PERFORMANCE ENGINEERING OF SOFTWARE SYSTEMS

6.172

What Compilers Can and Cannot Do

Saman Amarasinghe
Fall 2010

Outline

- Cool compiler hacks (and some failures)
- When to optimize
- Data-flow Analysis and Optimizations
- Instruction Scheduling

Do you need to inline?

```
#define max1(x,y) ((x)>(y)?(x):(y))

static uint64_t max2(uint64_t x,
                     uint64_t y)
{
  return (x>y)?x:y;
}


uint64_t first(uint64_t a, uint64_t b)
{
  return max1(a, b);
}

uint64_t second(uint64_t a, uint64_t b)
{
  return max2(a, b);
}
```

```
first:
        cmpq    %rdi, %rsi
        cmovae  %rsi, %rdi
        movq    %rdi, %rax
        ret

second:
        cmpq    %rdi, %rsi
        cmovae  %rsi, %rdi
        movq    %rdi, %rax
        ret
```

GCC knows bithacks!

```
uint64_t mul4(uint64_t a)
{
  return a*4;
}
```

```
mul4:
        leaq    0(,%rdi,4), %rax
        ret
```

```
uint64_t mul43(uint64_t a)
{
  return a*43;
}
```

```
mul43:
        leaq    (%rdi,%rdi,4), %rax    # %rax=a+4*a
        leaq    (%rdi,%rax,4), %rax    # %rax=a+20*a
        leaq    (%rdi,%rax,2), %rax    # %rax=a+42*a
        ret
```

```
uint64_t mul254(uint64_t a)
{
  return a*254;
}
```

```
mul254:
        leaq    (%rdi,%rdi), %rax    # %rax=2*a
        salq    $8, %rdi             # %rdi=128*(2*a)
        subq    %rax, %rdi           # %rdi = 256*a–2*a
        movq    %rdi, %rax           # %rax = 254*a
        ret
```

GCC knows bithacks!

```
int abs(int a)
{
  return (a>0)?a:(-a);
}
```

```
abs:
       movl    %edi, %edx
        sarl    $31, %edx
        movl    %edx, %eax
        xorl    %edi, %eax
        subl    %edx, %eax
        ret
```

Eliminate unnecessary tests

```
static char A[1048576];

int update(int ind, char val)
{
  if(ind>=0 && ind<1048576)
    A[ind] = val;
}
```

```
update:
        cmpl    $1048575, %edi
        ja      .L4
        movslq  %edi,%rax
        movb    %sil, A(%rax)
.L4:
        rep
        ret
```

Eliminate unnecessary tests II

```
int strange_update(int ind,
                   char val)
{
  if(ind>5 && ind <1000)
   A[ind] = val;
}
```

```
strange_update:
        leal    -6(%rdi), %eax
        cmpl    $993, %eax
        ja      .L13
        movslq  %edi,%rax
        movb    %sil, A(%rax)
.L13:
        rep
        ret
.
```

Vectorization I

```
static char A[1048576];
static char B[1048576];

void memcpy1()
{
  int i;
  for(i=0; i< 1048576; i++)
    B[i] = A[i];
}
```

```
memcpy1:
        xorl    %eax, %eax
.L2:
        movdqa  A(%rax), %xmm0
        movdqa  %xmm0, B(%rax)
        addq    $16,%rax
        cmpq    $1048576, %rax
        jne     .L2
        rep
        ret
```

Vectorization II

```
static char A[1048576];
static char B[1048576];

void memcpy2(int N)
{
  int i;
  for(i=0; i< N; i++)
    B[i] = A[i];
}
```

```
 memcpy2:
        testl   %edi, %edi
        jle     .L12
        movl    %edi, %esi
        shrl    $4, %esi
        movl    %esi, %edx
        sall    $4, %edx
        cmpl    $15, %edi
        ja      .L18
.L13:
        xorl    %ecx, %ecx
.L14:
        movslq  %ecx,%rdx
        addl    $1, %ecx
        movzbl  A(%rdx), %eax
        cmpl    %ecx,%edi
        movb    %al, B(%rdx)
        jg      .L14
.L12:
        rep
        ret
.L18:
        xorl    %ecx, %ecx
        xorl    %eax, %eax
        testl   %edx, %edx
        je      .L13
.L15:
        movdqa  A(%rax), %xmm0
        addl    $1, %ecx
        movdqa  %xmm0, B(%rax)
        addq    $16, %rax
        cmpl    %esi, %ecx
        jb      .L15
        cmpl    %edx, %edi
        movl    %edx, %ecx
        jne     .L14
        jmp     .L12
```

Vectorization III

```
static char A[1048576];
static char B[1048576];

void memcpy3(char X[],
             char Y[],
             int N)
{
  int i;
  for(i=0; i< N; i++)
   Y[i] = X[i];
}
```

```
memcpy3:
.LFB22:
        testl   %edx, %edx
        jle     .L28
        cmpl    $15, %edx
        ja      .L34
.L21:
        xorl    %ecx, %ecx
.L27:
        movzbl  (%rdi,%rcx), %eax
        movb    %al, (%rsi,%rcx)
        addq    $1, %rcx
        cmpl    %ecx, %edx
        jg      .L27
.L28:
        rep
        ret
.L34:
        testb   $15, %sil
        jne     .L21
        leaq    16(%rdi), %rax
        cmpq    %rax, %rsi
        jbe     .L35
.L29:
        movl    %edx, %r9d
        xorl    %ecx, %ecx
        xorl    %eax, %eax
        shrl    $4, %r9d
        xorl    %r8d, %r8d
        movl    %r9d, %r10d
        sall    $4, %r10d
        testl   %r10d, %r10d
        je      .L24
.L30:
        movdqu  (%rdi,%rax), %xmm0
        addl    $1, %ecx
        movdqa  %xmm0, (%rsi,%rax)
        addq    $16, %rax
        cmpl    %r9d, %ecx
        jb      .L30
        cmpl    %r10d, %edx
        movl    %r10d, %r8d
        je      .L28
.L24:
        movslq  %r8d,%rax
        leaq    (%rsi,%rax), %rcx
        addq    %rax, %rdi
        .p2align 4,,10
        .p2align 3
.L26:
        movzbl  (%rdi), %eax
        addl    $1, %r8d
        addq    $1, %rdi
        movb    %al, (%rcx)
        addq    $1, %rcx
        cmpl    %r8d, %edx
        jg      .L26
        rep
        ret
        .p2align 4,,10
        .p2align 3
.L35:
        leaq    16(%rsi), %rax
        cmpq    %rax, %rdi
        jbe     .L21
        jmp     .L29
```

Vectorization IV

```
static char A[1048576];
static char B[1048576];

void memcpy3(char X[],
             char Y[],
             int N)
{
  int i;
  for(i=0; i< N; i++)
   Y[i] = X[i];
}

void memcpy4()
{
  memcpy3(A, B, 1024);
}
```

```
memcpy4:
        xorl    %eax, %eax
.L37:
        movdqa  A(%rax), %xmm0
        movdqa  %xmm0, B(%rax)
        addq    $16, %rax
        cmpq    $1024, %rax
        jne     .L37
        rep
        ret
```

Vectorization V

```
static char A[1048576];

void memcpy3(char X[], char Y[],
                         int N)
{
  int i;
  for(i=0; i< N; i++)
    Y[i] = X[i];
}

void memcpy5()
{
  memcpy3(A+1, A, 1024);
}

void memcpy6()
{
  memcpy3(A, A+1, 1024);
}
```

```
memcpy5:
        movl    $A+1, %eax
.L41:
        movdqu  (%rax), %xmm0
        movdqa  %xmm0, -1(%rax)
        addq    $16, %rax
        cmpq    $A+1025, %rax
        jne     .L41
        rep
        ret


memcpy6:
       movzbl  A(%rip), %edx
        movl    $A+1, %eax
.L45:
        movb    %dl, (%rax)
        addq    $1, %rax
        cmpq    $A+1025, %rax
        jne     .L45
        rep
        ret
```

