation

- Compilation strategy (Wirth):
  - Compilation = Translation + Optimization
- Compiler determines quality of code:
  - Use of CPU resources
  - Memory access scheduling
  - Code size
  - …

Compilation Phases

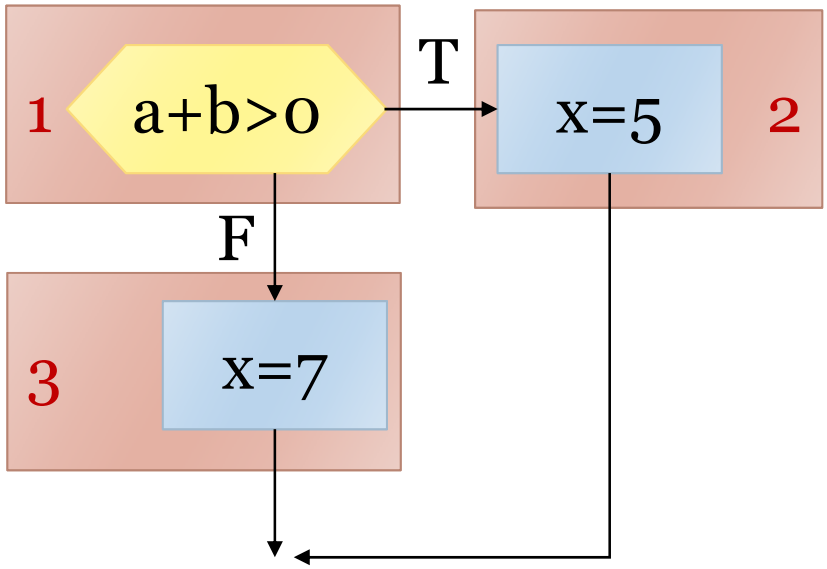


Translation and Optimization

- Source code is translated into intermediate form such as CDFG.
- CDFG is transformed/optimized.
- CDFG is translated into instructions with optimization decisions.
- Instructions are further optimized.

Example

```
if (a + b > 0)
    x = 5;
else
    x = 7;
```




```
1
    ADR     r5, a
    LDR     r1, [r5]
    ADR     r5, b
    LDR     r2, [r5]
    ADD     r3, r1, r2
    BLE     Label3

2
    LDR     r3, #5
    ADR     r5, x
    STR     r3, [r5]
    B       Exit

3
Label3:
    LDR     r3, #7
    ADR     r5, x
    STR     r3, [r5]
```

Compiler Optimizations

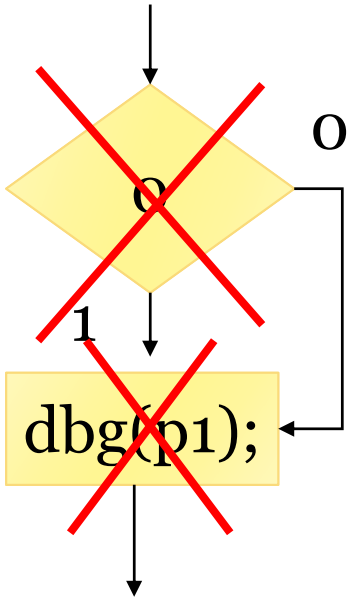
Expression Simplification

- Constant folding:
  - 8 + 1 = 9
- Algebraic:
  - a * b + a * c = a * (b + c)
- Strength reduction:
  - a * 2 = a << 1

Dead Code Elimination

- Dead code:

```
#define DEBUG 0
...
if (DEBUG)    dbg (p1);
```



- Cam be eliminated by analysis of control flow, constant folding.

