TDDB44/TDDD55 Lecture 10: Code Optimization

Peter Fritzson and Christoph Kessler and Martin Sjölund

Department of Computer and Information Science
Linköping University

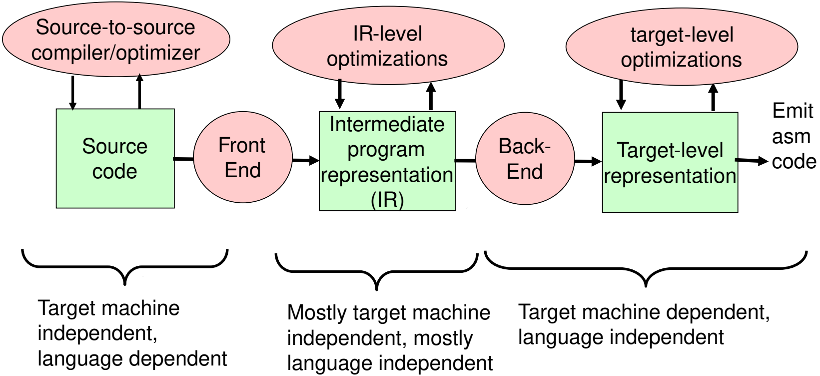
2020-11-30

Part I

Overview

Code Optimization – Overview

Goal Code that is faster and/or smaller and/or low energy consumption



Remarks

- Often multiple levels of IR:
  - high-level IR (e.g. abstract syntax tree AST),
  - medium-level IR (e.g. quadruples, basic block graph),
  - low-level IR (e.g. directed acyclic graphs, DAGs)
  - → do optimization at most appropriate level of abstraction
  - → code generation is continuous lowering of the IR towards target code
- "Postpass optimization": done on *binary code* (after compilation or without compilation)

Disadvantages of Compiler Optimizations

- Debugging made difficult
  - Code moves around or disappears
  - Important to be able to switch off optimization

Note: Some compilers have an optimization level `-Og` that avoids optimizations that makes debugging hard

- Increases compilation time
- May even affect program semantics
  - `A = B*C - D + E` $\longrightarrow$ `A = B*C + E - D`
    may lead to overflow if `B*C+E` is too large a number

Optimization at Different Levels of Program Representation

Source-level optimization

- Made on the source program (text)
- Independent of target machine

Intermediate code optimization

- Made on the intermediate code (e.g. on AST trees, quadruples)
- Mostly target machine independent

Target-level code optimization

- Made on the target machine code
- Target machine dependent

Source-level Optimization

At source code level, independent of target machine

- Replace a slow algorithm with a quicker one, e.g. Bubble sort $\rightarrow$ Quick sort
- Poor algorithms are the main source of inefficiency but difficult to automatically optimize
- Needs pattern matching, e.g. Kessler 1996; Di Martino and Kessler 2000

Intermediate Code Optimization

At the intermediate code (e.g., trees, quadruples) level

In most cases target machine independent

- Local optimizations within basic blocks (e.g. common subexpression elimination)
- Loop optimizations (e.g. loop interchange to improve data locality)
- Global optimization (e.g. code motion, within procedures)
- Interprocedural optimization (between procedures)

Target-level Code Optimization

At the target machine binary code level
Dependent on the target machine

- Instruction selection, register allocation, instruction scheduling, branch prediction
- Peephole optimization

Basic Block

A **basic block** is a sequence of textually consecutive operations (e.g. quadruples) that contains no branches (except perhaps its last operation) and no branch targets (except perhaps its first operation).

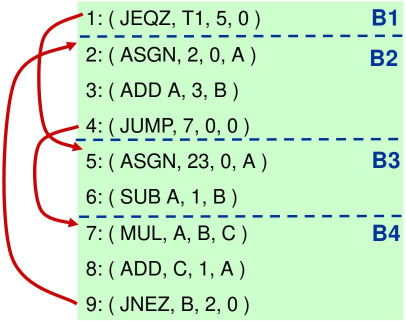
- Always executed in same order from entry to exit
- A.k.a. *straight-line code*