Vectorization VI

```
static char A[1048576];

void memcpy7()
{
  int i;
  for(i=1; i< 1025; i++)
    A[i] = A[0];
}

void memcpy8()
{
  int i;
  for(i=1; i< 1025; i++)
    A[i] = B[0];
}
```

```
memcpy7:
    movl    $A+1, %edx
.L49:
      movzbl  A(%rip), %eax
      movb    %al, (%rdx)
      addq    $1, %rdx
      cmpq    $A+1025, %rdx
      jne     .L49
      rep
      ret
```

```
memcpy8:
      movzbl  B(%rip), %edx
       pxor    %xmm0, %xmm0
       movl    $A+16, %eax
       movq    %rdx, -8(%rsp)
       movb    %dl, A+1(%rip)
       movq    -8(%rsp), %xmm1
       movb    %dl, A+2(%rip)
       movss   %xmm1, %xmm0
       movb    %dl, A+3(%rip)
       movb    %dl, A+4(%rip)
       movb    %dl, A+5(%rip)
       movb    %dl, A+6(%rip)
       punpcklbw       %xmm0, %xmm0
       movb    %dl, A+7(%rip)
       movb    %dl, A+8(%rip)
       movb    %dl, A+9(%rip)
       movb    %dl, A+10(%rip)
       movb    %dl, A+11(%rip)
       movb    %dl, A+12(%rip)
       punpcklbw       %xmm0, %xmm0
       movb    %dl, A+13(%rip)
       movb    %dl, A+14(%rip)
       movb    %dl, A+15(%rip)
       pshufd  $0, %xmm0, %xmm0
.L53:
       movdqa  %xmm0, (%rax)
       addq    $16, %rax
       cmpq    $A+1024, %rax
       jne     .L53
       movb    %dl, (%rax)
       ret
```

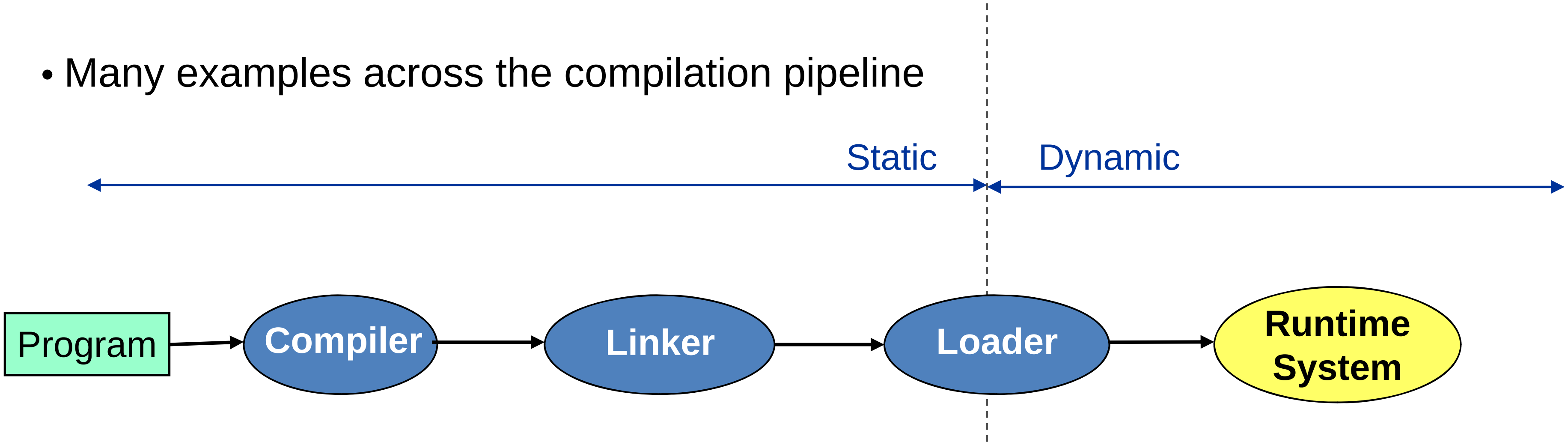
Tail Recursion

```
int fact(int x)
{
  if (x<= 0) return 1;
  return x*fact(x-1);
}
```

```
fact:
        testl   %edi, %edi
        movl    $1, %eax
        jg      .L4
        jmp     .L3
.L7:
        movl    %edx,%edi        # %edi gets X
.L4:
        leal    -1(%rdi), %edx   # X = X - 1
        imull   %edi, %eax       # fact *= X
        testl   %edx, %edx       # X ?
        jg      .L7
.L3:
        rep
        ret
```

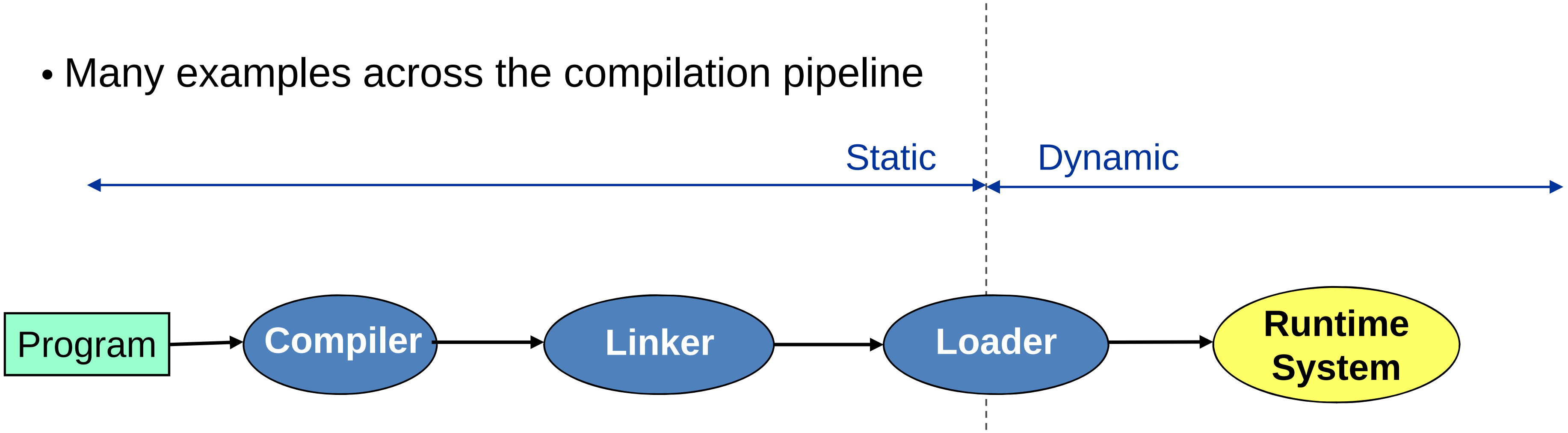
Optimization Continuum

- Many examples across the compilation pipeline



Optimization Continuum

- Many examples across the compilation pipeline

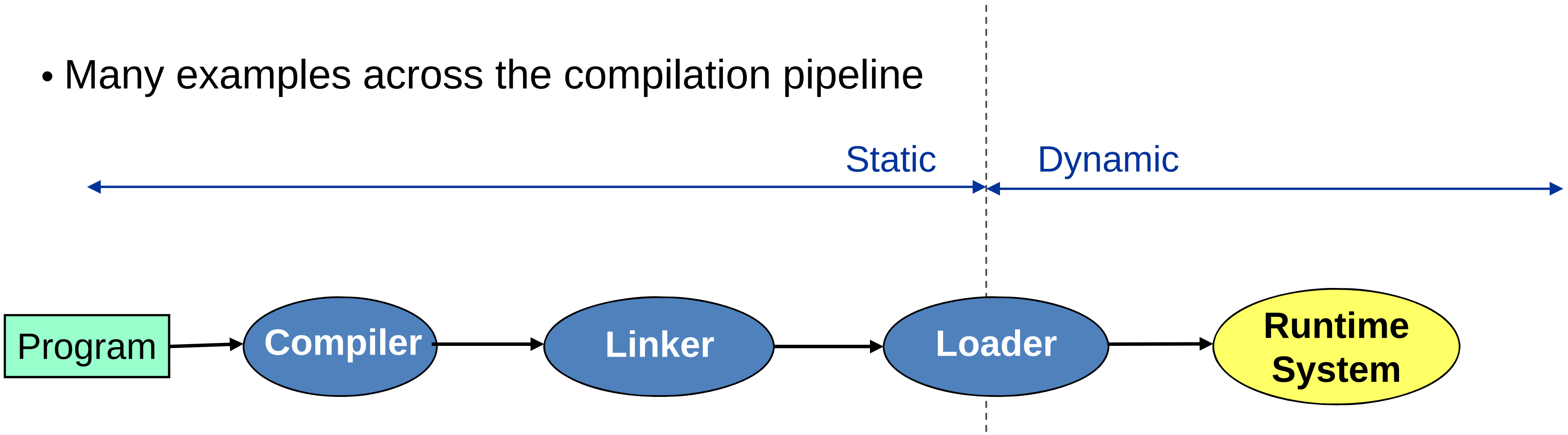


Compiler

- ➢ Pros: Full source code is available
- ➢ Pros: Easy to intercept in the high-level to low-level transformations
- ➢ Pros: Compile-time is not much of an issue
- ➢ Cons: Don't see the whole program
- ➢ Cons: Don't know the runtime conditions
- ➢ Cons: Don't know (too much about) the architecture

