

CS142B Language Processor Construction

SSA-based Compiler Optimizations

Yeoul Na

UCI

April 30, 2019

Redundancy – Source of Optimization

- Convenience for programming

```
foo(a + b);      a = 4; b = 10;
```

```
. . .           . . .
```

```
c = a + b;      c = a + b;
```

- A side effect of using a high-level language

```
b = a[i];
```

```
. . .
```

```
a[i] = x;
```

```
t1 = i*4
```

```
t2 = a + t1
```

```
b = *(t2)
```

```
. . .
```

```
t3 = i*4
```

```
t4 = a + t3
```

```
*(t4) = x
```

Optimizations to Remove Redundancy

- Common Sub-Expression Elimination
- Constant Propagation
- Copy Propagation
- Dead-Code Elimination

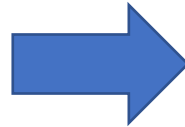
Common Sub-Expression Elimination

```
t1 = i*4  
t2 = a + t1  
b = *(t2)
```

...

```
t3 = i*4  
t4 = a + t3  
*(t4) = x
```

If *i* nor *a* has
not changed



```
t1 = i*4  
t2 = a + t1  
b = *(t2)
```

...

```
*(t2) = x
```

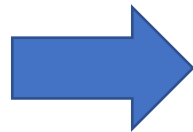
Constant Propagation

- If the result of an expression is known at compile time, we don't need to execute it at run time.

```
a = 4; b = 10;
```

```
...
```

```
c = a + b;
```



```
a = 4; b = 10;
```

```
...
```

```
c = 14;
```

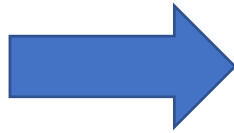
Dead-Code Elimination

- May appear as the result of previous transformations

```
a = 4; b = 10;
```

```
...
```

```
c = 14;
```



