

CSC2/455 Software Analysis and Improvement

Dead Code Elimination

Sreepathi Pai

March 8, 2021

URCS

Outline

Review

Dead Code Elimination

Postscript

Outline

Review

Dead Code Elimination

Postscript

So far

- Source code
- Three-address form
- Control-flow graphs
- SSA form
- Data flow analyses

Outline

Review

Dead Code Elimination

Postscript

Definitions

- Dead code
 - *Useless operation*: Not externally visible
 - *Unreachable code*: Cannot be executed
- *Critical operation*: (Direct) “Useful operation”
 - Operation that computes return value
 - Operation that stores to memory (i.e. is externally visible)
 - Operation that performs I/O
 - ...

Two Steps: Step 1

- Find all directly useful operations and mark them
- Find all indirectly useful operations and mark them
 - I.e. those that feed into directly useful operations
- Iterate until all operations that ultimately feed into directly useful operations have been found and marked

Two Steps: Step 2

- Remove all operations that remain unmarked

Example #1

```
void swap(int *x, int *y) {  
    int t;  
  
    t = *x;  
    *x = *y;  
    *y = t;  
}
```

Example #2

```
int min(int x, int y) {  
    int r;  
  
    if (x > y) {  
        r = y;  
    } else {  
        r = x;  
    }  
  
    return r;  
}
```

Example #2: 3AC

```
int min(int x, int y) {
    int r;
    int t;

    t = x > y;
    if(t == 0) goto L1;

    r = y;
    goto L2;

L1:
    r = x;

L2:
    return r;
}
```

Example #2: With useless operations removed

```
int min(int x, int y) {  
    int r;  
  
    r = y;  
    r = x;  
  
    return r;  
}
```

- Marking and removing useless operations uses only dataflow information
- Must also preserve control flow (i.e. control dependences)
 - How to identify useful branches?

Handling Control Flow

- Assume all “jumps” (unconditional branches) are useful
 - i.e. `goto Lx`
- What about conditional branches?

Conditional Branches: Example

```
int first_N_sum(int N) {  
    int s = 0;  
  
    for(int i = 1; i <= N; i++)  
        s = s + i;  
  
    return N * (N + 1) / 2;  
}
```

3AC code for conditional branches

```
int first_N_sum(int N) {
    int s = 0;
    int i, t;

    i = 1;
L1:
    t = i <= N;
    if(t == 0) goto L2;

    s = s + i;
    i++;
    goto L1;

L2:
    return N * (N + 1) / 2;
}
```

How do we recognize that the conditional branch is useless in this case?

GCC 8.2 for x86-64 (-O0)

```
first_N_sum(int):
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-20], edi
    mov     DWORD PTR [rbp-4], 0          ; s = 0
    mov     DWORD PTR [rbp-8], 1        ; i = 1

.L3:
    mov     eax, DWORD PTR [rbp-8]
    cmp     eax, DWORD PTR [rbp-20]
    jg     .L2
    ; s = s + i
    mov     eax, DWORD PTR [rbp-8]
    add     DWORD PTR [rbp-4], eax
    add     DWORD PTR [rbp-8], 1
    jmp    .L3

.L2:
    mov     eax, DWORD PTR [rbp-20]
    add     eax, 1
    imul   eax, DWORD PTR [rbp-20]
    mov     edx, eax
    shr    edx, 31
    add     eax, edx
    sar    eax
    pop    rbp
    ret
```


GCC 8.2 for x86-64 (-O1)

```
first_N_sum(int):
    test    edi, edi
    jle    .L2
    lea    edx, [rdi+1]
    mov    eax, 1          ; i = 1
.L3:      add    eax, 1          ; i = i + 1
    cmp    eax, edx
    jne    .L3
.L2:      lea    eax, [rdi+1]
    imul   edi, eax
    mov    eax, edi
    shr   eax, 31
    add    eax, edi
    sar   eax
    ret
```

GCC 8.2 for x86-64 (-O2)

```
first_N_sum(int):  
    lea    eax, [rdi+1]  
    imul  edi, eax  
    mov   eax, edi  
    shr  eax, 31  
    add  eax, edi  
    sar  eax  
    ret
```

All compiler output examples obtained using the Compiler Explorer.

