CC Lecture 4

Prepared for: 7th Sem, CE, DDU
Prepared by: Niyati J. Buch

Optimizing Transformations

1. Compile time evaluation
2. **Elimination of common subexpression**
3. Dead code elimination
4. Frequency reduction
5. Strength reduction

Common Subexpression Elimination

- **Common subexpression elimination** finds computations that are always performed at least twice on a given execution path and eliminates the second and later occurrences of them.

- An occurrence of an expression in a program is a common subexpression:-
  - if there is another occurrence of the expression whose evaluation always precedes this one in execution order
  - and if the operands of the expression remain unchanged between the two evaluations.

Common Subexpression Elimination

- **Common- subexpression elimination** is a transformation that removes the re-computations of common subexpressions and replaces them with the uses of saved values.

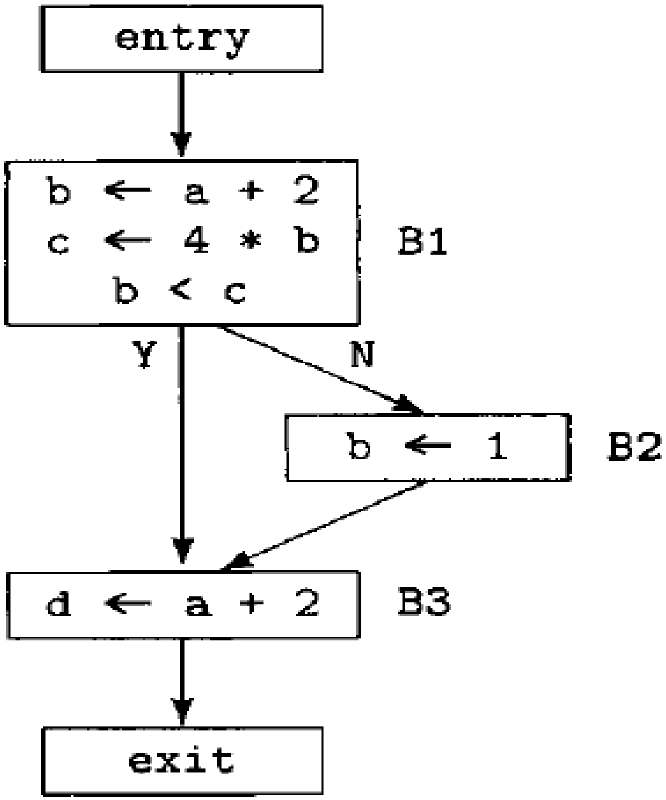
- Also, note that common-subexpression elimination may not always be worthwhile.

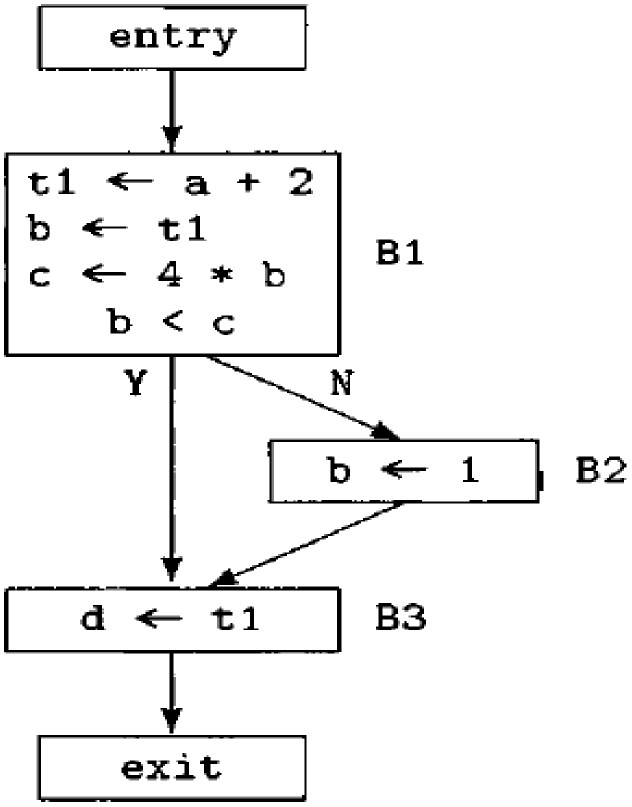
- Optimizers frequently divide common-subexpression elimination into two phases,
  1. **local**: done within each basic block,
  2. **Global:** done across an entire flowgraph.

