Code Optimization

CMPT 379: Compilers
Instructor: Anoop Sarkar
anoopsarkar.github.io/compilers-class

Code Optimization

- There is no fully optimizing compiler *O*
- Let's assume *O* exists. It takes a program P and produces output **Opt**(P) which is the *smallest* possible
- Imagine a program Q that produces no output and never terminates, then **Opt**(Q) could be:
  ```
  L1: goto L1
  ```
- Then to check if a program P never terminates on some inputs, check if **Opt**(P(i)) is equal to **Opt**(Q) = Solves the Halting Problem
- Full Employment Theorem for Compiler Writers, see Rice(1953)

Optimizations

- Non-Optimizations
- Correctness of optimizations
  - Optimizations must not change the meaning of the program
- Types of optimizations
  - Local optimization and peephole optimization
  - ~~Global dataflow analysis for optimization~~
  - Static Single Assignment (SSA) Form
- Amdahl's Law

Non-Optimizations

```
enum { GOOD, BAD };
extern int test_condition();

void check() {
  int rc;

  rc = test_condition();
  if (rc != GOOD) {
    exit(rc);
  }
}
```

```
enum { GOOD, BAD };
extern int test_condition();

void check() {
 int rc;

 if ((rc = test_condition())) {
   exit(rc);
 }
}
```

Which version of check runs faster?

Types of Optimizations

- High-level optimizations
  - function inlining
- Machine-dependent optimizations
  - e.g., peephole optimizations, instruction scheduling
- Local optimizations or Transformations
  - within basic block

Types of Optimizations

- Global optimizations or Data flow Analysis
  - across basic blocks
  - within one procedure (*intraprocedural*)
  - whole program (*interprocedural*)
  - pointers (*alias analysis*)

Maintaining Correctness

- What does this program output?

3

Not:

$ decafcomp byzero.decaf

Divide by zero exception

```
int main() {
    int x;
    if (false) {
        x = 3/(3-3);
    } else {
        x = 3;
    }
    print_int(x);
}
```

branch delay slot (cf. **load delay** slot)

Peephole Optimization

- Redundant instruction elimination
  - If two instructions perform that same function ***and*** are in the same basic block, remove one
  - Redundant loads and stores

    load t1 = 3

    load t1 = 4
  - Remove unreachable code

Peephole Optimization

- Flow control optimization

```
  goto L1
L1:
  goto L2
```

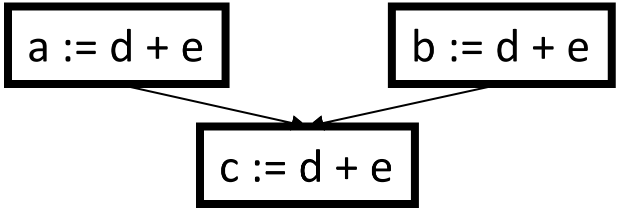
- Algebraic simplification
- Reduction in strength
  - Use faster instructions whenever possible
- Use of Machine Idioms
- Filling delay slots

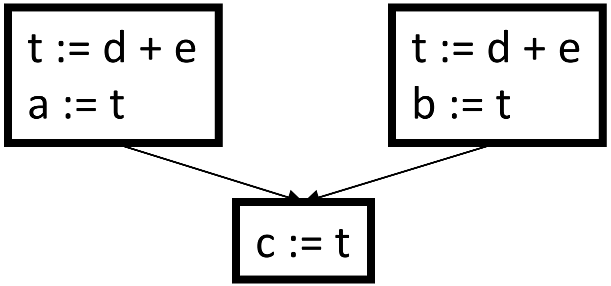
Constant folding & propagation

- Constant folding
  - compute expressions with known values at compile time
- Constant propagation
  - if constant assigned to variable, replace uses of variable with constant unless variable is reassigned

Constant folding & propagation

- Copy Propagation





Transformations

- Structure preserving transformations
- Common subexpression elimination

  a := b + c

  b := a - d

  c := b + c

  d := a - d ($\Rightarrow$ b)

Transformations

- Dead-code elimination (combines copy propagation with removal of unreachable code)

```
if (debug) { f(); } /* debug :=  false (as a constant) */
if (false) { f(); }  /* constant folding */
```

using dead-code elimination, code for f() is removed

```
x := t3                 x := t3
t4 := x     becomes    t4 := t3
```