Conditional Branches

- A conditional branch is useful only if:
 - A useful operation depends on it
- Control dependence
 - (informal) an operation O is dependent on a branch B if the direction of the branch B affects if O is executed
 - CFG property

Example of control dependence

```
t = x > y
if(t == 0) goto L1

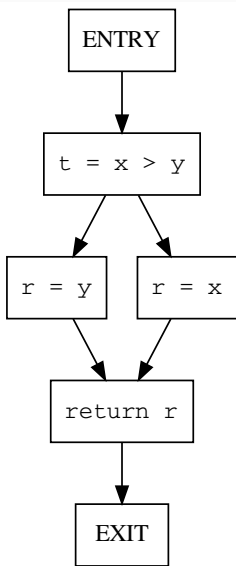
    r = y;
    goto L2;

L1:
    r = x;

L2:
    return r;
```

The assignments to `r` are dependent on `if(t == 0)`, but `return r` is not

Control dependence in the CFG

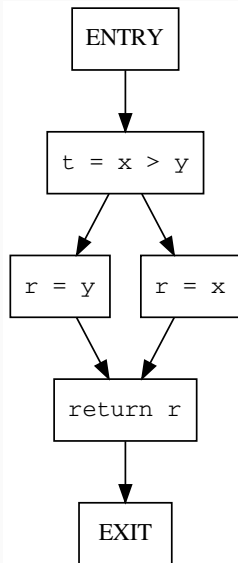


Control Dependence: Formal Definition

- Postdominance
 - A node n postdominates m if it occurs on all paths from m to EXIT
- A node k is control dependent on i if:
 - For a path $i \rightarrow j_0 \rightarrow j_1 \rightarrow \dots \rightarrow k$, k postdominates all j_x
 - k does not strictly postdominate i

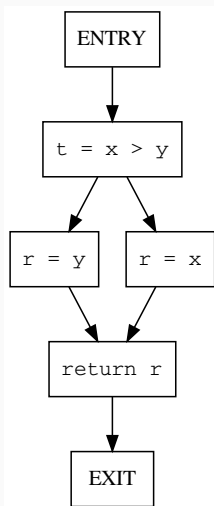
Control Dependence: Example #1

- Consider k : $r = y$
- Is it control dependent on i :
 $t = x > y$?
- Only one path $i \rightarrow k$
 - $r = y$ post-dominates $r = y$
 - $r = y$ does not strictly postdominate i
 - Because it is not a post-dominator of i , and $k \neq i$
- So k is control-dependent on i



Control Dependence: Example #2

- Now, consider k : return r
 - i is still $t = x > y$
- Two paths, first path
 $i \rightarrow j_0 \rightarrow k$
 - j_0 is $r = y$
 - return r post-dominates $r = y$ and itself
 - return r strictly postdominates i
 - Because it is a post-dominator of i , and $k \neq i$
- So k is *not* control-dependent on i

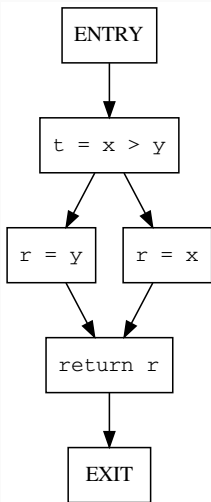


Path #2 of Example #2

- Second path is $i \rightarrow j_1 \rightarrow k$
 - j_1 is $r = x$
- Similar arguments show that k is control-dependent on i

Using Reverse Dominance Frontiers (RDF)

- Given that return r is useful, so are $r = x$ and $r = y$
- We can see that $t = x > y$ is in the *reverse dominance frontier* (*RDF*) of $r = x$ and $r = y$
 - *RDF* is *DF* on edge-reversed CFG.
- Indeed, *RDFs* identify control dependences



Marking unconditional branches useful

- If node k contains useful operations,
- And if k is control-dependent on node i ,
- Then the (conditional) branch in i is useful.
- Operationalized as:
 - If block k contains useful operations
 - Mark all conditional branches in k 's *reverse dominance frontier* $RDF(k)$ as useful
 - RDF computed as DF on *edge-reversed* CFG

Dead Code Elimination: High-level algorithm

- Mark all directly useful operations
- Repeat until convergence
 - Mark all indirectly useful operations
 - Mark all conditional branches in RDFs of useful operations as useful
- Remove all unmarked operations
- Remove empty nodes in CFG / remove all useless control flow

See algorithms in Figure 10.1 and 10.2 in Turczon and Cooper.

Outline

Review

Dead Code Elimination

Postscript

References

- Chapter 10 of Torczon and Cooper
 - Section 10.2