Common-Subexpression Elimination

Example of a common subexpression, namely, **a + 2**,

the result of doing common-subexpression elimination on it.





To do local common-subexpression elimination

- we iterate through the basic block
- adding entries to and removing them from **AEB** (Available Expression Block) as appropriate
- inserting instructions to save the expressions' values in temporaries
- modifying the instructions to use the temporaries instead.

Available Expressions

- **Available expressions** is an analysis algorithm that determines for each point in the program, the set of expressions that need not be recomputed.

- Those expressions are said to be ***available*** at such a point.

- To be available on a program point, the operands of the expression should not be modified on any path from the occurrence of that expression to the program point.

Algorithm using AEB

- For each instruction inst at position i, we determine whether it computes a binary expression or not and then execute one of two cases accordingly.

- The (nontrivial) binary case is as follows:

1. We compare inst's operands and operator with those in the quintuples in AEB.

   If we find a match, say, **(pos, opd1, opr, opd2, tmp),** we check whether **tmp is nil**.

Algorithm using AEB

If it is, we

(a) generate a new temporary variable name **ti** and replace the nil in the identified triple by it,

(b) insert the instruction **ti ← opd1 opr opd2** immediately before the instruction at position pos, and

(c) replace the expressions in the instructions at positions **pos** and **i by ti**.

Algorithm using AEB

If we found a match with **tmp = ti**, where ti ≠ nil, we replace the expression in **inst by ti**.

If we did not find a match for inst’s expression in AEB, we insert a quintuple for it, with tmp = nil, into AEB.

2. We check whether the **result variable** of the current instruction, if there is one, occurs as an operand in any element of AEB.

   If it does, we **remove all such quintuples** from AEB.

Example: basic block before local common-subexpression elimination.

Position	Instruction
1	c ← a + b
2	d ← m & n
3	e ← b + d
4	f ← a + b
5	g ← - b
6	h ← b + a
7	a ← j + a
8	k ← m & n
9	j ← b + d
10	a ← - b
11	If m & n goto L2

Position	Instruction
1	c ← a + b
2	d ← m & n
3	e ← b + d
4	f ← a + b
5	g ← - b
6	h ← b + a
7	a ← j + a
8	k ← m & n
9	j ← b + d
10	a ← - b
11	If m & n goto L2

Entry: AEB = ∅

Position 1 :
AEB = {**<1, a, +, b, nil>**}

Position 2 :
AEB = { <1, a, +, b, nil>,
<2, m, &, n, nil>}

Position 3 :
AEB = { <1, a, +, b, nil>,
<2, m, &, n, nil>,
<3, b, +, d, nil>}

Position	Instruction
1	c ← a + b
2	d ← m & n
3	e ← b + d
4	f ← a + b
5	g ← - b
6	h ← b + a
7	a ← j + a
8	k ← m & n
9	j ← b + d
10	a ← - b
11	If m & n goto L2

Position 4 :

f ← a + b matches with the first quintuple in AEB.

AEB = { <1, a, +, b, nil>,
<2, m, &, n, nil>,
<3, b, +, d, nil>}

So, insert **t1** into that quintuple in place of nil, generate the instruction **t1 ← a + b** before position 1 and renumber the entries in AEB,

replace the instruction that was in position 1 but that is now in position 2 by **c ← t1**, set i = 5, and replace the instruction in position 5 by **f ← t1**.

Position	Instruction
1	**t1 ← a + b**
2	c ← **t1**
3	d ← m & n
4	e ← b + d
5	f ← **t1**
6	g ← - b
7	h ← b + a
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 4 :

f ← a + b matches with the first quintuple in AEB.

AEB = { <1, a, +, b, nil>,
<2, m, &, n, nil>,
<3, b, +, d, nil>}

So, insert **t1** into that quintuple in place of nil, generate the instruction **t1 ← a + b** before position 1 and renumber the entries in AEB,

replace the instruction that was in position 1 but that is now in position 2 by **c ← t1**, set i = 5, and replace the instruction in position 5 by **f ← t1**.