Transformations

- Renaming temporary variables

  t1 := b+c can be changed to t2 := b+c

  replace all instances of t1 with t2

- Interchange of statements

  t1 := b+c   t2 := x+y

  t2 := x+y   can be converted to t1 := b+c

(Can be combined with branch delay slots or load delay slots)

Transformations

- Algebraic transformations

  d := a + 0 ($\Rightarrow$ a)

  d := d * 1 ($\Rightarrow$ *eliminate*)

- Reduction of strength

  d := a ** 2 ($\Rightarrow$ a * a)

Code Optimization for SSA Form

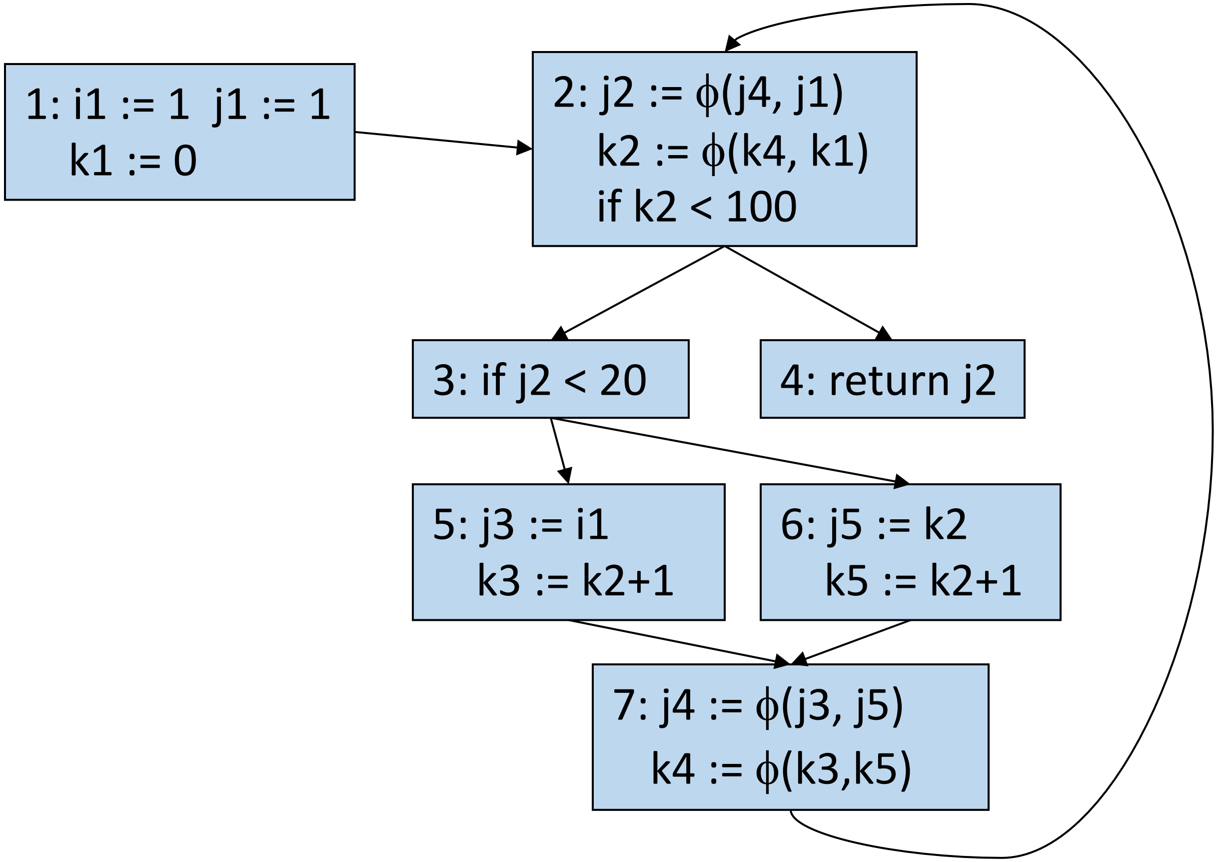
Optimizations using SSA

- SSA form contains *statements*, *basic blocks* and *variables*
- Dead-code elimination
  - if there is a variable *v* with no *use*s and *def* of *v* has no side-effects, delete statement defining *v*
  - if $z := \phi(x, y)$ then eliminate this stmt if no *def*s for *x,y*

Optimizations using SSA

- Constant Propagation
  - if $v := c$ for some constant c then replace v with c for all uses of v
  - $v := \phi\ (c1, c2, \ldots, cn)$ where all c_i are equal to c can be replaced by $v := c$
  - In practice, all phi functions will be binary: $\phi\ (c1, c2)$

