Compilers

Third year
Spring term

Alfonso Ortega: alfonso.ortega@uam.es
Enrique Alfonseca: Enrique.Alfonseca@uam.es

Lesson 7: Code optimisation

Code optimisation

Introductory concepts

- This process can be performed in three places:
  - Between the semantic analysis and the code generation, to obtain optimised quadruples.
  - During the generation of the code.
  - After the generation (by optimising the code just produced)
- Perfect code optimisation is an undecidable problem (Aho, 70). Therefore,
  - It can only be performed partially.
- Optimisation and debugging are not compatible. For instance, the optimisation may reorder the instructions, or eliminate the code associated to a part of the code, so it will be much more difficult to debug the program.

Code optimisation

Kinds of optimisations

- The different kinds of optimisation can be grouped in the following types:
  - **Machine-dependent**:
    - Assignment of registers
    - Special instructions
    - **Code reordering**
  - **Machine-independent**:
    - **Execution** at compilation time.
    - Elimination of **redundancies**
    - Reordering of operations
    - Frequency reduction
    - **Strength reduction**

Machine-dependent code optimisation

Special instructions

- Most assemblers provide special instructions that facilitate the generation of the code in assembly language.
  - For instance, in nasm
    - **`loop, loope, loopz, loopne, loopnz`**.
      - implement loops with a counter that is decremented at each iteration, even though there is no corresponding instruction in machine code.

Machine-dependent code optimisation

Code reordering

- The underlying idea is that the order of the assembly instructions may influence the size of the final program.
- The following is an example:
  - The following ASPLE code

```
a:=b/c;
d:=b%c;
```

  - can produce the following code in assembly language

```
mov dword eax, [b]
cdq
idiv dword eax, [c]
mov dword [a], eax
mov dword eax, [b]
cdq
idiv dword eax, [c]
mov dword [d], edx
```


Machine-dependent code optimisation

Code reordering

- That code is not as efficient as it could be, as the instruction **`idiv`** leaves the remainder of the division in **`edx`**, and it is not necessary to repeat it.

- The following code is equivalent:

```
mov dword eax, [b]
cdq
idiv dword eax, [c]
mov dword [a], eax
mov dword [d], edx
```


Machine independent code optimisation

Execution during the compilation

- The motivation of this kind of optimisation is that there are some expressions whose value can be determined by the compiler during execution time.

- To do that, it is necessary to know the value of each variable at each point in the program:
  - It can be stored in the symbols table
  - or in a special table used just for this purpose.

- This optimisation is usually applied:
  - To arithmetic and logic operations.
  - To type conversions.

Machine independent code optimisation

Execution during the compilation

- The following **algorithm** will be applied to each of the quadruples:
  - `(op, op1, op2, res)`, where `op1` is an identifier, and `(op1,v1)` is in the symbols table `T`
    - We substitute the quadruple `op1` by `v1`.
  - `(op, op1, op2, res)`, where `op2` is an identifier, and `(op2,v2)` is in the symbols table `T`
    - We substitute the quadruple `op2` by `v2`.
  - `(op, v1, v2, res)`, where `v1` and `v2` are constant values
    - We eliminate the quadruple, remove from `T` the pair `(res, v)`, if it exists, and add to `T` the pair `(res, v1 op v2)`, unless `v1 op v2` produces an error. In this case, we shall output a warning message and leave the quadruple as it was.
    - Example: `if (false) f = 1/0;` (warning, not error message)
  - `(=, v1, , res)`, remove from `T` the pair `(res, v)`, if exists. If `v1` is a constant value, add to `T` the pair `(res, v1)`.