Procedure Inlining

- Eliminates procedure linkage overhead:

```
int foo (a, b, c)  {
    return a + b – c;
}
...
z = foo (w, x, y);
```

```
z = w + x – y;
```

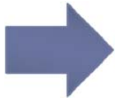
Loop Transformations

- Goals
  - Reduce loop overhead
  - Increase opportunities for pipelining
  - Improve memory system performance

Code Motion

- Move loop-invariant code outside the body of a loop

```
for (i = 0; i < n; i++) {
    x = y + z;
    a[i] = 6 * i + x * x;
}
```



```
x = y + z;
t1 = x * x;
for (i = 0; i < n; i++) {
    a[i] = 6 * i + t1;
}
```

Induction Var. Elimination

- Induction variable: loop index
- Rather than recompute (i * M + j) for each array in each iteration, share induction variable between arrays, increment at end of loop body

```
for ( i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        z[i, j] = b[i, j];
```

Loop Unrolling

- Reduces loop overhead, enables some other optimizations

```
for (i = 0; i < 4; i++)
    a[i] = b[i] * c[i];
```

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

Loop Fusion & Distribution

- Fusion combines two loops into one:

```
for (i = 0; i < N; i++)      a[i] = b[i] * 5;
for (j = 0; j < N; j++)      w[j] = c[j] * d[j];
```

```
for (i = 0; i < N; i++)  {
    a[i] = b[i] * 5;    w[i] = c[i] * d[i];
}
```

- Distribution breaks one loop into two.
- Changes optimizations within loop body.

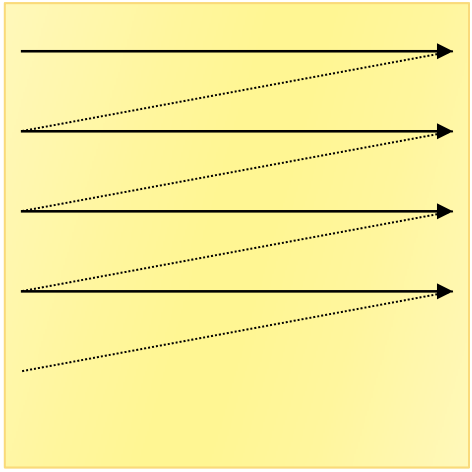
Loop Tiling

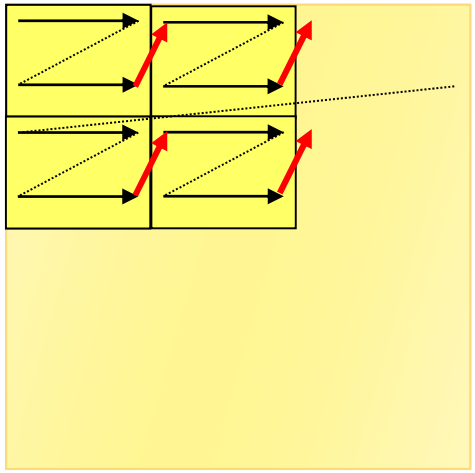
- Breaks one loop into a nest of loops
- Changes order of accesses within array
  - Changes cache behavior

Loop Tiling Example

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

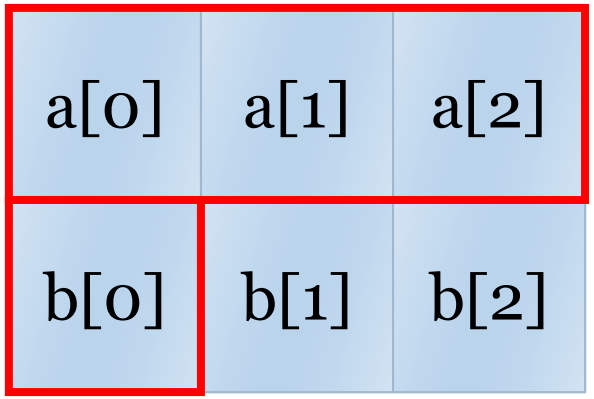
```
for (i = 0; i < N; i += 2)
  for (j = 0; j < N; j += 2)
    for (ii = i; ii < min(i+2, N); ii++)
      for (jj = j; jj < min(j+2, N); jj++)
        c[ii] = a[ii, jj] * b[ii];
```