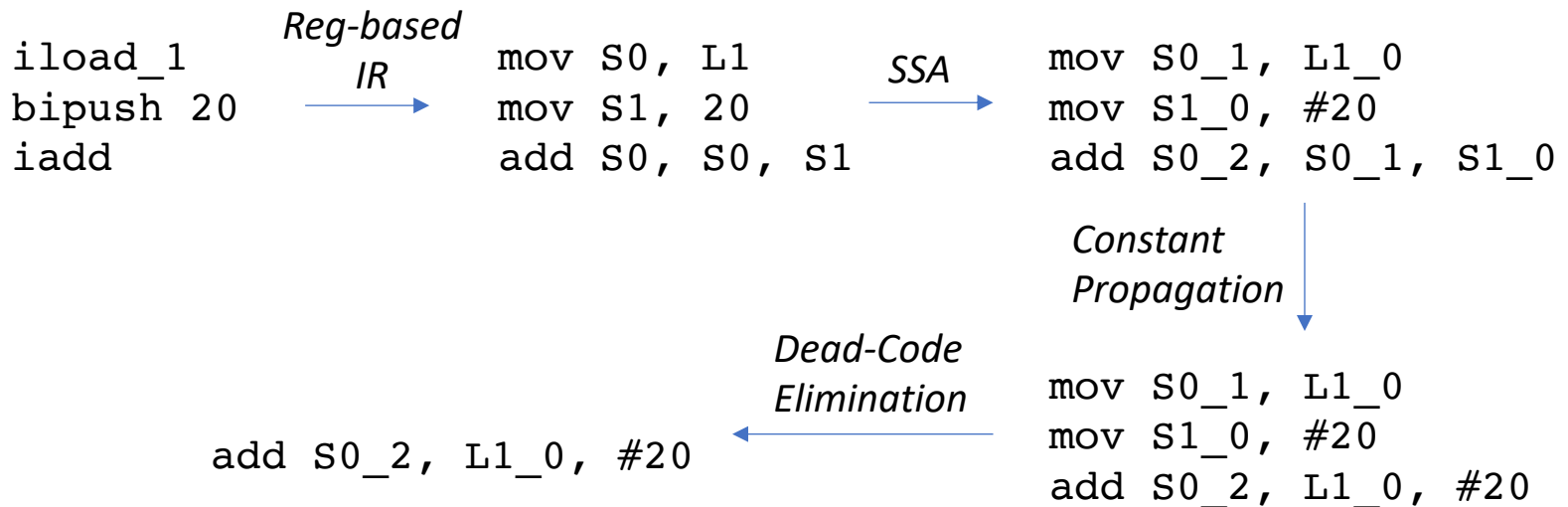
```
...
```

```
c = 14;
```

Assuming `c` is used
in the program
afterwards

Copy Propagation

- May appear as the result of previous transformations
- Our case: after converting stack-based bytecode to register-based IR



Deep Dive - Constant Propagation

- Data-flow analysis
 - After each statement, maintain a set C of values for each variable
 - C contains either a constant value v , or a symbol \perp (bottom), or a symbol \top (top)
- Meanings of the values
 - v : the variable must have this constant value during execution
 - \top : the initial (known) value of each variable
 - \perp : the variable may have different values during execution

Constant Propagation Analysis

L1: a = 3;

C1 = {T}

L2: b = 5;

C2 = {T}

L3: d = 4;

C3 = {T}

L4: x = 100;

C4 = {T}

L5: if a > b then

C5 = {T}

L6: c = a + b;

C6 = {T}

L7: d = 2;

C7 = {T}

L8: endif

C8 = {T}

L9: c = 4;

C9 = {T}

L10: return b * d + c;

C10 = {T}

Constant Propagation Analysis


L1: a = 3;	value _a [L1] = {3}	C1 = {3}
L2: b = 5;	value _b [L1] = {T}	C2 = {T}
L3: d = 4;	value _d [L1] = {T}	C3 = {T}
L4: x = 100;	value _x [L1] = {T}	C4 = {T}
L5: if a > b then	value _c [L1] = {T}	C5 = {T}
L6: c = a + b;		C6 = {T}
L7: d = 2;		C7 = {T}
L8: endif		C8 = {T}
L9: c = 4;		C9 = {T}
L10: return b * d + c;		C10 = {T}

Constant Propagation Analysis

L1: a = 3;	value _a [L2] = {3}	C1 = {3}
L2: b = 5;	value _b [L2] = {5}	C2 = {5}
L3: d = 4;	value _d [L2] = {T}	C3 = {T}
L4: x = 100;	value _x [L2] = {T}	C4 = {T}
L5: if a > b then	value _c [L2] = {T}	C5 = {T}
L6: c = a + b;		C6 = {T}
L7: d = 2;		C7 = {T}
L8: endif		C8 = {T}
L9: c = 4;		C9 = {T}
L10: return b * d + c;		C10 = {T}

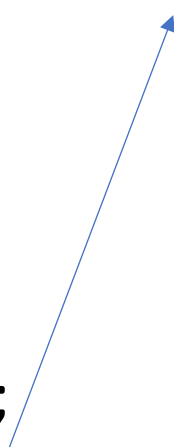
Constant Propagation Analysis

L1: a = 3;	value _a [L6] = {3}	C1 = {3}
L2: b = 5;	value _b [L6] = {5}	C2 = {5}
L3: d = 4;	value _d [L6] = {4}	C3 = {4}
L4: x = 100;	value _x [L6] = {100}	C4 = {100}
L5: if a > b then	value _c [L6] = {8}	C5 = {T}
L6: c = a + b;		C6 = {8}
L7: d = 2;		C7 = {T}
L8: endif		C8 = {T}
L9: c = 4;		C9 = {T}
L10: return b * d + c;		C10 = {T}



Constant Propagation Analysis

L1: a = 3;	$value_a[L10] = \{3\}$	$C1 = \{3\}$
L2: b = 5;	$Value_b[L10] = \{5\}$	$C2 = \{5\}$
L3: d = 4;	$Value_d[L10] = \{\perp\}$	$C3 = \{4\}$
L4: x = 100;	$value_x[L10] = \{100\}$	$C4 = \{100\}$
L5: if a > b then	$value_c[L10] = \{4\}$	$C5 = \{T\}$
L6: c = a + b;		$C6 = \{8\}$
L7: d = 2;		$C7 = \{2\}$
L8: endif		$C8 = \{T\}$
L9: c = 4;		$C9 = \{4\}$
L10: return b * d + c;		$C10 = \{\perp\}$



$$value_d[s] = \text{join}_{s1, s2 \in \text{prec}(s)} (value_d[s1], value_d[s2])$$