Optimization Continuum

- Many examples across the compilation pipeline

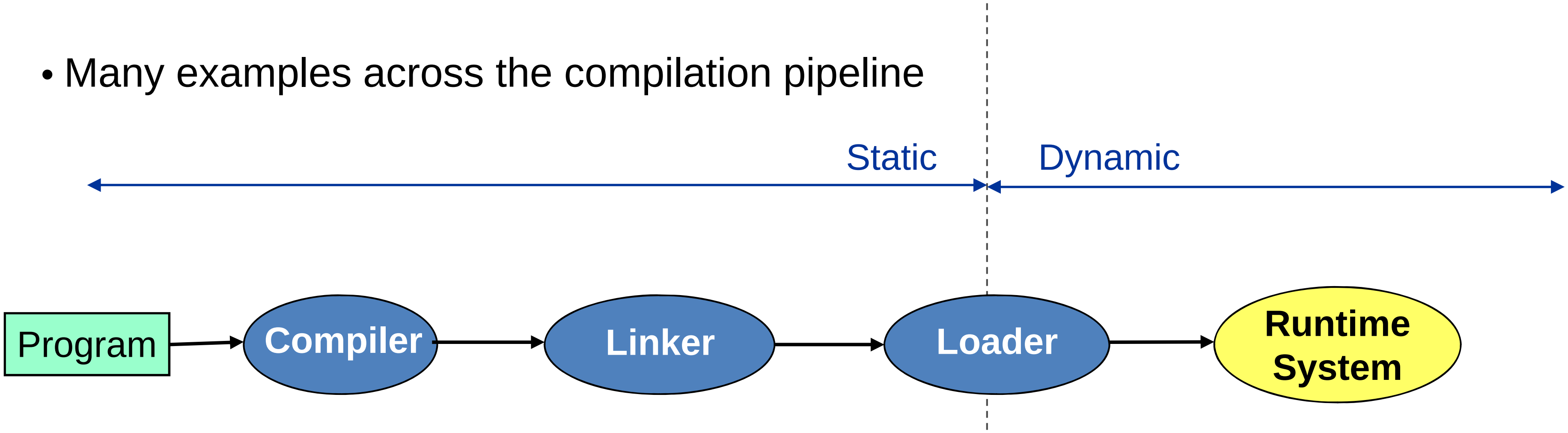


Linker

- ➢ Pros: Full program available
- ➢ Cons: May not have the full program…
- ➢ Cons: Don't have access to the source
- ➢ Cons: Don't know (too much about) the architecture

Optimization Continuum

- Many examples across the compilation pipeline

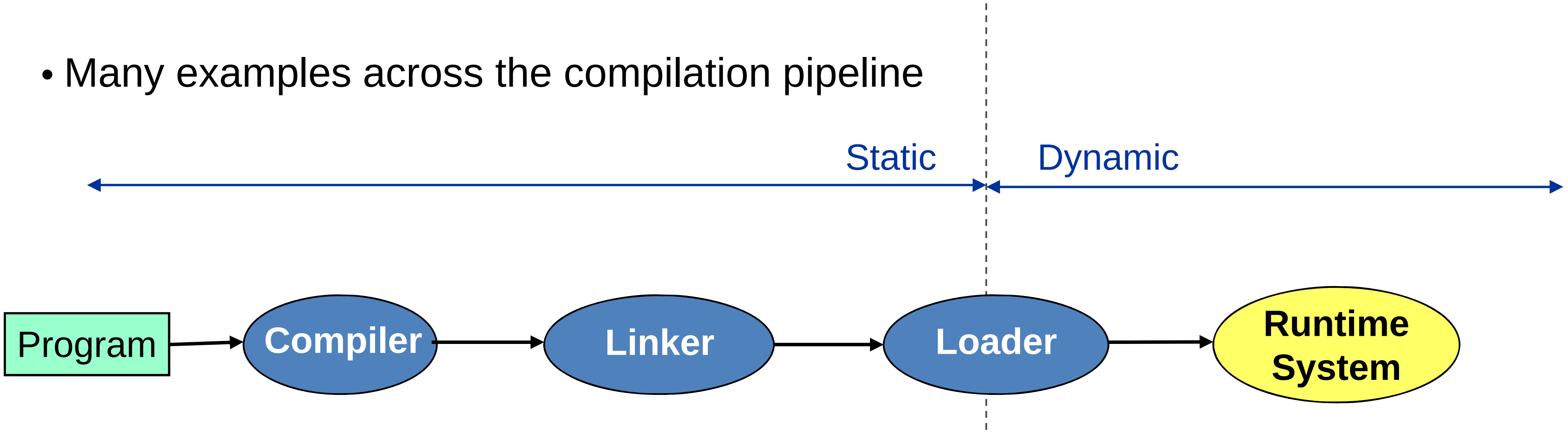


Loader

- ➢ Pros: Full program available
- ➢ (Cons: May not have the full program…)
- ➢ Cons: Don’t have access to the source
- ➢ Cons: Don’t know the runtime conditions
- ➢ Cons: Time pressure to get the loading done fast

Optimization Continuum

- Many examples across the compilation pipeline



Static

Dynamic

Program

Compiler

Linker

Loader

Runtime System

Runtime

- ➢ Pros: Full program available
- ➢ Pros: Knows the runtime behavior
- ➢ Cons: Don’t have access to the source
- ➢ Cons: Time in the optimizer is time away from running the program

Dataflow Analysis

Compile-Time Reasoning About

Run-Time Values of Variables or Expressions

At Different Program Points

- Which assignment statements produced value of variable at this point?
- Which variables contain values that are no longer used after this program point?
- What is the range of possible values of variable at this program point?

Example

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x + b*y;
    }
    return x;
```

```
sumcalc:
        pushq   %rbp
        movq    %rsp, %rbp
         movl    %edi, -20(%rbp)   # a
        movl    %esi, -24(%rbp)   # b
        movl    %edx, -28(%rbp)   # N

        movl    $0, -8(%rbp)      # x = 0
        movl    $0,  4(%rbp)      # y = 0
        movl    $0, -12(%rbp)     # i = 0
        jmp     .L2

.L3:    movl    -20(%rbp), %eax   # %eax <- a
        sall    $2, %eax          # %eax <- a * 4
        movl    %eax, -36(%rbp)
        movl    -36(%rbp), %edx
        movl    %edx, %eax        # %eax <- a*4
        sarl    $31, %edx
        idivl   -24(%rbp)         # %eax <- a*4/b
        movl    %eax, %ecx
        imull   -12(%rbp), %ecx   # %ecx <- (a*4/b)*i
        movl    -12(%rbp), %eax   # %eax <- i
        leal    1(%rax), %edx     # %edx <- i+1
        movl    -12(%rbp), %eax   # %eax <- i
        addl    $1, %eax          # %eax <- i+1
        imull   %edx, %eax        # %eax <- (i+1)*(i+1)
        leal    (%rcx,%rax), %eax # (i+1)*(i+1)+ (a*4/b)*i
        addl    %eax, -8(%rbp)
                                  # x = x + …
```

```
        movl    -24(%rbp), %eax    # %eax <- b
        imull   -4(%rbp), %eax     # %eax <- b*y
        addl    %eax, -8(%rbp)     # x = x + b*y

        addl    $1, -12(%rbp)      # i = i+1

.L2:
        movl    -12(%rbp), %eax    # %eax <  i
        cmpl    -28(%rbp), %eax   #  N ? i
        jle     .L3

        movl    -8(%rbp), %eax     # $eax <- x
        leave
        ret
```

Constant Propagation

In all possible execution paths a value of a variable at a given use point of that variable is a known constant.

- Replace the variable with the constant

Pros:

- No need to keep that value in a variable (freeing storage/register)
- Can lead to further optimization.

Constant Propagation

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x + b*y;
    }
    return x;
```

Constant Propagation

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x + b*y;
    }
    return x;
```

Constant Propagation

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x + b*y;
    }
    return x;
```

Constant Propagation

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x + b*y;
    }
    return x;
```

Constant Propagation

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
        x = x + (4*a/b)*i + (i+1)*(i+1);
        x = x + b*0;
    }
    return x;
}
```

Algebraic Simplification

If an expression can be calculated/simplified at compile time, then do it.