Machine independent code optimisation

Execution during the compilation

`(+,2,3,t1)`	Remove quadruple, `T = {(t1,5)}`
`(=,t1,,i)`	Subst. by `(=,5,,i)`, `T = {(t1,5),(i,5)}`
`(=,4,,i)`	`T = {(t1,5),(i,4)}`
`(CIF,i,,t2)`	Subst. by `(CIF,4,,t2)`, Rem. quadruple, `T = {(t1,5),(i,4),(t2,4.0)}`
`(+,t2,2.5,t3)`	Subst. by `(+,4.0,2.5,t3)` Rem. quadruple, `T = {(t1,5),(i,4),(t2,4.0),(t3,6.5)}`
`(=,t3,,f)`	Subst. by `(=,6.5,,f)`

- Optimised quadruples:

```
(=,5,,i)
(=,4,,i)
(=,6.5,,f)
```


Machine independent code optimisation

Execution during the compilation: observations

- The compiler must have the values stored in the table permanently, and it has to be correctly updated.
- Whenever a variable may possibly change its value, it has to keep track of it:
  - After a label, every value is forgotten.
  - Loops
  - Function calls
  - etc.

- Problem: if the compiler executes in a computer, and generates code for a different computer, the first compiler might have less precision than the second one.

Machine independent code optimisation

Removing redundancies: introductory example

```
int a,b,c,d;

a = a+b*c;    (*,b,c,t1)        (*,b,c,t1)
              (+,a,t1,t2)       (+,a,t1,t2)
              (=,t2,,a)         (=,t2,,a)
d = a+b*c;    (*,b,c,t3)
              (+,a,t3,t4)       (+,a,t1,t4)
              (=,t4,,d)         (=,t4,,d)
b = a+b*c;    (*,b,c,t5)
              (+,a,t5,t6)
              (=,t6,,b)         (=,t4,,b)
```

Machine independent code optimisation

Removing redundancies: algorithm

- Each of the variable in the symbols table is assigned the dependency -1.
- Number the quadruples.
- for (i=0; i< number-of-quadruples ; i++) {
  - The identifier that stores the result of quadruple (i) is assigned the dependency i.
  - The quadruple number (i) will be assigned as dependency:

    1 + max. number of dependencies of the operands.
  - If
    - Either quadruple i is not an assignment, and it has the same operation code and operands as j, j<i,
    - Or quadruple i is an assignment, and it has the same operation code, operands, and result variable,
  - and the dependencies of both quadruples are the same, then:
    - Substitute the quadruple i by a null quadruple (SAME,j,0,0), which will not generate code.
    - In the next quadruples, we are going to substitute the result of quadruple i with the result of quadruple j.

Machine independent code optimisation

Removing redundancies: example

					DEP
0	*	b	c	t1	
1	+	a	t1	t2	
2	=	t2		a	
3	*	b	c	t3	
4	+	a	t3	t4	
5	=	t4		d	
6	*	b	c	t5	
7	+	a	t5	t6	
8	=	t6		b	

VAR	DEP
a	-1
b	-1
c	-1
d	-1

INITIAL SET OF QUADRUPLES

Machine independent code optimisation

Removing redundancies: example

i					DEP
0	*	b	c	t1	0
1	+	a	t1	t2	
2	=	t2		a	
3	*	b	c	t3	
4	+	a	t3	t4	
5	=	t4		d	
6	*	b	c	t5	
7	+	a	t5	t6	
8	=	t6		b	

VAR	DEP
a	-1
b	-1
c	-1
d	-1
t1	0

QUADRUPLE 0

Machine independent code optimisation

Removing redundancies: example

					DEP
0	*	b	c	t1	0
1	+	a	t1	t2	1
2	=	t2		a	
3	*	b	c	t3	
4	+	a	t3	t4	
5	=	t4		d	
6	*	b	c	t5	
7	+	a	t5	t6	
8	=	t6		b	

VAR	DEP
a	-1
b	-1
c	-1
d	-1
t1	0
t2	1

QUADRUPLE 1

Machine independent code optimisation

Removing redundancies: example

					DEP
0	*	b	c	t1	0
1	+	a	t1	t2	1
2	=	t2		a	2
3	*	b	c	t3	
4	+	a	t3	t4	
5	=	t4		d	
6	*	b	c	t5	
7	+	a	t5	t6	
8	=	t6		b	

VAR	DEP
a	~~-1~~ 2
b	-1
c	-1
d	-1
t1	0
t2	1

QUADRUPLE 2

Machine independent code optimisation

Removing redundancies: example

					DEP
0	*	b	c	t1	0
1	+	a	t1	t2	1
2	=	t2		a	2
3	*	b	c	t3	0
4	+	a	t3	t4	
5	=	t4		d	
6	*	b	c	t5	
7	+	a	t5	t6	
8	=	t6		b	