Data Flow Analysis Schema

- Calculate Data-flow values before and after each statement s : $IN[s], OUT[s]$. $IN[s_{i+1}] = OUT[s_i]$
- Transfer Function
 - Forward : $OUT[s] = f_s(IN[s])$
 - Backward : $IN[s] = f_s(OUT[s])$
 - E.g. Constant Propagation
 - $C[s: a = b + c] = value_a[s]$
 - $value_a[s] = \{value_b[s] + value_c[s]\}$, if they are both values
 $\{\perp\}$, otherwise
- Join (Meet)
 - $IN[B] = \bigwedge_{P: Pred\ of\ B} OUT[P]$

(Classical) Data-Flow Analysis

- A static technique for gathering information about the possible set of values calculated at every possible point in a program
- Performed on top of a control-flow graph
- Evaluating at every statement -> Inefficient!

Sparse Data-Flow Analysis

- Data-flow information can be propagated more efficiently using a sparse representation of the program such as SSA.
 - For certain data-flow problems definition points are exactly the set of program points where data-flow values may change
 - Associate data-flow values directly with variable names
 - Rather than maintaining a vector of data-flow values indexed over all variables, at each program point

Sparse Constant Propagation (Eg.)

L1: a0 = 3;
L2: b0 = 5;
Def(d0) L3: d0 = 4;
L4: x0 = 100;
L5: if a0 > b0 then
L6: c0 = a0 + b0;
L7: d1 = 2;
L8: endif
Use(d0) d2 = φ (d0, d1)
L9: c1 = 4;
L10: return b0 * d2 + c1;

L1: a0 = 3;
L2: b0 = 5;
L3: d0 = 4;
L4: x0 = 100;
L5: if a0 > b0 then
L6: c0 = a0 + b0;
L7: d1 = 2;
L8: endif
 d2 = φ (**4**, d1)
L9: c1 = 4;
L10: return b0 * d2 + c1;

Sparse Constant Propagation Algorithm

```
worklist = all statements in SSA
while worklist  $\neq \emptyset$ 
  remove S=<e,d> from worklist
  c = constant_fold(S)
  if c  $\neq$  null
    for each statement T that uses d
      substitute d with c in T
      worklist = worklist union {T}
    end
  delete S from program
end
```

```
procedure constant_fold(S):
  if S is d = phi(c,c,c,...)
    return c
  if S is d = mov(c)
    return c
  if S is d = a + b
    if con(a) and con(b)
      return a + b
    else
      return null
  // handle other operations too

return null
```

Sparse Constant Propagation (Eg.)

Def(a0) L1: **a0** = 3;

L2: b0 = 5;

L3: d0 = 4;

L4: x0 = 100;

Use(a0) L5: if **a0** > b0 then

Use(a0) L6: c0 = **a0** + b0;

L7: d1 = 2;

L8: endif

d2 = φ (d0, d1)

L9: c1 = 4;

L10: return b0 * d2 + c1;

L1:

L2: b0 = 5;

L3: d0 = 4;

L4: x0 = 100;

L5: if 3 > b0 then

L6: c0 = 3 + b0;

L7: d1 = 2;

L8: endif

d2 = φ (d0, d1)

L9: c1 = 4;

L10: return b0 * d2 + c1;

Sparse Constant Propagation (Eg.)

```
L1:  
Def(b0) L2: b0 = 5;  
L3: d0 = 4;  
L4: x0 = 100;  
Use(b0) L5: if 3 > b0 then  
Use(b0) L6:   c0 = 3 + b0;  
L7:   d1 = 2;  
L8:   endif  
       d2 =  $\varphi$  (d0, d1)  
L9: c1 = 4;  
Use(b0) L10: return b0 * d2 + c1;
```

```
L1:  
L2:  
L3: d0 = 4;  
L4: x0 = 100;  
L5: if 3 > 5 then  
L6:   c0 = 3 + 5;  
L7:   d1 = 2;  
L8:   endif  
       d2 =  $\varphi$  (d0, d1)  
L9: c1 = 4;  
L10: return 5 * d2 + c1;
```