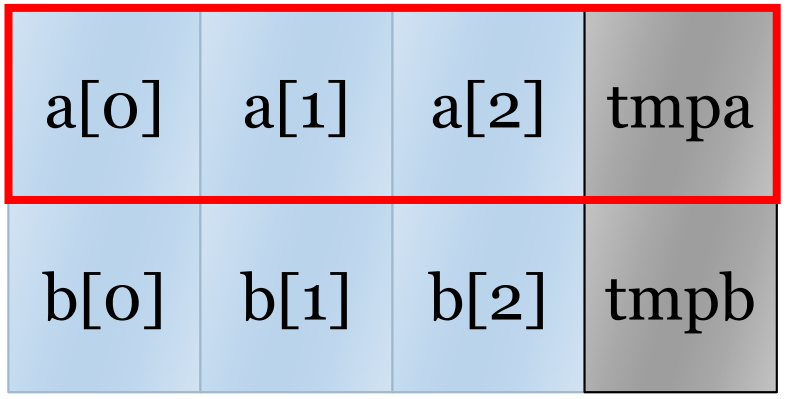


Array Padding

- Add array elements to change mapping into cache:



before



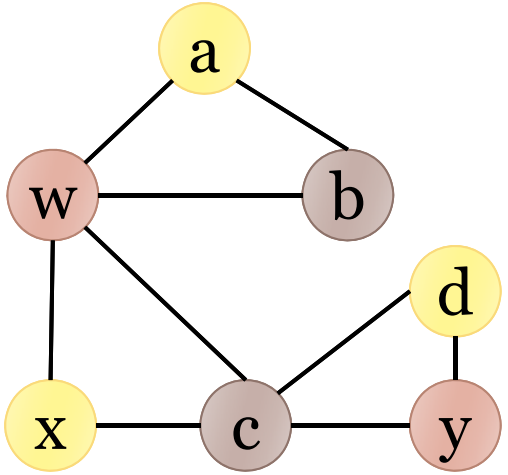
after

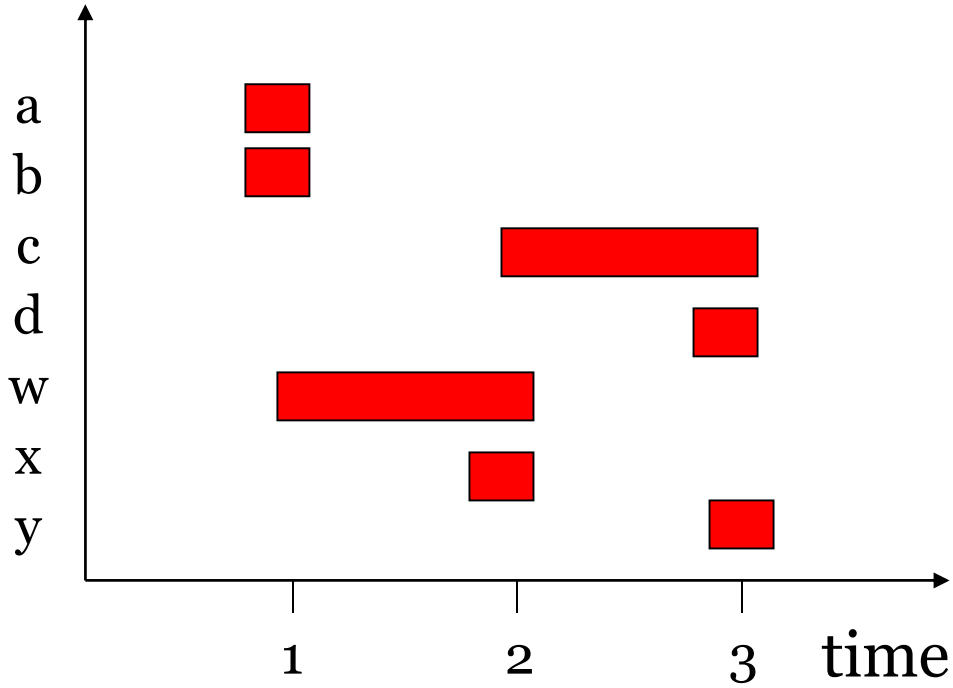
Register Allocation

- Goals
  - Choose register to hold each variable
  - Determine lifespan of variable in the register
- Basic case: within basic block
- May be improved by changing the order in which operations are performed