VAR	DEP
a	~~-1~~ 2
b	-1
c	-1
d	-1
t1	0
t2	1
t3	3

QUADRUPLE 3.1

Machine independent code optimisation

Removing redundancies: example

					DEP
0	*	b	c	t1	0
1	+	a	t1	t2	1
2	=	t2		a	2
3	*	b	c	t3	0
	SAME	0	-	-	
4	+	a	t3	t4	
5	=	t4		d	
6	*	b	c	t5	
7	+	a	t5	t6	
8	=	t6		b	

VAR	DEP
a	~~-1~~ 2
b	-1
c	-1
d	-1
t1	0
t2	1
t3	3

QUADRUPLE 3.2

Machine independent code optimisation

Removing redundancies: example

					DEP
0	*	b	c	t1	0
1	+	a	t1	t2	1
2	=	t2		a	2
3	SAME	0	-	-	(0)
4	+	a	t3	t4	
5	=	t4		d	
6	*	b	c	t5	
7	+	a	t5	t6	
8	=	t6		b	

VAR	DEP
a	~~-1~~ 2
b	-1
c	-1
d	-1
t1	0
t2	1
t3	3

QUADRUPLE 3.3

Machine independent code optimisation

Removing redundancies: example

					DEP
0	*	b	c	t1	0
1	+	a	t1	t2	1
2	=	t2		a	2
3	SAME	$\boxed{0}$	-	-	(0)
4	+	a	t3	t4	
			t1		
5	=	t4		d	
6	*	b	c	t5	
7	+	a	t5	t6	
8	=	t6		b	

VAR	DEP
a	2
b	-1
c	-1
d	-1
t1	0
t2	1
t3	3

QUADRUPLE 4.1

Machine independent code optimisation

Removing redundancies: example

					DEP
0	*	b	c	$\boxed{t1}$	0
1	+	a	t1	t2	1
2	=	t2		a	2
3	SAME	0	-	-	(0)
4	+	a	~~t3~~	t4	
			$\boxed{t1}$		3
5	=	t4		d	
6	*	b	c	t5	
7	+	a	t5	t6	
8	=	t6		b	

VAR	DEP
a	2
b	-1
c	-1
d	-1
t1	0
t2	1
t3	3
t4	4

QUADRUPLE 4.2

Machine independent code optimisation

Removing redundancies: example

					DEP
0	*	b	c	t1	0
1	+	a	t1	t2	1
2	=	t2		a	2
3	SAME	0	-	-	(0)
4	+	a	t1	t4	3
5	=	t4		d	5
6	*	b	c	t5	
7	+	a	t5	t6	
8	=	t6		b	

VAR	DEP
a	2
b	-1
c	-1
d	~~-1~~ 5
t1	0
t2	1
t3	3
t4	4

QUADRUPLE 5

Machine independent code optimisation

Removing redundancies: example

					DEP
0	*	b	c	t1	$\boxed{0}$
1	+	a	t1	t2	1
2	=	t2		a	2
3	SAME	0	-	-	(0)
4	+	a	t1	t4	3
5	=	t4		d	5
6	*	b	c	t5	$\boxed{0}$
	SAME	0	-	-	
7	+	a	t5	t6	
8	=	t6		b	

VAR	DEP
a	2
b	-1
c	-1
d	5
t1	0
t2	1
t3	3
t4	4
t5	6

QUADRUPLE 6

Machine independent code optimisation

Removing redundancies: example

					DEP	VAR	DEP
0	*	b	c	t1	0	a	2
1	+	a	t1	t2	1	b	-1
2	=	t2		a	2	c	-1
3	SAME	0	-	-	(0)	d	5
4	+	a	t1	t4	3	t1	0
5	=	t4		d	5	t2	1
6	SAME	0	-	-	(0)	t3	3
7	+	a	~~t5~~	t6	3	t4	4
			t1			t5	6
8	=	t6		b		t6	7

