

Optimizing Transformations

Nikhil Hegde

Compiler Optimizations course @ QUALCOMM India Pvt. Ltd.

Transformations

- Constant folding

```
area = (22.0/7.0)*pow(r,2)
```

- Constant propagation

```
a = 1.23;
```

```
...
```

```
b = a*7.66;
```

- CSE

```
a = b*c;
```

```
...
```

```
x = b*c/2.0
```

- Copy propagation

```
a = b;
```

```
..
```

```
x = a + w;
```

```
y = (b+w)/3.4
```

- Code movement
- Loop optimizations
- Strength reduction
- Dead code elimination
- ..many more.

All require rewriting code

Liveness – Recap..

X defined here

1: $X = 10$

.....

N: $Y = X + 5$

X used here

X is live at 1

..used in future

- A variable X is live at statement S if:
 - There is a statement S' that uses X
 - There is a path from S to S'
 - There are no intervening definitions of X

Liveness – Recap..

1: $X = 10$ X is dead at 1

2: $X = Y + 2$

...

N: $Y = X + 5$

- A variable X is dead at statement S if it is not live at S
 - What about $\dots; X = X + 1$?

Common subexpression elimination

- Goal: remove redundant computation, don't calculate the same expression multiple times

1: $A = B * C$

2: $E = B * C$

Keep the result of statement 1 in a temporary and reuse for statement 2

- Difficulty: how do we know when the same expression will produce the same result?

1: $A = B * C$


2: $B = \text{<new value>}$

3: $E = B * C$

B is “killed.” Any expression using B is no longer “available,” so we cannot reuse the result of statement 1 for statement 3

- This becomes harder with pointers (how do we know when B is killed?)

Common subexpression elimination

- Two varieties of common subexpression elimination (CSE)
- Local: within a single basic block  Maximal sequence of instructions that are executed one after another (i.e. there are no jump instructions OR no instruction is the target of a jump)
 - Easier problem to solve (why?)
- Global: within a single procedure or across the whole program
 - Intra- vs. inter-procedural
 - More powerful, but harder (why?)

Local optimizations are done on basic blocks. Global optimizations on control flow graphs (CFGs), where the basic blocks are the nodes of the graph. Then, there are inter-procedural optimizations, which span function calls. Later on CFGs and other kinds of optimizations.

CSE in practice

- Idea: keep track of which expressions are “available” during the execution of a basic block
 - Which expressions have we already computed?
 - Issue: determining when an expression is no longer available
 - This happens when one of its components is assigned to, or “killed.”
- Idea: when we see an expression that is already available, rather than generating code, copy the temporary
 - Issue: determining when two expressions are the same

Maintaining available expressions

- For each 3AC operation in a basic block
 - Create name for expression (based on lexical representation)
 - If name not in available expression set, generate code, add it to set
 - Track register that holds result of and any variables used to compute expression
 - If name in available expression set, generate move instruction
 - If operation assigns to a variable, kill all dependent expressions

Example

3 Address Code	Available expression(s)	Killed expression(s)
ADD A B T1	{ }	
ADD T1 C T2	{ "A + B" }	
ADD A B T3	{ "A + B", "T1 + C" }	
ADD T1 T2 C	{ "A + B", "T1 + C" }	{ "T1+C" }
ADD T1 C T4	{ "A + B", "T1 + T2" }	
ADD T3 T2 D	{ "A + B", "T1 + T2", "T1 + C" }	
	{ "A + B", "T1 + T2", "T1 + C", "T3 + T2" }	

Downsides (CSE)

- What are some downsides to this approach? Consider the two highlighted operations

Three address code

```
+ A B T1
+ T1 C T2
+ A B T3
+ T1 T2 C
+ T1 C T4
+ T3 T2 D
```

Generated code

```
ADD A B R1
ADD R1 C R2
MOV R1 R3
ADD R1 R2 R5; ST R5 C
ADD R1 C R4
ST R5 D
```

T1 and T3 compute the same expression. This can be handled by an optimization called *value numbering*.

Aliasing

- One of the biggest problems in compiler analysis is to recognize aliases – different names for the same location in memory

exercise: are T1 and T3 aliased in previous example?

- Why do aliases occur?
 - Pointers referring to the same location
 - Function calls passing the same reference in two arguments
 - Arrays referencing the same element
 - Unions
- What problems does aliasing pose for CSE?
 - when talking about “live” and “killed” values in optimizations like CSE, we’re talking about particular variable names
 - In the presence of aliasing, we may not know which variables get killed when a location is written to

Memory disambiguation

- Most compiler analyses rely on *memory disambiguation*
 - Otherwise, they need to be too conservative and are not useful
- Memory disambiguation is the problem of determining whether two references point to the same memory location
 - *Points-to* and *alias* analyses try to solve this
 - Will cover basic pointer analyses in a later lecture

Available Expressions

- **Goal:** determine a set of expressions that have already been computed.
 - E.g. to perform global CSE
- **Direction of the analysis:**
 - How does information flow w.r.t. control flow?
- **Join operator:**
 - What happens at merge points? E.g. what operator to use Union or Intersection?
- **Transfer function:**
 - Define sets AvailIn(b), AvailOut(b), Compute(b), Kill(b)
- **Initializations?**