Register Lifetime Graph

```
w = a + b;      // t = 1
x = c + w;      // t = 2
y = c + d;      // t = 3
```





Instruction Scheduling

- Non-pipelined machines do not need instruction scheduling
  - Any order of instructions that satisfies data dependencies runs equally fast.
- In pipelined machines, execution time of one instruction depends on the nearby instructions: opcode, operands

Reservation Table

- A reservation table relates instructions/time to CPU resources

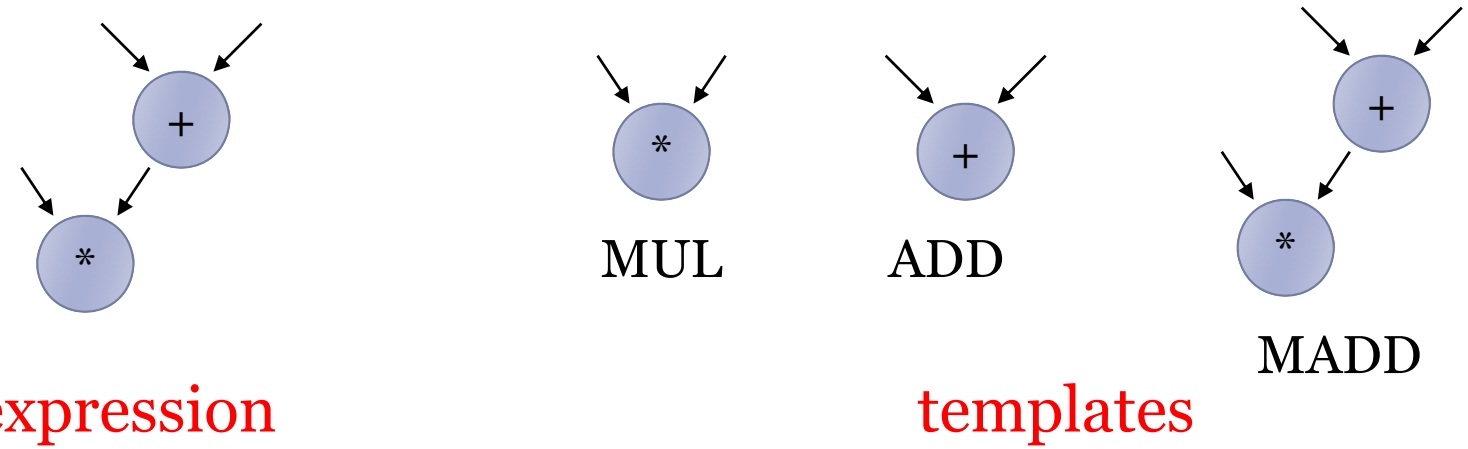
Time/instr	A	B
instr1	X	
instr2	X	X
instr3	X	
instr4		X

Software Pipelining

- Schedules instructions across loop iterations.
- Reduces instruction latency in iteration i by inserting instructions from iteration i+1.

Instruction Selection

- May be several ways to implement an operation or sequence of operations.
- Represent operations as graphs, match possible instruction sequences onto graph.



Using Your Compiler

- Understand various optimization levels
  - -O1, -O2, etc.
- Look at mixed compiler/assembler output
- Modifying compiler output requires care:
  - Correctness
  - Compiler conventions (e.g., procedure call linkage)
  - Loss of hand-tweaked code