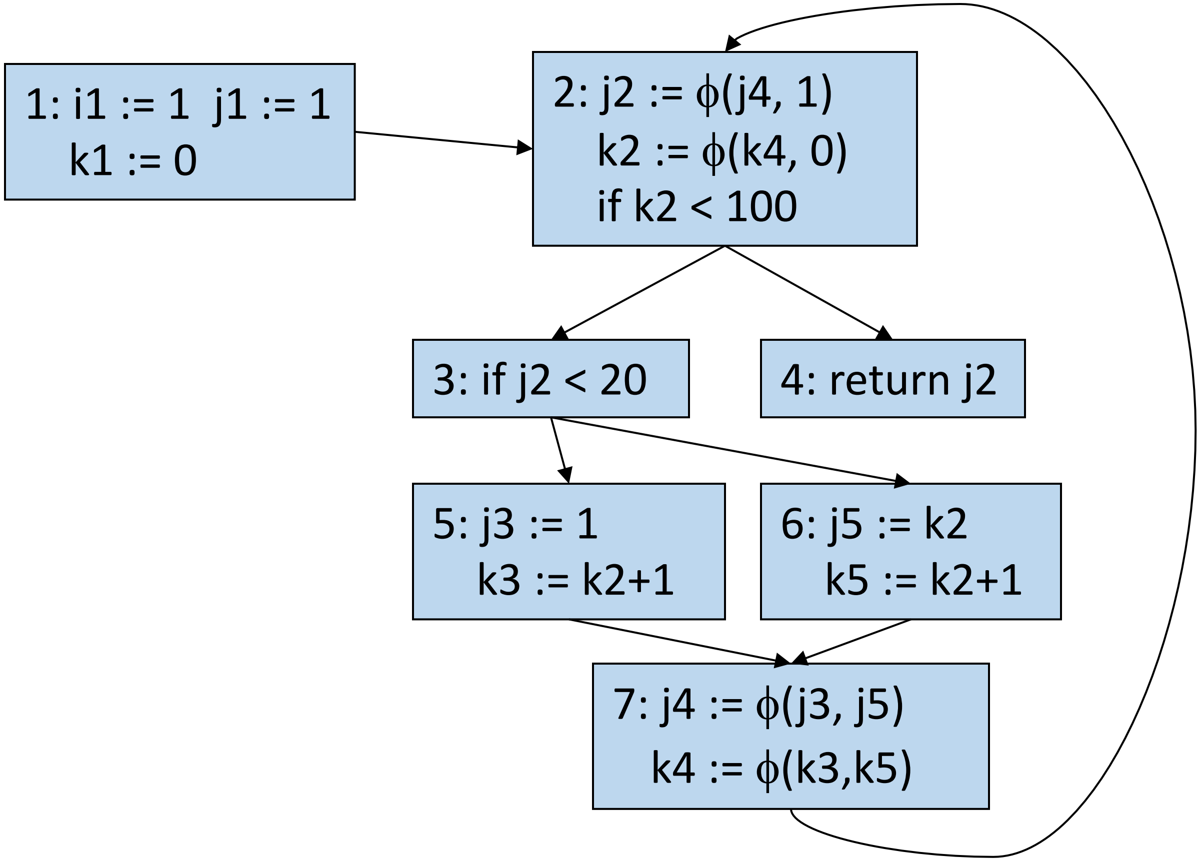
Optimizations using SSA



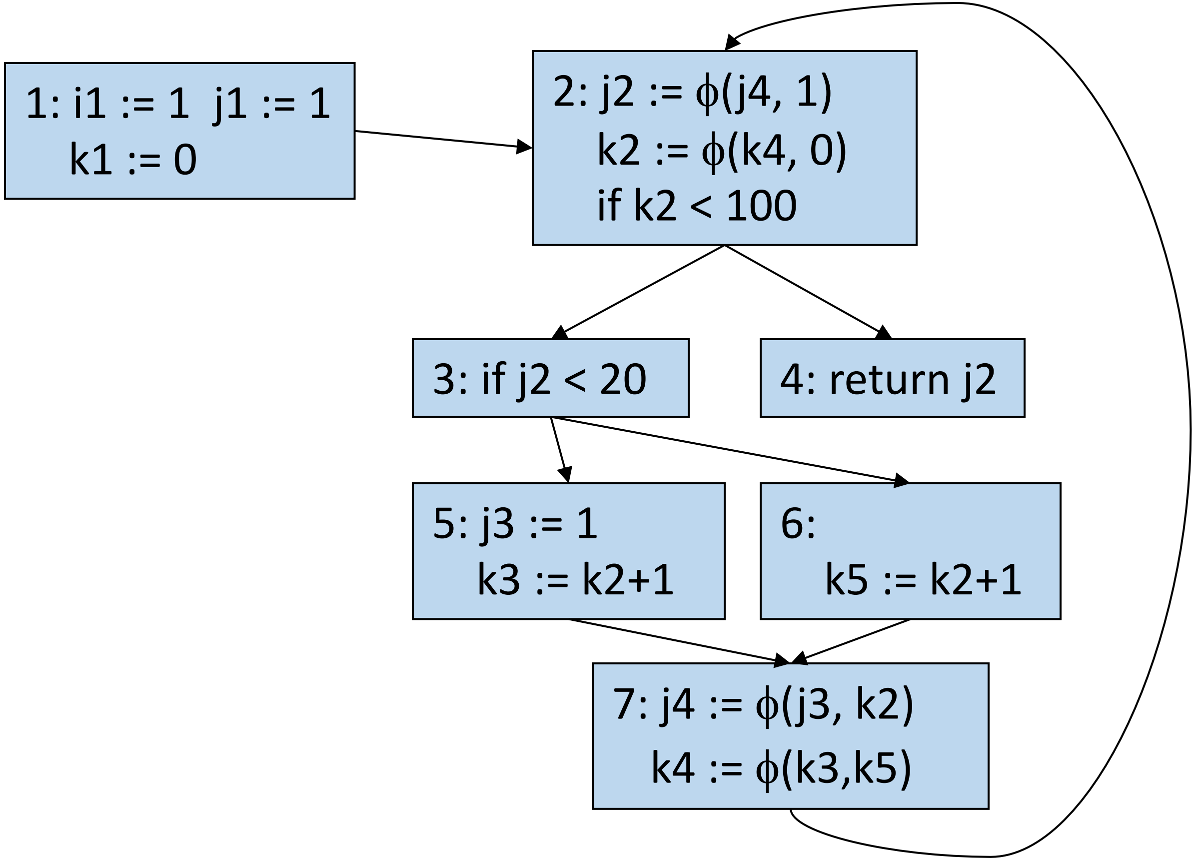
Optimizations using SSA

- Conditional Constant Propagation
  - In previous flow graph, is j always equal to 1?
  - If j = 1 always, then block 6 will never execute and so j := i and j := 1 always
  - If j > 20 then block 6 will execute, and j := k will be executed so that eventually j > 20
  - Which will happen? Using SSA we can find the answer.