Sparse Constant Propagation (Eg.)

```
L1:
L2:
Def(d0) L3: d0 = 4;
L4: x0 = 100;
L5: if 3 > 5 then
L6:   c0 = 3 + 5;
L7:   d1 = 2;
L8: endif
Use(d0)   d2 =  $\varphi$  (d0, d1)
L9: c1 = 4;
L10: return 5 * d2 + c1;
```

```
L1:
L2:
L3:
L4: x0 = 100;
L5: if 3 > 5 then
L6:   c0 = 3 + 5;
L7:   d1 = 2;
L8: endif
          d2 =  $\varphi$  (4, d1)
L9: c1 = 4;
L10: return 5 * d2 + c1;
```

Sparse Constant Propagation (Eg.)

```
L1:  
L2:  
L3:  
L4:  
L5: if 3 > 5 then  
Def(c0) L6:   c0 = 3 + 5;  
L7:   d1 = 2;  
L8: endif  
      d2 =  $\varphi$  (4, d1)  
L9: c1 = 4;  
L10: return 5 * d2 + c1;
```

```
L1:  
L2:  
L3:  
L4:  
L5: if 3 > 5 then  
L6:  
L7:   d1 = 2;  
L8: endif  
      d2 =  $\varphi$  (4, d1)  
L9: c1 = 4;  
L10: return 5 * d2 + c1;
```

Sparse Constant Propagation (Eg.)

L1

L2:

L3:

L4:

L5: if 3 > 5 then

L6:

Def(d1) L7: d1 = 2;

L8: endif

Use(d1) d2 = φ (4, d1)

L9: c1 = 4;

L10: return 5 * d2 + c1;

L1:

L2:

L3:

L4:

L5: if 3 > 5 then

L6:

L7:

L8: endif

d2 = φ (4, 2)

L9: c1 = 4;

L10: return 5 * d2 + c1;

Sparse Constant Propagation (Eg.)

L1

L2:

L3:

L4:

L5: if 3 > 5 then

L6:

L7: d1 = 2;

L8: endif

 d2 = φ (4, 2)

Def(c1) L9: c1 = 4;

Use(c1) L10: return 5 * d2 + c1;

L1:

L2:

L3:

L4:

L5: if 3 > 5 then

L6:

L7:

L8: endif

 d2 = φ (4, 2)

L9:

L10: return 5 * d2 + 4;

Sparse Conditional Constant Propagation

- We can further optimize the code!

Entry:

L5: if 3 > 5 then **FALSE!**

L8: endif

$d2 = \varphi(4, 2)$

L10: return 5 * d2 + 4;

Entry:

L5: if 3 > 5 then **FALSE!**

L8: endif

$d2 = 4$

L10: return 5 * **4** + 4;

Sparse Conditional Constant Propagation

- We can further optimize the code!

Entry:

L5: if 3 > 5 then **FALSE!**

L8: endif

$d2 = \varphi(4, 2)$

L10: return 5 * d2 + 4;

Entry:

L10: return **24**;

“Constant propagation with conditional branches,” Mark N. Wegman and F. Kenneth Zadeck, ACM TOPLAS 1991.

Copy Propagation Algorithm

```
worklist = all statements in SSA
while worklist  $\neq \emptyset$ 
  remove statement S from worklist
  if S is  $x = \text{phi}(y)$  or  $x = y$ 
    for each statement T that uses x
      replace all use of x with y
      worklist = worklist union {T}
    end
  delete S from program
end
```

Project Tips

- Construct def-use chains during SSA renaming
 - Eg. Add a list of `Uses{Instruction, Index}`
 - `Inst (L2)` has `Uses[0] = {Inst(L3), 1}`, as its field

L1: `a = 3;`

L2: `b = 5;`

L3: `c = a + b;`