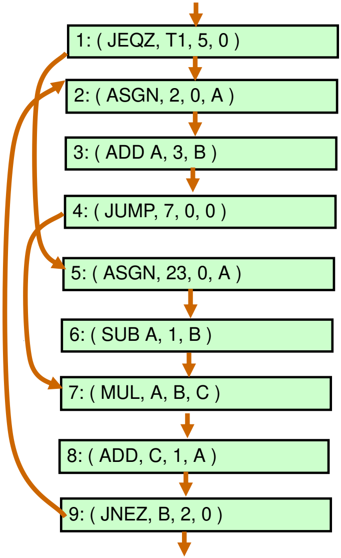
```
1: ( JEQZ, T1, 5, 0 )          B1
-----------------------------------
2: ( ASGN, 2, 0, A )           B2
3: ( ADD A, 3, B )
4: ( JUMP, 7, 0, 0 )
-----------------------------------
5: ( ASGN, 23, 0, A )          B3
6: ( SUB A, 1, B )
-----------------------------------
7: ( MUL, A, B, C )            B4
8: ( ADD, C, 1, A )
9: ( JNEZ, B, 2, 0 )
```

Control Flow Graph

Nodes primitive operations (e.g. quadruples), or basic blocks

Edges control flow transitions

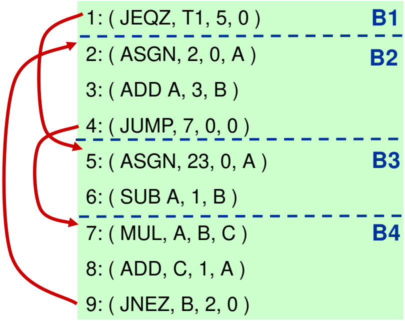


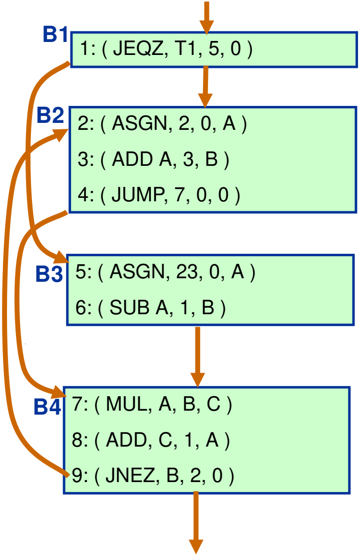


Basic Block Control Flow Graph

Nodes basic blocks

Edges control flow transitions





Part II

Local Optimization
(within single Basic Block)

Local Optimization

Within a single basic block (needs no information about other blocks)
Example: **Constant folding** (Constant propagation)
Computes constant expressions at compile time

```
const int NN = 4;
// ...
i = 2 + NN;
j = i * 5 + a;
```

$\rightarrow$

```
const int NN = 4;
// ...
i = 6;
j = 30 + a;
```

Local Optimization (cont.)

Elimination of common subexpressions

Common subexpression elimination builds DAGs (directed acyclic graphs) from expression trees and forests.

```
A[i+1] = B[i+1]          →    tmp = i+1;
                              A[tmp] = B[tmp];

D = D + C * B;           →    T = C * B;
A = D + C * B;                D = D + T;
                              A = D + T;
```

Note: Redifinition of `D = D+T` is *not* a common subexpression (does not refer to the same *value*).

Local Optimization (cont.)

Reduction in operator strength

Replace an expensive operation by a cheaper one (on the given target machine)

```
x = y ^ 2.0;          →   x = y * y;
x = 2.0 * y;          →   x = y + y;
x = 8 * y;            →   x = y << 3;
(S1+S2).length()      →   S1.length() + S2.length()
```

Some Other Machine-Independent Optimizations

Array-references

- `C = A[I,J] + A[I,J+1]`
- Elements are beside each other in memory.
  Ought to be “give me the next element”.

Inline expansion of code for small routines

`x = sqr(y)` $\rightarrow$ `x = y * y`

Short-circuit evaluation of tests

```
(a > b) and (c-b < k) and /* ... */
```

If the first expression is false, the rest of the expressions do not need to be evaluated if they do not contain side-effects (or if the language stipulates that the operator must perform short-circuit evaluation)