➢ Or use faster instructions to do the operation

Examples:

- X * 0 → 0
- X*1 → X
- X * 1024 → X << 10
- X + 0 → X
- X + X → X << 2

Pros:

➢ Less work at runtime

➢ Leads to more optimizations

Cons:

➢ Machine that runs the code may behave differently than the machine that is used to compile/compiler

- Ex: Overflow, underflow

➢ Use commutivity and transitivity can slightly change the results

Algebraic Simplification

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x + b*0;
    }
    return x;
```

Algebraic Simplification

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x + b*0;
    }
    return x;
```

Algebraic Simplification

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x + 0;
    }
    return x;
```

Algebraic Simplification

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x + 0;
    }
    return x;
```

Algebraic Simplification

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x + 0;
    }
    return x;
```

Algebraic Simplification

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x;
    }
    return x;
```

Copy Propagation

If you are just making a copy of a variable, try to use the original and eliminate the copy.

Pros:

- Less instructions
- Less memory/registers

Con:

- May make an "interference graph" no longer "colorable", leading to register spills

Copy Propagation

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x;
    }
    return x;
```

Copy Propagation

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);

    }
    return x;
```

Common Subexpression Elimination

Same subexpression is calculated multiple times → Calculate is once is use the result

Pros:

- Less computation

Cons:

- Need additional storage/register to keep the results. May lead to register spill.
- May hinder parallelization by adding dependences

Common Subexpression Elimination

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);

    }
    return x;
```

Common Subexpression Elimination

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
        x = x + (4*a/b)*i + (i+1)*(i+1);

    }
    return x;
```

Common Subexpression Elimination

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y, t;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + (4*a/b)*i + (i+1)*(i+1);
    }
    return x;
}
```

Common Subexpression Elimination

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y, t;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + (4*a/b)*i + t*t;
    }
    return x;
}
```

Dead Code Elimination

If the result of a calculation is not used, don’t do the calculation

Pros:

- ➢ Less computation
- ➢ May be able to release the storage earlier
- ➢ No need to store the results

Dead Code Elimination

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y, t;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       t = i+1;
       x = x + (4*a/b)*i + t*t;
    }
    return x;
}
```

Dead Code Elimination

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y, t;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       t = i+1;
       x = x + (4*a/b)*i + t*t;
    }
    return x;
}
```

Dead Code Elimination

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y, t;
    x = 0;

    for(i = 0; i <= N; i++) {
       t = i+1;
       x = x + (4*a/b)*i + t*t;
    }
    return x;
}
```

Dead Code Elimination

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t;
    x = 0;

    for(i = 0; i <= N; i++) {
       t = i+1;
       x = x + (4*a/b)*i + t*t;
    }
    return x;
}
```

Loop Invariant Removal

If an expression always calculate to the same value in all the loop iterations, move the calculation outside the loop

Pros:

- A lot less work within the loop

Cons

- Need to store the result from before loop starts and throughout the execution of the loop → more live ranges → may lead to register spills

Loop Invariant Removal

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t;
    x = 0;

    for(i = 0; i <= N; i++) {
       t = i+1;
       x = x + (4*a/b)*i + t*t;
    }
    return x;
}
```

Loop Invariant Removal

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t;
    x = 0;

    for(i = 0; i <= N; i++) {
       t = i+1;
       x = x + (4*a/b)*i + t*t;
    }
    return x;
}
```

Loop Invariant Removal

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u;
    x = 0;
    u = (4*a/b);
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + u*i + t*t;
    }
    return x;
```

Strength Reduction

In a loop, instead of recalculating an expression updated the previous value of the expression if that requires less computation.

Example

- for(i= 0 …) t=a*i; → t=0; for(i= 0 …) t=t+a;

Pros:

- Less computation

Cons:

- More values to keep → increase number of live variables → possible register spill
- Introduces a loop-carried dependence → a parallel loop become sequential
  - Strength increase transformation: Eliminate the loop carried dependence (but more instructions) by the inverse transformations to strength reduction.

Strength Reduction

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u;
    x = 0;
    u = (4*a/b);
    for(i = 0; i <= N; i++) {
       t = i+1;
       x = x + u*i + t*t;
    }
    return x;
```

Strength Reduction

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u;
    x = 0;
    u = (4*a/b);
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + u*i + t*t;
    }
    return x;
```

```
u*0,    v=0,
u*1,    v=v+u,
u*2,    v=v+u,
u*3,    v=v+u,
u*4,    v=v+u,
...     ...
```

Strength Reduction

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u, v;
    x = 0;
    u = (4*a/b);
    v = 0;
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + u*i + t*t;
        v = v + u;
    }
    return x;
}
```