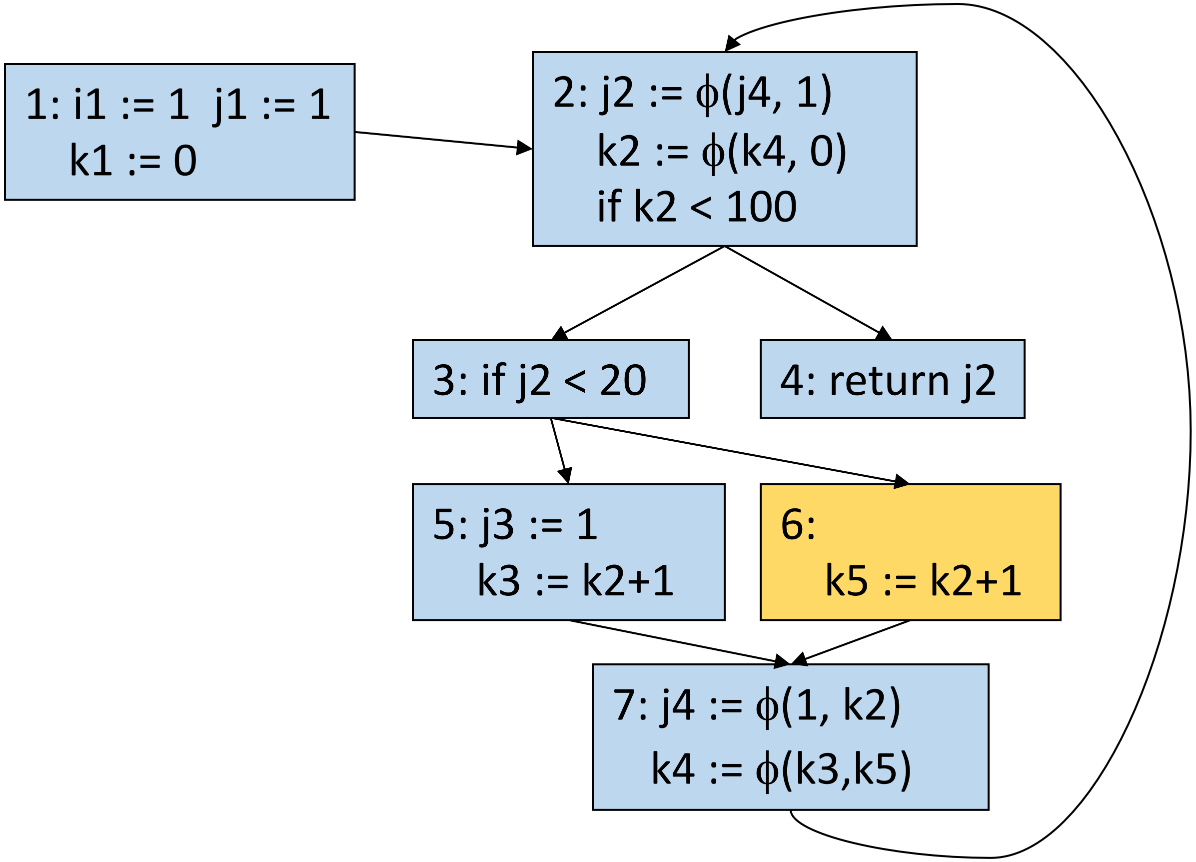
Optimizations using SSA



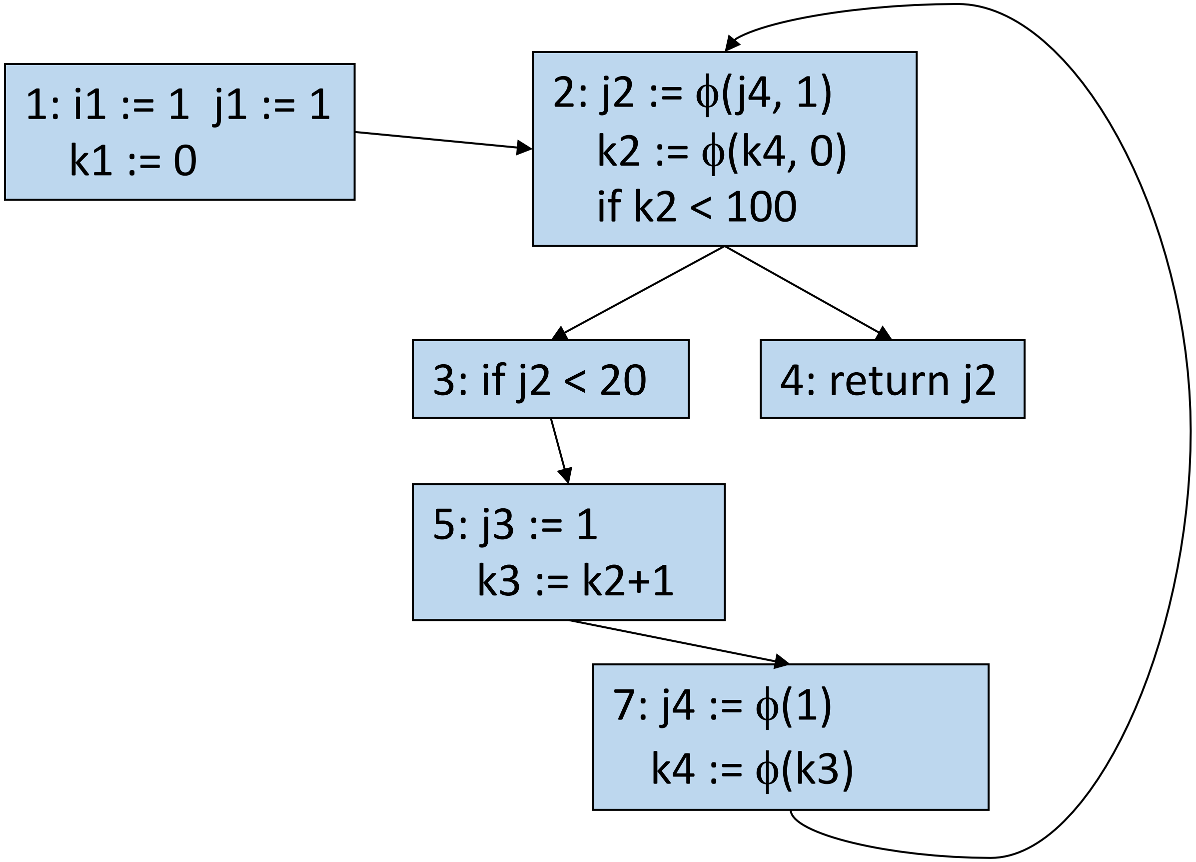
Optimizations using SSA



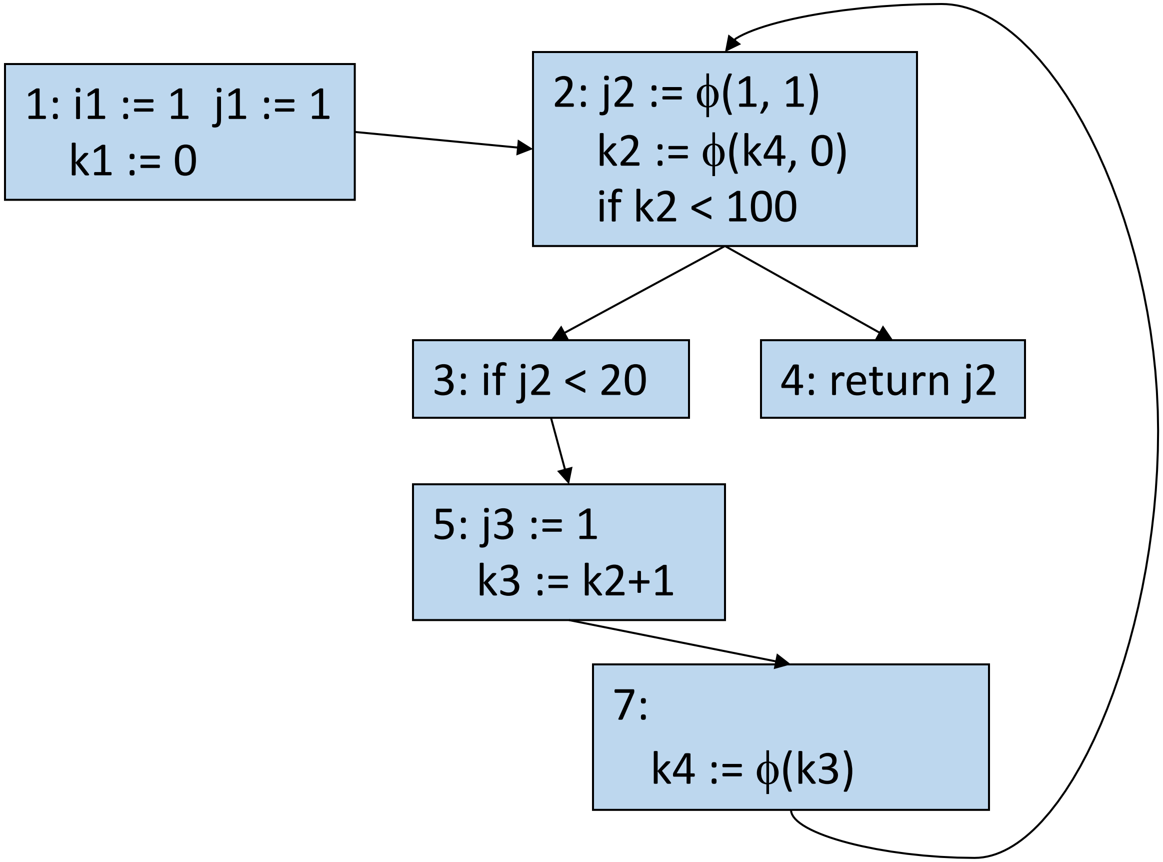
Optimizations using SSA



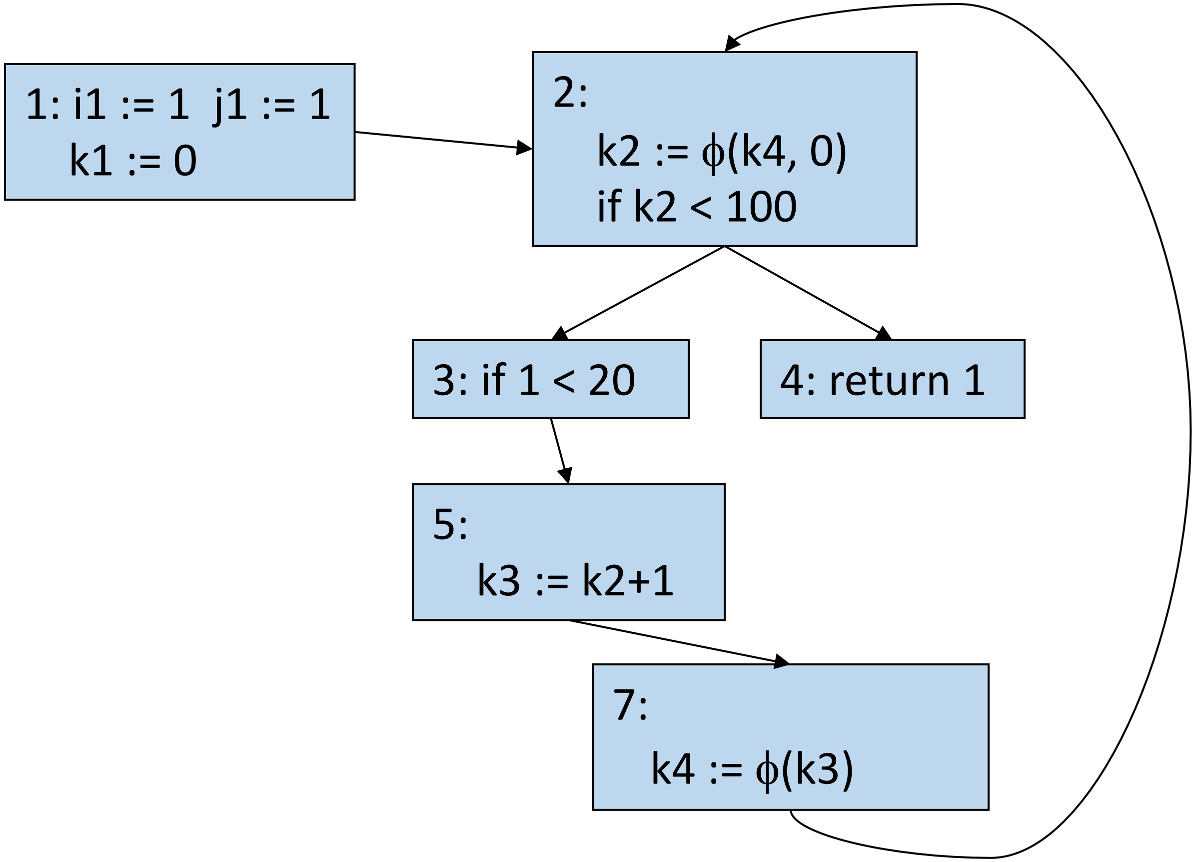
Optimizations using SSA



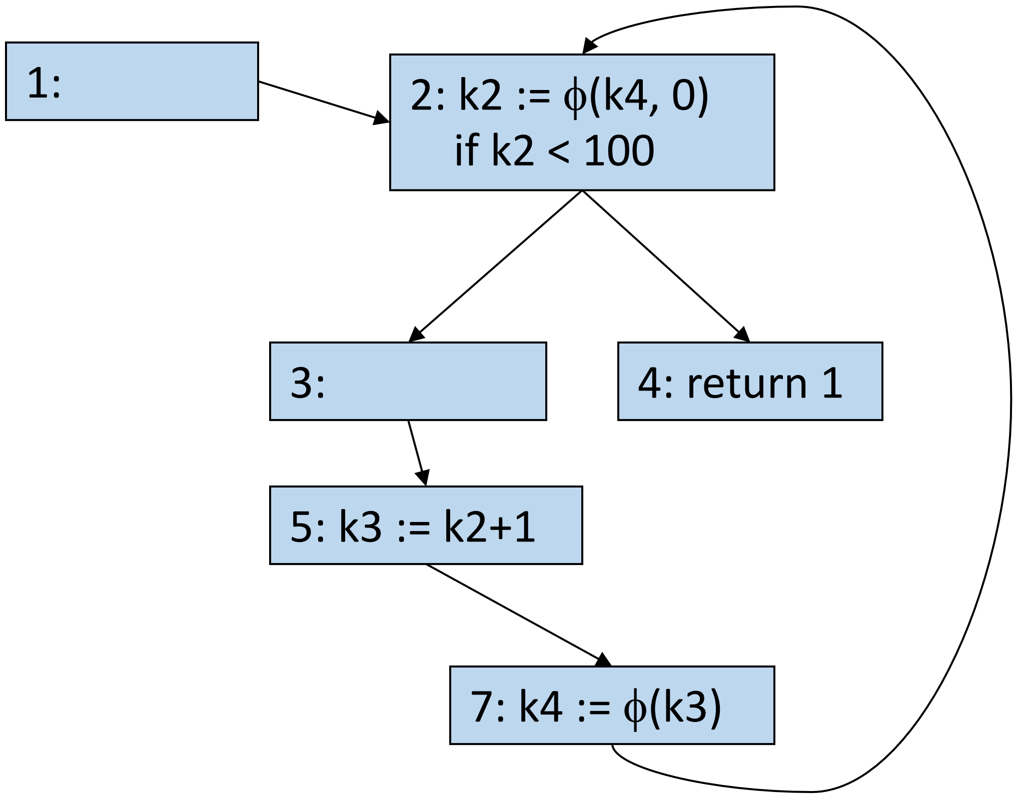
Optimizations using SSA



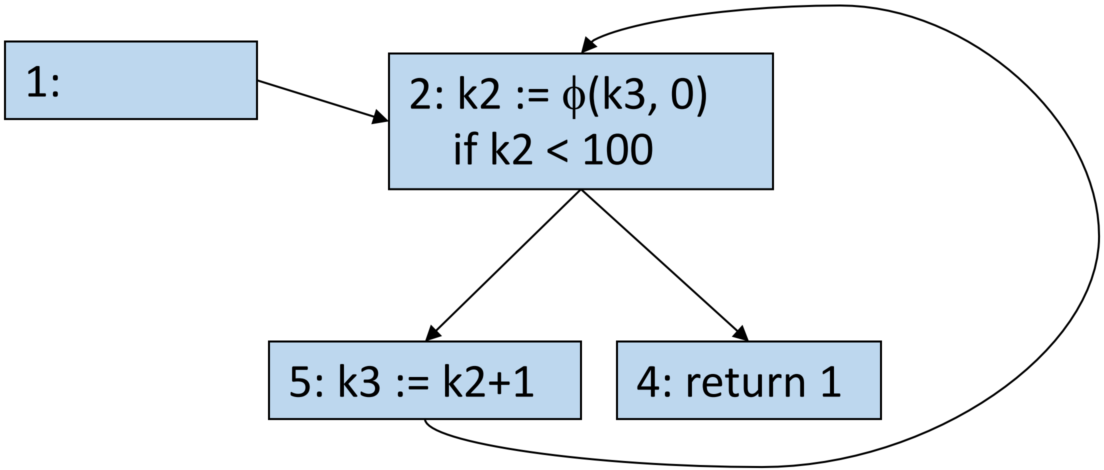
Optimizations using SSA



Optimizations using SSA



Optimizations using SSA



Optimizations using SSA

- Arrays, Pointers and Memory