More examples of machine-independent optimization

See for example the OpenModelica Compiler (`https://github.com/OpenModelica/OpenModelica/blob/master/OMCompiler/Compiler/FrontEnd/ExpressionSimplify.mo`) optimizing abstract syntax trees:

```
// listAppend(e1,{}) => e1 is O(1) instead of O(len(e1))
case DAE.CALL(path=Absyn.IDENT("listAppend"),
              expLst={e1,DAE.LIST(valList={})})
     then e1;
// atan2(y,0) = sign(y)*pi/2
case (DAE.CALL(path=Absyn.IDENT("atan2"),expLst={e1,e2}))
guard Expression.isZero(e2)
algorithm
  e := Expression.makePureBuiltinCall(
    "sign", {e1}, DAE.T_REAL_DEFAULT);
then DAE.BINARY(
  DAE.RCONST(1.570796326794896619231321691639751442),
  DAE.MUL(DAE.T_REAL_DEFAULT),
e);
```

Exercise 1:
Draw a basic block control flow graph (BB CFG)

Part III

Loop Optimization

Loop Optimization

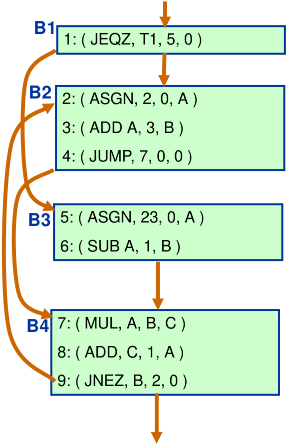
Minimize time spent in a loop

- Time of loop body
- Data locality
- Loop control overhead

What is a loop?

- A **strongly connected component** (SCC) in the control flow graph resp. basic block graph
- SCC strongly connected, i.e., all nodes can be reached from all others
- Has a **unique** entry point

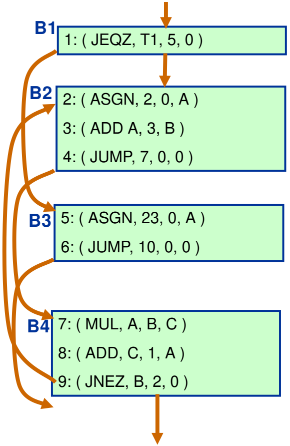
Ex. { B2, B4 } is an SCC with 2 entry points $\rightarrow$ not a loop in the strict sense (spaghetti code).



Loop Example

- Removed the 2nd entry point

Ex. { B2, B4 } is an SCC with 1 entry point $\rightarrow$ is a loop.



Loop Optimization Example: Loop-invariant code hoisting

Move loop-invariant code out of the loop

```
for (i=0; i<10; i++) {
  a[i] = b[i] + c / d;
}
```

$\longrightarrow$

```
tmp = c / d;
for (i=0; i<10; i++) {
  a[i] = b[i] + tmp;
}
```

Loop Optimization Example: Loop unrolling

- Reduces loop overhead (number of tests/branches) by duplicating loop body. Faster code, but code size expands.
- In general case, e.g. when odd number loop limit – make it even by handling 1st iteration in an if-statement before loop .

```
i = 1;
while (i <= 50) {
  a[i] = b[i];
  i = i + 1;
}
```

$\rightarrow$

```
i = 1;
while (i <= 50) {
  a[i] = b[i];
  i = i + 1;
  a[i] = b[i];
  i = i + 1;
}
```

Loop Optimization Example: Loop interchange

To improve data locality, change order of inner/outer loop to make data access sequential; this makes accesses within a cached block (reduce cache misses / page faults).

```
for (i=0;  i<N;  i++)
   for (j=0;  j<M;  j++)
      a[ j ][ i ] = 0.0 ;
```

→

```
for (j=0;  j<M;  j++)
   for (i=0;  i<N;  i++)
      a[ j ][ i ] = 0.0 ;
```

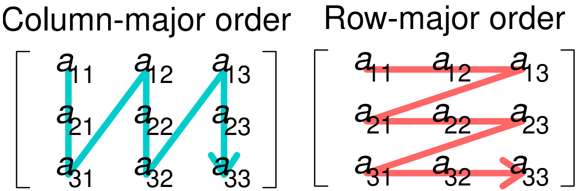


Figure: By Cmglee – Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=65107030

Loop Optimization Example: Loop fusion

- Merge loops with identical headers
- To improve data locality and reduce number of tests/branches

```
for (i=0; i<N; i++)
   a[ i ] = /* ... */;
for (i=0; i<N; i++)
   f(a[ i ]);
```

$\rightarrow$

```
for (i=0; i<N; i++) {
   a[ i ] = /* ... */;
   f(a[ i ]);
}
```

Loop Optimization Example: Loop collapsing

- Flatten a multi-dimensional loop nest
- May simplify addressing (relies on consecutive array layout in memory)
- Loss of structure