Strength Reduction

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u, v;
    x = 0;
    u = (4*a/b);
    v = 0;
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + v + t*t;
        v = v + u;
    }
    return x;
}
```

Strength Reduction

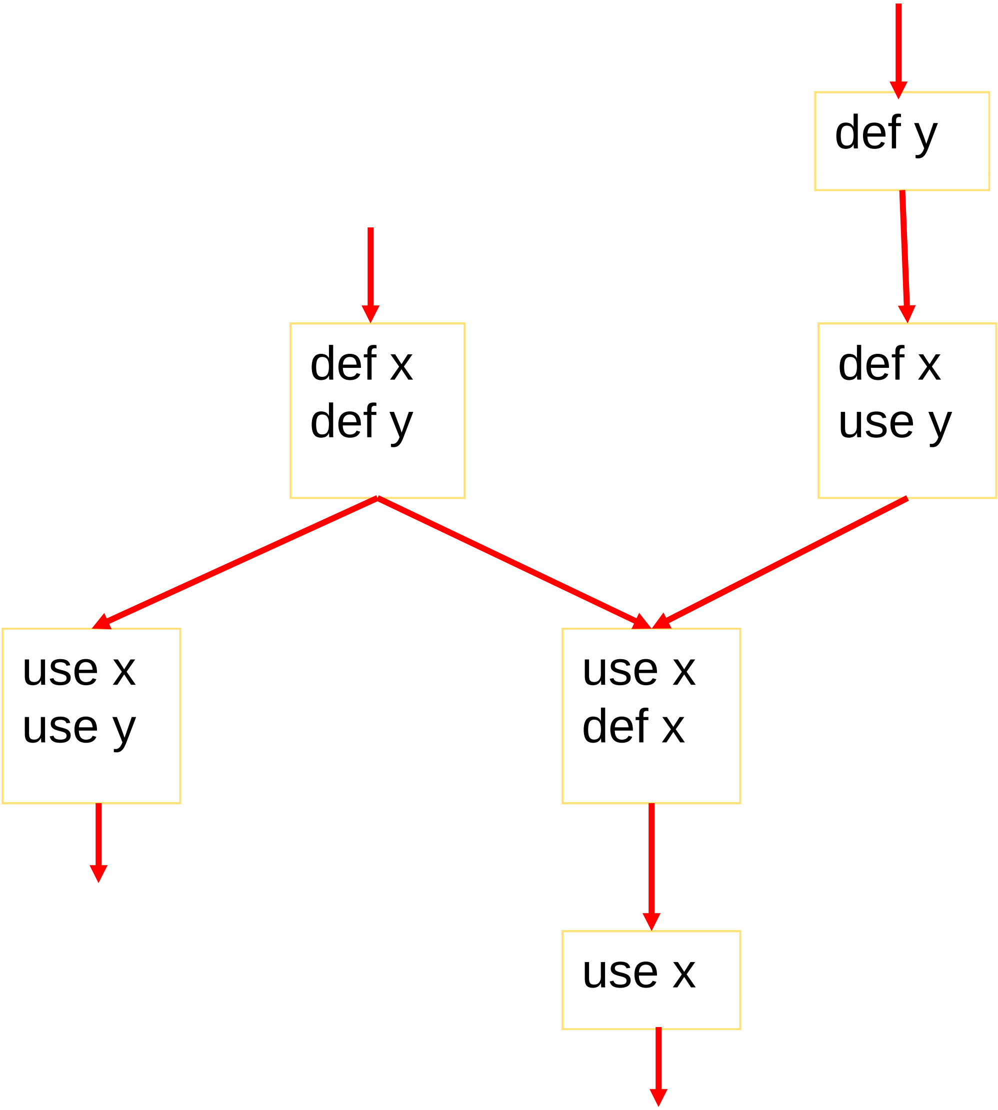
```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u, v;
    x = 0;
    u = (4*a/b);
    v = 0;
    for(i = 0; i <= N; i++) {
       t = i+1;
       x = x + v + t*t;
       v = v + u;
    }
    return x;
}
```

Register Allocation

Use the limited registers effectively to keep some of the live values instead of storing them in memory.

- In the x86 architecture, this is very important(that is why x86-64 have more registers) However, this is critical in the RISC architectures.

Example



def y

def x
def y

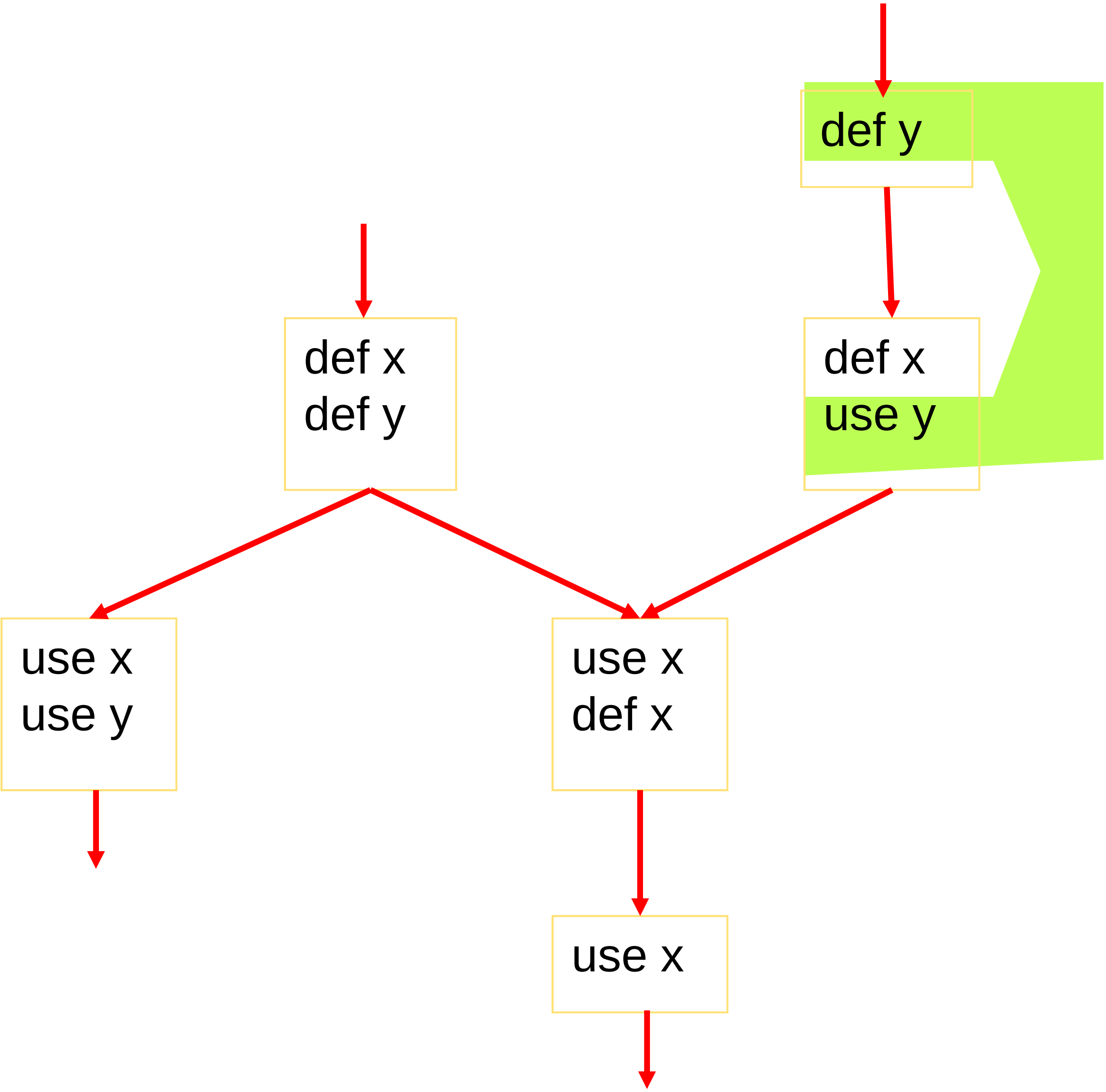
def x
use y

use x
use y

use x
def x

use x

Example