QUADRUPLE 7.1

Machine independent code optimisation

Removing redundancies: example

					DEP	VAR	DEP
0	*	b	c	t1	0	a	2
1	+	a	t1	t2	1	b	-1
2	=	t2		a	2	c	-1
3	SAME	0	-	-	(0)	d	5
4	+	a	t1	t4	3	t1	0
5	=	t4		d	5	t2	1
6	SAME	0	-	-	(0)	t3	3
7	+	a	t1	t6	3	t4	4
	SAME	4	-	-		t5	6
8	=	t6		b		t6	7

QUADRUPLE 7.2

Machine independent code optimisation

Removing redundancies: example

					DEP	VAR	DEP
0	*	b	c	t1	0	a	2
1	+	a	t1	t2	1	b	-1
2	=	t2		a	2	c	-1
3	SAME	0	-	-	(0)	d	5
4	+	a	t1	t4	3	t1	0
5	=	t4		d	5	t2	1
6	SAME	0	-	-	(0)	t3	3
7	SAME	4	-	-	(3)	t4	4
8	=	~~t6~~		b	5	t5	6
		t4				t6	7

QUADRUPLE 8.1

Machine independent code optimisation

Removing redundancies: example

					DEP	VAR	DEP
0	*	b	c	t1	0	a	2
1	+	a	t1	t2	1	b	~~-1~~ 8
2	=	t2		a	2	c	-1
3	SAME	0	-	-	(0)	d	5
4	+	a	t1	t4	3	t1	0
5	=	t4		d	5	t2	1
6	SAME	0	-	-	(0)	t3	3
7	SAME	4	-	-	(3)	t4	4
8	=	t4		b	5	t5	6
						t6	7

QUADRUPLE 8.2

Machine independent code optimisation

Removing redundancies: example

(*	b	c	t1)	t1 = b * c
(+	a	t1	t2)	a = a + t1
(=	t2		a)	
(+	a	t1	t4)	d = a + t1
(=	t4		d)	
(=	t4		b)	b = t4

FINAL RESULT

Machine independent code optimisation

Reordering operations

- We can take advantage of the commutability and associativity of some operations to obtain a code which is more efficient, without altering the result.
- For instance, let us consider the following quadruple:

  `(*, b, c , -) ≡ (*, c, b, -)`

- We shall see the following possibilities:
  - Fixing a canonical ordering of the operands in commutative operations.
  - Maximising the use of monadic operators, to increase the number of equivalent quadruples.
  - Reusing intermediate variables, when applying the properties of the operations, to transform the initial expressions.

Machine independent code optimisation

Reordering operations

- We can order the operands according to the following order:
  1. Terms that are neither variables nor constants.
  2. Variables, by alphabetical order.
  3. Constants.
- Example:

  `a=1+c+d+3; a=c+d+1+3; b=d+c+2; b=c+d+2;`

  - In this case, the reordering allows us:
    - To optimise `1+3` because it will be done during compilation time.
    - Optimise the calculations, because `c+d` will be identified as common parts of those expressions.
- We shall not attempt to optimise every possible situation. For instance:

  `a=1+c+d+3; a=c+d+1+3; b=d+c+c+d; b=c+c+d+d`

  - In this case, one of the two `c+d` is not recognised as common part of those expressions.

Machine independent code optimisation

Reordering operations: increasing the use of unary operators

- Some sub-expressions may appear in other parts of the code, affected by a monadic operator. For instance, in this example, we have `c-d` and `d-c`:

```
a = c-d;     (-,c,d,t1)   (-,c,d,t1)
             (=,t1,,a)    (=,t1,,a)
b = d-c;     (-,d,c,t2)   (-,t1,,t2)
             (=,t2,,b)    (=,t2,,b)
```

- In this case, the optimisation does not reduce the number of clauses, but a clause is substituted by other which is more efficient.

