

# TDT4205 Compilers

## Exercise 6

Stian Hvatum (hvatum)  
MTDT

April 9, 2012

### Contents

<b>PART 1</b>	<b>Theory</b>	<b>2</b>
Task 1.1	Optimization . . . . .	2
1.	Easy and common . . . . .	2
2.	Hard to do . . . . .	2
3.	Optimalization impact . . . . .	3
Task 1.2	Misc . . . . .	4
1.	Array bounds checking . . . . .	4
2.	Pointer arithmetics . . . . .	4
3.	Graph coloring and register spilling . . . . .	5
4.	DAGs . . . . .	5

## PART 1 Theory

### Task 1.1 Optimization

#### 1. Easy and common

One easy and common optimization done by compilers is common sub-expression elimination. A common sub-expression is an expression that occurs multiple times either alone or within another expression during the space where the involved variables are not redefined.

```
1 | int main(int argc, char** argv)
2 | {
3 |     int a = 5;
4 |     int b = 10;
5 |     int c = 8;
6 |     int d = a + b * c;
7 |     int e = d - b * c;
8 | }
```

At line 6 and 7 we see that  $b \cdot c$  occurs at both lines. We can reduce the cost of multiplying with this code:

```
1 | int main(int argc, char** argv)
2 | {
3 |     int a = 5;
4 |     int b = 10;
5 |     int c = 8;
6 |     int tmp = b * c;
7 |     int d = a + tmp;
8 |     int e = d - tmp;
9 | }
```

We must notice that sometimes it is still cheaper to calculate the sub-expression twice, since pre-calculating it may involve more memory usage and eventually force the processor to use cache or main memory instead of the faster registers.

#### 2. Hard to do

Register allocation  
Meet-over-all-pases

### 3. Optimization impact

#### Copy Progragation

L7:  $g = f$ , where  $f = e$  from L6.

```
1 | a=1
2 | b=2
3 | c=3
4 | d=a+x
5 | e=b+c
6 |
7 | g=e
8 | g=d+y
9 | a=b+c
```

#### Common subexpression elimination

L5 and L9 both has  $b + c$ .

```
1 | a=1
2 | b=2
3 | c=3
4 | d=a+x
5 | t=b+c
6 | e=t
7 | f=e
8 | g=f
9 | g=d+y
10| a=t
```

#### Contant propagation

a, b and c are all assigned to constants. There are used in L4, L5, L8, L9, and can be replaced with the assigned number.

```
1 | a=1
2 | b=2
3 | c=3
4 | d=1+x
5 | e=2+3
6 | f=e
7 | g=f
8 | g=d+y
9 | a=2+3
```

We could also used constant folding to eliminate all lines except L1, L4 and L8, where we have unknown variables, given none of the other variables are used after the given code

block. If dead code elimination was used, we could eliminate L7, since `g` is assigned to something else at L8.

## Task 1.2 Misc

### 1. Array bounds checking

To do bounds checking on array lookups and assignments, we need to know how large the array is, and we need to check this length against the value of the lookup-variable. This means that we need to store the array length somewhere, most likely in the symbol table, and we need to add some extra assembly code around each array lookup and assignment, unless the lookup is a constant or given to be within bounds by external factors.

A high level abstraction of this:

```
1      int array[5];
2
3      /* No check is needed here, since index is constant */
4      array[3] = 2;
5
6      /* Here we need to check the index,
7         and throw an error if out of bounds */
8
9      if (i >= 0 && i < 5)
10         array[i] = 4;
11     else
12         throw index_out_of_bounds;
```

### 2. Pointer arithmetics

**Live variables** When we are looking for live variables, we usually go backwards and check where the last occurrence of each variable is used. Since pointers point to data areas, there may be problems related to when that memory area was read. If you write to a pointer's data area, and read another pointer's data area, they may refer to the same area, and you cannot declare the first one as dead, even if it is written only (and never “used”).

**Available expressions** Regarding available expressions, we have sort of the same problem as with Live variables. You may have great trouble knowing wherever your expression result is still valid if your data area has been modified by another pointer pointing to the same location.

### **3. Graph coloring and register spilling**

Register allocation is the problem of assigning registers to those results and variables that is needed the most in the current state. Since you have a given number of registers, and the expression calculated at the given state holds a number of variables you can reduce this to the Graph coloring problem, where the colors are registers and the areas of the graph are the variables. Each sub-expression has to be calculated without using the same register twice, hence we cannot have two neighbouring areas that share the same color.

Register spilling is the phenomenon that occurs once we don't have enough registers to calculate a problem. When register spilling occurs, performance is greatly reduced. One of the main objectives of register allocation is to minimize the occurrences of memory spilling.

### **4. DAGs**

Compilers!!!