def y

def x
def y

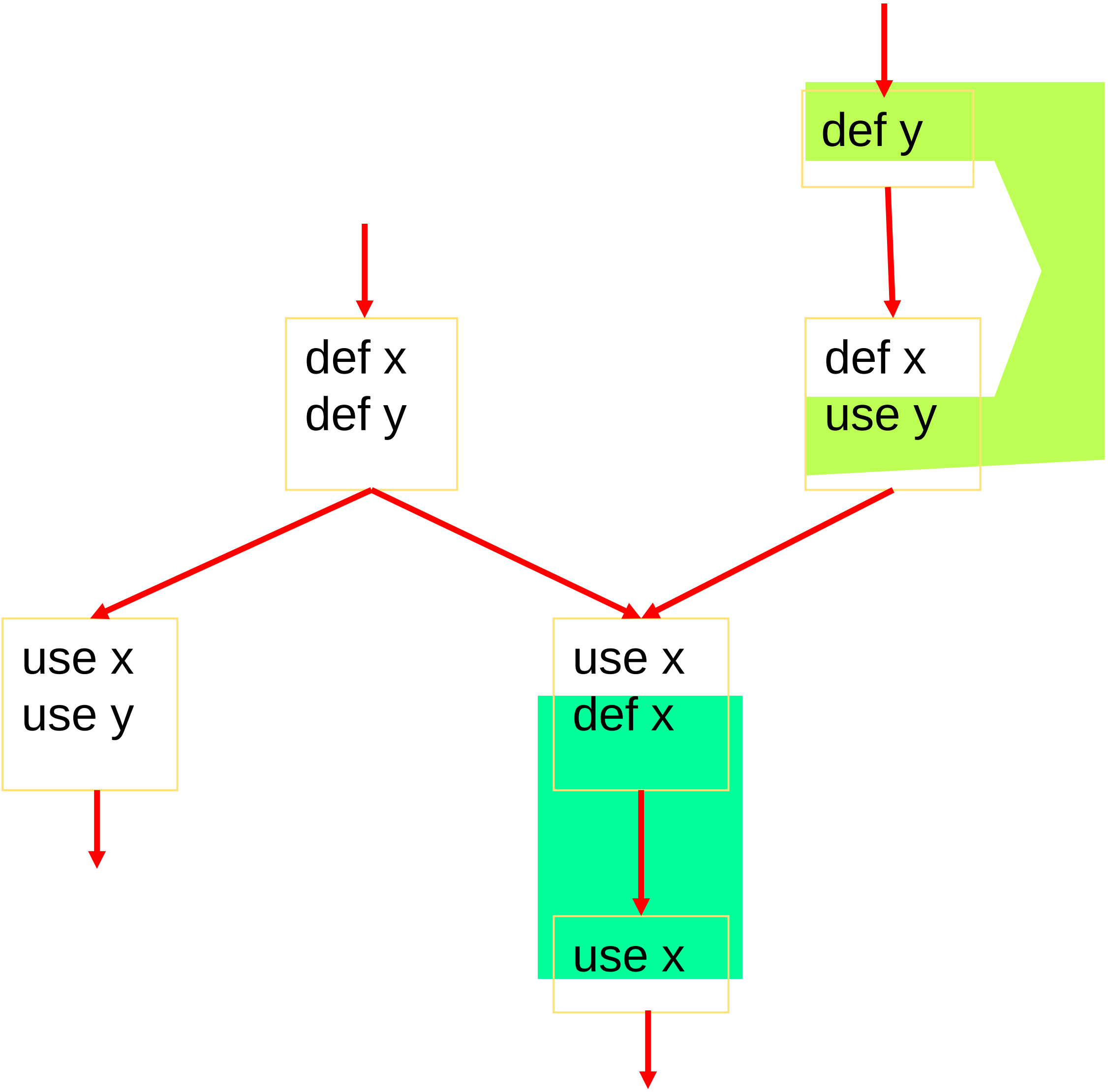
def x
use y

use x
use y

use x
def x

use x

Example



def y

def x
def y

def x
use y

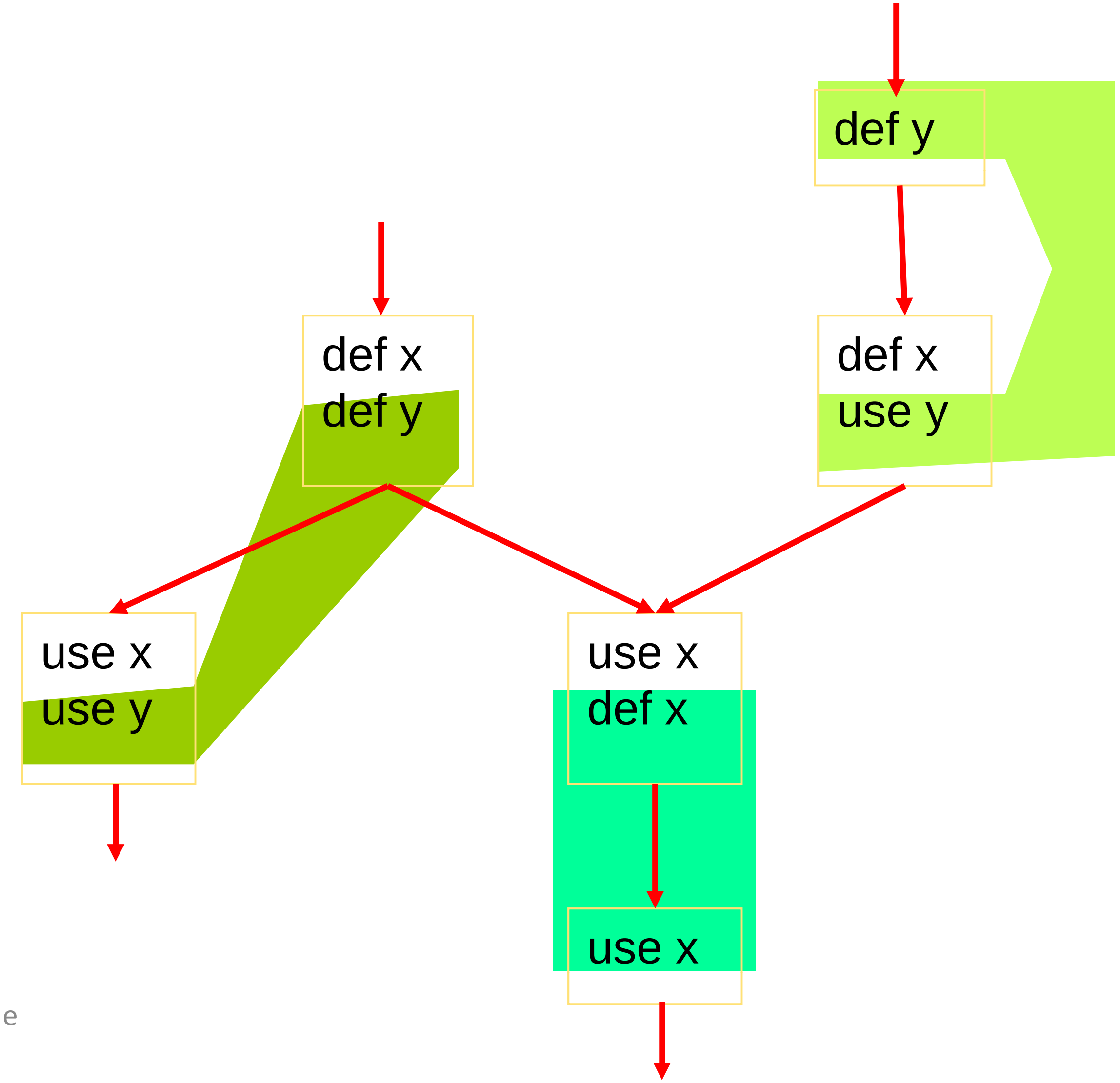
use x
use y

use x
def x

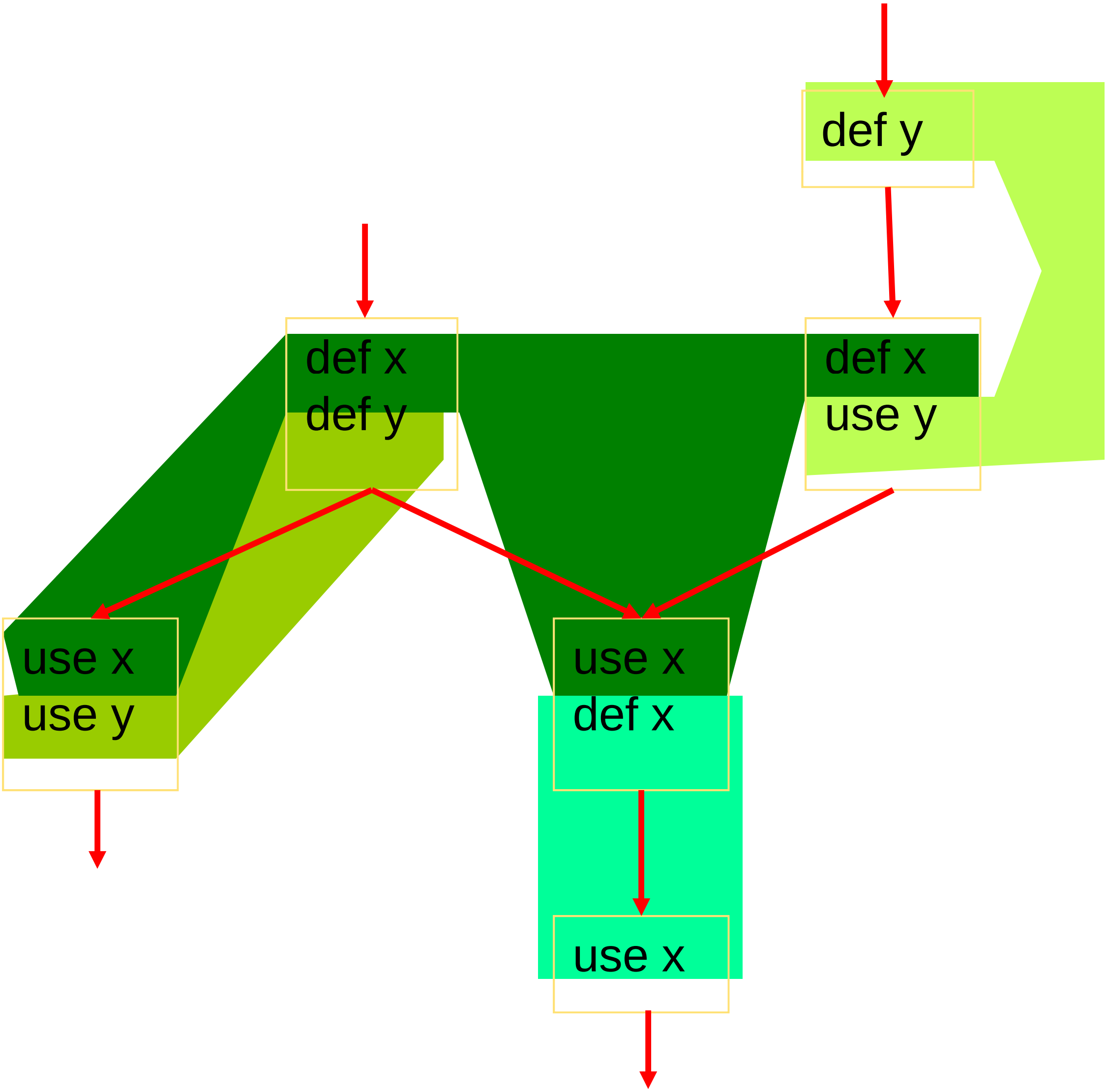
use x

Saman Amarasinghe

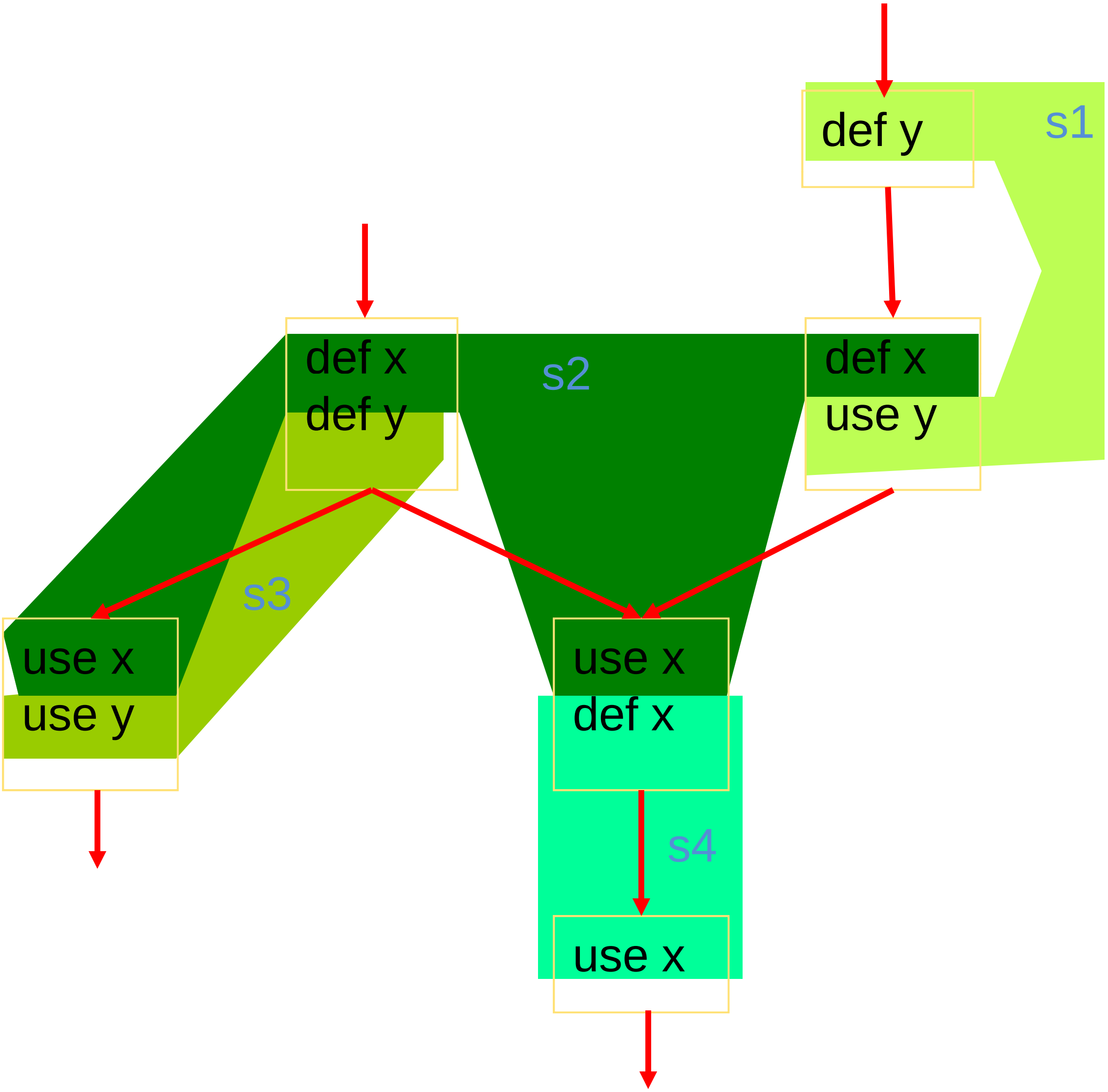
Example



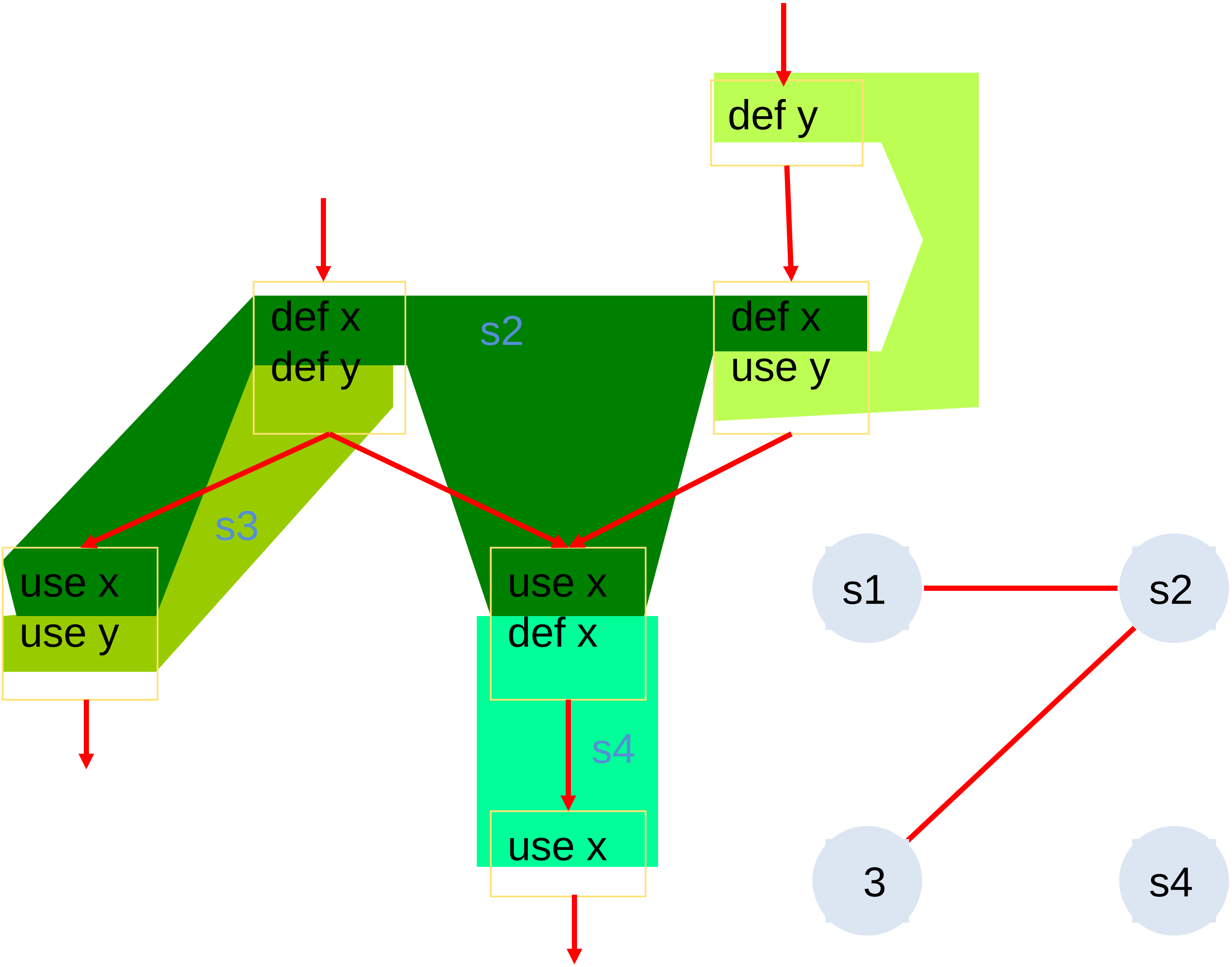
Example



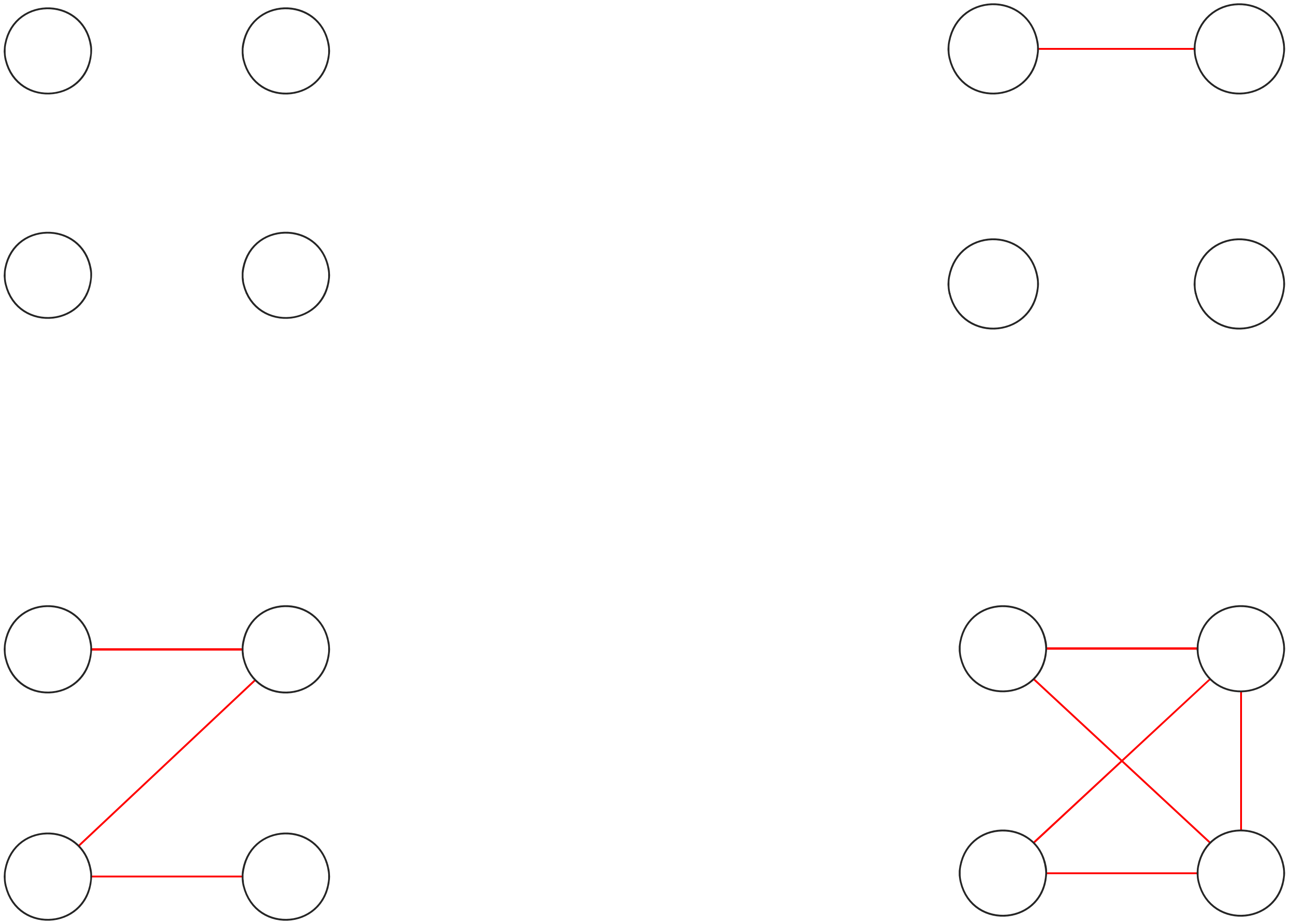
Example



Example


def y
def x
def y
s2
def x
use y
s3
use x
use y
use x
def x
s4
use x
s1
s2
3
s4

Graph Coloring



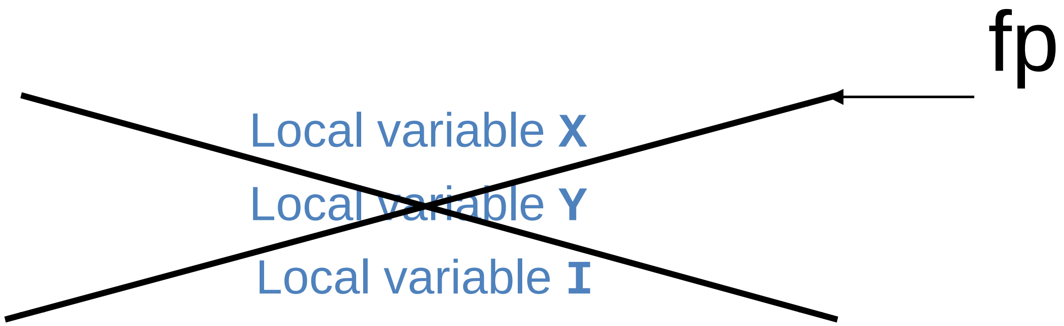
Register Allocation

If the graph can be colored with # of colors < # of registers → allocate a register for each variable

If too many colors

- Eliminate a edge → spill that register
- Recolor the graph

Register Allocation



fp

~~Local variable **X**~~

~~Local variable **Y**~~

~~Local variable **I**~~