Machine independent code optimisation

Reordering operations: reducing intermediate variables

- Note that reducing the number of intermediate variables may be incompatible with other kinds of optimisation.
- The procedure will be illustrated with one example. Consider the following expression:

  `(a*b)+(c+d)`

- Because addition is associative:

  `(a*b)+(c+d) ≡ ((a*b)+c )+d`

  - We have two possible sets of tuples for this expression:

```
(*,a,b,t1)    (*,a,b,t1)
(+,c,d,t2)    (+,t1,c,t1)
(+,t1,t2,t1)  (+,t1,d,t1)
```

- Because addition is commutative,

  `(a+b)+(c*d) ≡ (c*d)+(a+b)`

  - the following are alternative sets of tuples for the expression:

```
(+,a,b,t1)    (*,c,d,t1)
(*,c,d,t2)    (+,a,t1,t1)
(+,t1,t2,t1)  (+,t1,b,t1)
```


Machine independent code optimisation

Algorithm for calculating the min. number of variables required

1. Build a graph for the expression.
2. If `(j,k)` are the labels of the children of node `i`:
3. If `(j=k)`
   1. Associate `(k+1)` to node `i`.
4. Otherwise, associate `max(j,k)` to node `i`.

Machine independent code optimisation: example

(a*b)+(c+d)

```
          +(2)
  ---------------------
   *(1)           +(1)
 -------------   ------------
 a(0)   b(0)    c(0)   d(0)
```

((a*b)+c)+d

```
               +(1)
        --------------------
         +(1)           d(0)
 -------------------
  *(1)          c(0)
-------------
a(0)    b(0)
```

(a+b)+(c*d)

```
         +(2)
   -----------------
   +(1)          *(1)
-------------  --- ---------
a(0)   b(0)   c(0)   d(0)
```

a+(c*d)+b

```
              +(1)
        -------------------
        +(1)            b(0)
------------------
a(0)        *(1)
        ------------
        c(0)    d(0)
```


Machine independent code optimisation

Loop optimisation

- It is possible to optimise the execution of loops in the following two ways:
  1. By **loop invariance**:
     - An operation is **invariant with respect to a loop** if none of its operands changes its value during the execution of the loop.
     - In these cases, it is possible to take the comparison out of the loop, because it is not necessary to execute it inside the loop.
  2. By **strength reduction**:
     - This consists in substituting, inside the loop, a **strong** operation (one that is computationally expensive, such as the product) with a **weak** operation (a less expensive one), such as the addition or the change of sign.

Machine independent code optimisation

Optimising loops with **strength reduction**

- In the following loop:

```
for (i=a; i<c; i+=b) {... d=i*k; ...}
```

  - the variable `d` receives the following values:

```
d=a*k
d=(a+b)*k
d=(a+b+b)*k
...
```

  - We can assume that
    - `b`,`k` are invariant inside the loop (if `b` is an expression, all its operands should be invariant)
    - `i` is not modified inside the body of the loop. It only changes in the instruction `i+=b`
    - `d` is never used before the instruction, and it is not modified afterwards.

Machine independent code optimisation

Optimising loops with **strength reduction**

- In this case, we can substitute the loop with the following code:

```
d=a*k;
t1=b*k;
for (i=a; i<c; i+=b, d+=t1) {...}
```

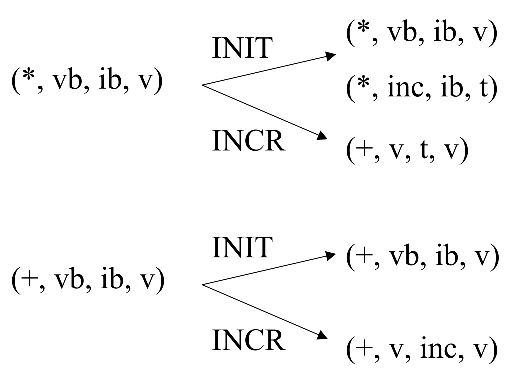
- In this case, the values of `d` are the same as before:

```
d=a*k
d=a*k+b*k
d=a*k+b*k+b*k
...
```

- But, at each iteration of the loop, rather than executing a product we only need to execute an addition.