  - For more complex programs, we need *dependencies*: how does statement B depend on statement A?
  - **Read after write**: A defines variable *v*, then B uses *v*
  - **Write after write**: A defines *v*, then B defines *v*
  - **Write after read**: A uses *v*, then B defines *v*
  - **Control**: A controls whether B executes

Optimizations using SSA

- Memory dependence

  M[i] := 4

  x := M[j]

  M[k] := j

- We cannot tell if *i, j, k* are all the same value which makes any optimization difficult
- Similar problems with Control dependence
- SSA does not offer an easy solution to these problems

More on Optimization

- Control Flow Analysis
- Data Flow Analysis
- Dependence Analysis
- Alias Analysis
- Early Optimizations
- Redundancy Elimination
- Loop Optimizations
- Procedure Optimizations
- Code Scheduling (pipelining)
- Low-level Optimizations
- Interprocedural Analysis
- Memory Hierarchy

- *Advanced Compiler Design and Implementation* by Steven S. Muchnick

Amdahl’s Law

- $\text{Speedup}_{\text{total}} =$
  $((1 - \text{Time}_{\text{Fractionoptimized}}) + \text{Time}_{\text{Fractionoptimized}}/\text{Speedup}_{\text{optimized}})\text{-}1$
- Optimize the common case, 90/10 rule
- Requires quantitative approach
  - Profiling + Benchmarking
- Problem: Compiler writer doesn’t know the application beforehand

Summary

- Optimizations can improve speed, while maintaining correctness
- Many types of local optimizations
- Static Single-Assignment Form (SSA)
- Optimization using SSA Form