```
$edi = X
$r8d = a
$esi = b
$r9d - N
$ecx = I
$r10d = t
```

Optimized Example

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u, v;
    x = 0;
    u = ((a<<2)/b);
    v = 0;
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + v + t*t;
        v = v + u;
    }
    return x;
}
```

```
sumcalc:
.LFB2:
        leal    0(,%rdi,4), %r8d  # %r8 <- 4*a
        movl    %edx, %r9d        # %r9 <- N
        xorl    %ecx, %ecx        # i = 0
        xorl    %edi, %edi        # x = 0
        jmp     .L2
.L3:
        movl    %r8d, %eax        # %eax <- 4*a
        leal    1(%rcx), %r10d    # %r10 <- i+1
        cltd                      # sign extend %eax
        idivl   %esi              # %eax <- 4*a/b
        imull   %eax, %ecx        # %ecx <- (4*a/b)*i
        movl    %r10d, %eax       # %eax <- i+1
        imull   %r10d, %eax       # %eax <- (i+1)*(i+1)
        leal    (%rcx,%rax), %eax # (4*a/b)*i+(i+1)*(i+1)
        movl    %r10d, %ecx       # i = i+1
        addl    %eax, %edi        # x = x + …
.L2:
        cmpl    %r9d, %ecx        # N ? i
        jle     .L3
        movl    %edi, %eax
        ret
```