Machine independent code optimisation

Optimising loops with **strength reduction**: generalisation



vb – variable of the loop

ib – invariant variable in the loop

inc – increment of vb

Machine independent code optimisation

Optimising loops with **strength reduction**: example

- In the following loop:

```
for (i=0; i<10; i++) {... a=(b+c*i)*d; ...}
```

  - The variables `b`, `c`, `d` are invariant in the loop.
  - We only need to identify in each sentence which is invariant, which is the increment, and which is the loop variable.

```
INIT:  (=,  0,  ,   ,i)
LOOP:       ...
       (*,  c,  i,  t1)
       (+,  b,  t1, t2)
       (*,  t2, d,  t3)
       (=,  t3, ,   a)
            ...
INCR:  (+,  i,  1,  i)
```

inc' vb

```
INIT:  (=,  0,  ,   i)
       (*,  c,  i,  t1)
       (*,  c,  1,  t4)
LOOP:       ...
       (+,  b,  t1, t2)
       (*,  t2, d,  t3)
       (=,  t3, ,   a)
            ...
INCR:  (+,  i,  1,  i)
       (+,  t1, t4, t1)
```

Machine independent code optimisation

Optimising loops with **strength reduction**: example

```
INIT: (=,   0,    ,    i)
      (*,   c,    0,   t1)
      (*,   c,    1,   t4)
LOOP:       ...
      (+,   b,    t1,  t2)
      (*,   t2,   d,   t3)
      (=,   t3,   ,    a)
            ...
INCR: (+,   i,    1,   i)
      (+,   t1,   t4,  t1)
```

inc’ vb’

```
INIT: (=,   0,    ,    i)
      (*,   c,    0,   t1)
      (*,   c,    1,   t4)
      (+,   b,    t1,  t2)
LOOP:       ...
      (*,   t2,   d,   t3)
      (=,   t3,   ,    a)
            ...
INCR: (+,   i,    1,   i)
      (+,   t1,   t4,  t1)
      (+,   t2,   t4,  t2)
```

This is not used inside the loop, so we may remove it

Machine independent code optimisation

Optimising loops with **strength reduction**: example

```
INIT: (=,   0,    ,    i)
      (*,   c,    0,   t1)
      (*,   c,    1,   t4)
      (+,   b,    t1,  t2)
LOOP:       ...
      (*,   t2,   d,   t3)
      (=,   t3,   ,    a)
            ...
INCR: (+,   i,    1,   i)
      (+,   t2,   t4,  t2)
```

inc’ vb’

```
INIT: (=,   0,    ,    i)
      (*,   c,    0,   t1)
      (*,   c,    1,   t4)
      (+,   b,    t1,  t2)
      (*,   t2,   d,   t3)
      (*,   t4,   d,   t5)
LOOP:       ...
      (=,   t3,   ,    a)
            ...
INCR:
      (+,   i,    1,   i)
      (+,   t2,   t4,  t2)
      (+,   t3,   t5,  t3)
```

This is not used inside the loop, so we may remove it

Machine independent code optimisation

Optimising loops with **strength reduction**: example

- Final result:

```
INIT:   (=,   0,    ,    i)
        (*,   c,    0,   t1)
        (*,   c,    1,   t4)
        (+,   b,    t1,  t2)
        (*,   t2,   d,   t3)
        (*,   t4,   d,   t5)
LOOP:   ...
        (=,   t3,   ,    a)
        ...
INCR:   (+,   i,    1,   i)
        (+,   t3,   t5,  t3)
```


Machine independent code optimisation

Optimising loops with **strength reduction**: example

- If we have nested loops, strength reduction should be applied starting in the inner ones.
- If there are functions calls inside the loop,
  - The compiler must know whether the function receives the reference of a variable (in this case it may not be invariant). Equally, the function might change the value of global variables.
- It a loop will only be executed a few times, the optimisation will not be appreciated. It may even degrade the performance.
- This means that loop optimisation is not always recommended. Sometimes, it is performed in two steps:
  - Analysing of the program, and gathering information about the variables.
  - Loop optimisation.

Machine independent code optimisation

Other optimisations

- **Region optimisation**:
  - When we try to optimise a complete program, we apply all these optimisations, and the program is next represented as a graph, with blocks, indicating the program's control flow.
  - There are techniques to simplify the blocks.
- Concerning **dead assignments**:
  - In a program, a variable might receive a value, which is not used before the variable is assigned a different value.
  - In this case, we say that the first assignment is a dead assignment, and we can remove it.
  - To identify these circumstances in the whole program may be too costly, because it may be necessary to consider every possible control flow.