Transfer functions for meet

- What do the transfer functions look like if we are doing a meet?

$$\begin{aligned} IN(S) &= \bigcap_{t \in pred(s)} OUT(t) \\ OUT(S) &= \mathbf{gen}(s) \cup (IN(S) - \mathbf{kill}(s)) \end{aligned}$$

- $\mathbf{gen}(s)$: expressions that *must be* computed in this statement
- $\mathbf{kill}(s)$: expressions that use variables that *may* be defined in this statement
 - Note difference between these sets and the sets for reaching definitions or liveness
- Insight: \mathbf{gen} and \mathbf{kill} must never lead to incorrect results
 - Must not decide an expression is available when it isn't, but OK to be safe and say it isn't
 - Must not decide a definition *doesn't* reach, but OK to overestimate and say it does

Analysis initialization

- How do we initialize the sets?
 - If we start with everything initialized to \perp , we compute the smallest sets
 - If we start with everything initialized to \top , we compute the largest
- Which do we want? It depends!
 - Reaching definitions: a definition that *may* reach this point
 - We want to have as few reaching definitions as possible $\rightarrow \perp$
 - Available expressions: an expression that *was definitely* computed earlier
 - We want to have as many available expressions as possible $\rightarrow \top$
 - Rule of thumb: if confluence operator is \sqcup , start with \perp , otherwise start with \top

Intermediate code (assuming int is 4 bytes):
 (Ignore the temporary counter value for now)

void quicksort(int m, int n) available expression

<pre> { int i, j; int v, x; if (n <= m) return; /* fragment begins here */ i = m-1; j = n; v = a[n]; while (1) { do i = i+1; while (a[i] < v); do j = j-1; while (a[j] > v); if (i >= j) break; x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */ } x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */ /* fragment ends here */ quicksort(m,j); quicksort(i+1,n); } </pre>	<pre> {} {"4*i"} S₁={"4*i", "a+t6"} set S₁ S₂=S₁ U {"4*j"} S₃=S₂ U {"a+t8"} set S₃ set S₃ set S₃ </pre>	<pre> t6 = 4*i x = a[t6] t7 = 4*i t8 = 4*j t9 = a[t8] a[t7] = t9 t10 = 4*j a[t10] = x </pre>	<p><u>Can be rewritten:</u></p> <p>t7 = t6</p> <p>a[t6] = t9</p> <p>t10 = t8</p> <p>a[t8] = x</p> <p>copy propagation</p>
---	--	--	---

Intermediate code (assuming int is 4 bytes):
 (Ignore the temporary counter value for now)

<code>void quicksort(int m, int n)</code>	<u>available expression</u>	
<code>{</code>	$\{$	<code>t6 = 4*i</code>
	$\{ "4*i" \}$	<code>x = a[t6]</code> <u>apply dead-code elim.</u>
<code>int i, j;</code>	$S_1 = \{ "4*i", "a+t6" \}$	<code>t7 = 4*i</code> $t7 = t6$
<code>int v, x;</code>	set S_1	<code>t8 = 4*j</code>
<code>if (n <= m) return;</code>	$S_2 = S_1 \cup \{ "4*j" \}$	<code>t9 = a[t8]</code>
<code>/* fragment begins here */</code>	$S_3 = S_2 \cup \{ "a+t8" \}$	<code>a[t7] = t9</code> $a[t6] = t9$
<code>i = m-1; j = n; v = a[n];</code>	set S_3	<code>t10 = 4*j</code> $t10 = t8$
<code>while (1) {</code>	set S_3	<code>a[t10] = x</code> $a[t8] = x$
<code>do i = i+1; while (a[i] < v);</code>		
<code>do j = j-1; while (a[j] > v);</code>		
<code>if (i >= j) break;</code>		
<code>x = a[i]; a[i] = a[j]; a[j] = x;</code>		<code>/* swap a[i], a[j] */</code>
<code>}</code>		
<code>x = a[i]; a[i] = a[n]; a[n] = x;</code>		<code>/* swap a[i], a[n] */</code>
<code>/* fragment ends here */</code>		
<code>quicksort(m,j); quicksort(i+1,n);</code>		
<code>}</code>		

```
void quicksort(int m, int n)
```

```
{
```

```
    int i, j;
```

```
    int v, x;
```

```
    if (n <= m) return;
```

```
    /* fragment begins here */
```

```
    i = m-1; j = n; v = a[n];
```

```
    while (1) {
```

```
        do i = i+1; while (a[i] < v);
```

```
        do j = j-1; while (a[j] > v);
```

```
        if (i >= j) break;
```

```
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
```

```
    }
```

```
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
```

```
    /* fragment ends here */
```

```
    quicksort(m,j); quicksort(i+1,n);
```

```
}
```

```
t6 = 4*i
```

```
x = a[t6]
```

```
t7 = 4*i
```

```
t8 = 4*j
```

```
t9 = a[t8]
```

```
a[t7] = t9
```

```
t10 = 4*j
```

```
a[t10] = x
```

```
t6 = 4*i
```

```
x = a[t6]
```

```
t8 = 4*j
```

```
t9 = a[t8]
```

```
a[t6] = t9
```

```
a[t8] = x
```