Unoptimized Code

```
sumcalc:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movl    %edx, -28(%rbp)
        movl    $0, -8(%rbp)
        movl    $0, -4(%rbp)
        movl    $0, -12(%rbp)
        jmp     .L2
.L3:
        movl    -20(%rbp), %eax
        sall    $2, %eax
        movl    %eax, -36(%rbp)
        movl    -36(%rbp), %edx
        movl    %edx, %eax
        sarl    $31, %edx
        idivl   -24(%rbp)
        movl    %eax, %ecx
        imull   -12(%rbp), %ecx
        movl    -12(%rbp), %eax
        leal    1(%rax), %edx
        movl    -12(%rbp), %eax
        addl    $1, %eax
        imull   %edx, %eax
        leal    (%rcx,%rax), %eax
        addl    %eax, -8(%rbp)
        movl    -24(%rbp), %eax
        imull   -4(%rbp), %eax
        addl    %eax, -8(%rbp)
        addl    $1, -12(%rbp)
.L2:
        movl    -12(%rbp), %eax
        cmpl    -28(%rbp), %eax
        jle     .L3
        movl    -8(%rbp), %eax
        leave
        ret
```

Optimized Code

```
sumcalc:
        leal    0(,%rdi,4), %r8d
        movl    %edx, %r9d
        xorl    %ecx, %ecx
        xorl    %edi, %edi
        jmp     .L2
.L3:
        movl    %r8d, %eax
        leal    1(%rcx), %r10d
        cltd
        idivl   %esi
        imull   %eax, %ecx
        movl    %r10d, %eax
        imull   %r10d, %eax
        leal    (%rcx,%rax), %eax
        movl    %r10d, %ecx
        addl    %eax, %edi
.L2:
        cmpl    %r9d, %ecx
        jle     .L3
        movl    %edi, %eax
        ret
```

Execution time = 54.56 sec

Execution time = 8.19 sec

What Stops Optimizations?

Optimizer has to guarantee program equivalence

- For all the valid inputs
- For all the valid executions
- For all the valid architectures

In order to optimize the program, the compiler needs to understand

- The control-flow
- Data accessors

Most of the time, the full information is not available, then the compiler has to…

- Reduce the scope of the region that the transformations can apply
- Reduce the aggressiveness of the transformations
- Leave computations with unanalyzable data alone

Control-Flow

All the possible paths through the program

Representations within the compiler

- Call graphs
- Control-flow graphs

What hinders the compiler analysis?

- Function pointers
- Indirect branches
- Computed gotos
- Large switch statements
- Loops with exits and breaks
- Loops where the bound are not known
- Conditionals where the branch condition is not analyzable

Data-Accessors

Who else can read or write the data

Representations within the compiler

- Def-use chains
- Dependence vectors

What hinders the compiler analysis?

- Address taken variables
- Global variables
- Parameters
- Arrays
- Pointer accesses
- Volatile types

6.172 Performance Engineering of Software Systems
Fall 2010