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← - b
7	h ← b + a
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 5 :

AEB = { <1, a, +, b, **t1**>,
<3, m, &, n, nil>,
<4, b, +, d, nil>}

Position 6:

AEB = { <1, a, +, b, t1>,
<3, m, &, n, nil>,
<4, b, +, d, nil>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← - b
7	h ← b + a
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 7:
h ← b + a, matches (commutative property)

AEB = { <1, a, +, b, t1>,
<3, m, &, n, nil>,
<4, b, +, d, nil>}

[no change]

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← - b
7	h ← **t1**
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 7:
h ← b + a, matches
(commutative property)

AEB = { <1, a, +, b, t1>,
<3, m, &, n, nil>,
<4, b, +, d, nil>}
[no change]

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← - b
7	h ← t1
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 8:

a ← j + a

here variable matches in operand of quintuple so remove **<1, a, +, b, t1>** from AEB

AEB = {<3, m, &, n, nil>,
<4, b, +, d, nil>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← - b
7	h ← t1
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 9:

m & n is recognized as common sub expression

So,

AEB = {<3, m, &, n, **t2**>,
<**5**, b, +, d, nil>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← - b
7	h ← t1
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 9:
m & n is recognized as common sub expression
So,

AEB = {<3, m, &, n, **t2**>,
<**5**, b, +, d, nil>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	**t2 ← m & n**
4	d ← t2
5	e ← b + d
6	f ← t1
7	g ← - b
8	h ← t1
9	a ← j + a
10	k ← **t2**
11	j ← b + d
12	a ← - b
13	If m & n goto L2

Position 9:
m & n is recognized as common sub expression
So,

AEB = {<3, m, &, n, **t2**>,
<**5**, b, +, d, nil>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	e ← b + d
6	f ← t1
7	g ← - b
8	h ← t1
9	a ← j + a
10	k ← t2
11	j ← b + d
12	a ← - b
13	If m & n goto L2

New Position 11:
b + d is recognized as common sub expression

AEB = {<3, m, &, n, t2>,
<5, b, +, d, **t3**>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	e ← b + d
6	f ← t1
7	g ← - b
8	h ← t1
9	a ← j + a
10	k ← t2
11	j ← b + d
12	a ← - b
13	If m & n goto L2

New Position 11:
b + d is recognized as common sub expression

AEB = {<3, m, &, n, t2>,
<5, b, +, d, **t3**>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	e ← b + d
6	f ← t1
7	g ← - b
8	h ← t1
9	a ← j + a
10	k ← t2
11	j ← b + d
12	a ← - b
13	If m & n goto L2

New Position 11:
b + d is recognized as common sub expression

AEB = {<3, m, &, n, t2>,
<5, b, +, d, **t3**>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	t3 ← b + d
6	e ← t3
7	f ← t1
8	g ← - b
9	h ← t1
10	a ← j + a
11	k ← t2
12	j ← t3
13	a ← - b
14	If m & n goto L2

New Position 11:
b + d is recognized as common sub expression

AEB = {<3, m, &, n, t2>,
<5, b, +, d, **t3**>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	t3 ← b + d
6	e ← t3
7	f ← t1
8	g ← - b
9	h ← t1
10	a ← j + a
11	k ← t2
12	j ← t3
13	a ← - b
14	If m & n goto L2

Position 13:

AEB = {<3, m, &, n, t2>,
<5, b, +, d, t3>}

[no change]

Position 14 :

m & n is a common sub expression found

AEB = {<3, **m, &, n**, t2>,
<5, b, +, d, t3>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	t3 ← b + d
6	e ← t3
7	f ← t1
8	g ← - b
9	h ← t1
10	a ← j + a
11	k ← t2
12	j ← t3
13	a ← - b
14	If **t2** goto L2

Position 13:

AEB = {<3, m, &, n, t2>,
<5, b, +, d, t3>}

[no change]

Position 14 :

m & n is a common sub expression found

AEB = {<3, **m, &, n**, t2>,
<5, b, +, d, t3>}

Conclusion

- In the original form of this code there are 11 instructions, 12 variables, and 9 binary operations performed,
- while in the final form there are 14 instructions, 15 variables, and 4 binary operations performed.

Conclusion

- Assuming all the variables occupy registers and that each of the register-to-register operations requires only a single cycle, as in any RISC and the more advanced CICSs, the original form is to be preferred, since it has fewer instructions and uses fewer registers.
- On the other hand, if some of the variables occupy memory locations or the redundant operations require more than one cycle, the result of the optimization is to be preferred.
- Thus, **whether an optimization actually improves the performance of a block of code depends on both the code and the machine it is executed on.**