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    f( a[ i ][ j ] );
```

$\rightarrow$

```
for ( ij=0; ij<M*N; ij++) {
  f( a[ ij ] );
}
```

Exercise 2:
Draw CFG and find possible loops

Part IV

Global Optimization
(within a single procedure)

Global Optimization

- More optimization can be achieved if a *whole procedure* (= global optimization) is analyzed
  (Whole program analysis = interprocedural analysis)
  - Global optimization is done within a single procedure
  - Needs data flow analysis
- Example global optimizations:
  - Remove variables which are never referenced.
  - Avoid calculations whose results are not used.
  - Remove code which is not called or reachable (i.e., *dead code elimination*).
  - Code motion
  - Find uninitialized variables

Data Flow Analysis (1)

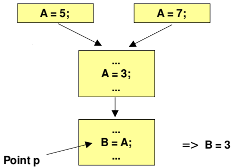
Definition `A = 5` (`A` *is defined*)

Use `D = A*C` (`A` *is used*)

Data is flowing from definition to use

The flow analysis is performed in two phases, forwards and backwards.

Forward analysis Finds *Reaching definitions*. Which definitions apply at a point *p* in a flow graph?



Data Flow Analysis (2), Forward – *Available expressions*

Used to eliminate common subexpressions **over block boundaries**.

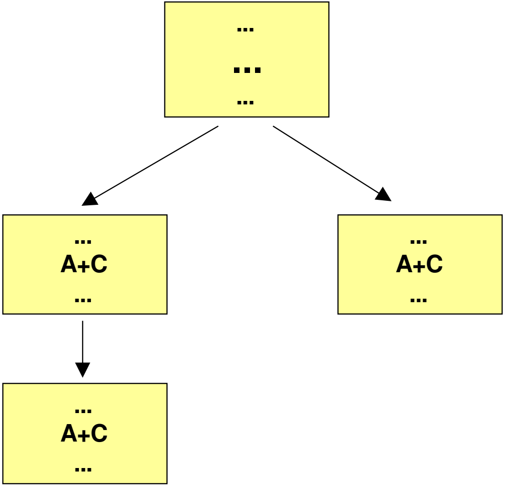


Figure: An available expression A+C

Data Flow Analysis (3), Backward – *Live variables*

A variable v is *live* at point *p* if its value is used after *p* before any new definition of v is made. For example if variable A is in a register and is dead (not live, will not be referenced) the register can be released.

```
// ...
v = A;
// ...
c = v;
// ...
```

We know there is a definition of v at the highlighted line, but is there another definition of v before it is used?

```
// ...
v = A;
x = 35;
c = v;
// ...
```

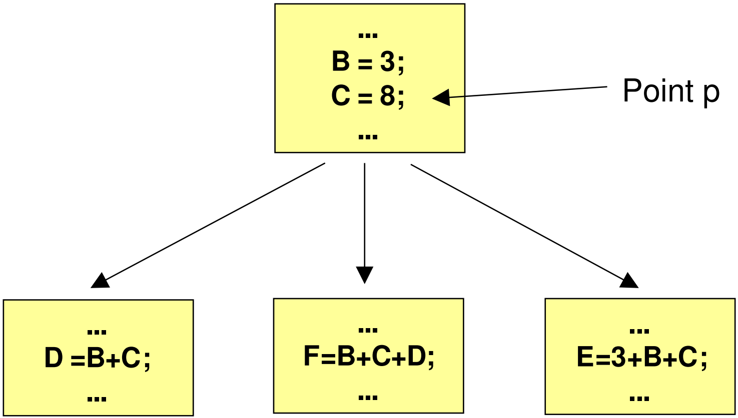
v is *live* at the highlighted line since there is no new definition of v in-between (and v is used after this line).

```
// ...
v = A;
// ...
v = 999;
c = v;
// ...
```

The first v is *not live* at the highlighted line, since v was redefined before the next use.

Data Flow Analysis (4), Backward – Very-Busy or Anticipated expressions

An expression B+C is very-busy at point p if all paths leading from the point p eventually compute the value of the expression B+C from the values of B and C available at p.



Remarks

- Need to analyze data dependencies to make sure that transformations do not change the semantics of the code
- Global transformations need control and data flow analysis (within a procedure – **intra**procedural)
- **Inter**procedural analysis deals with the whole program
- Covered in more detail in courses
  - (Discontinued) TDDC86 Compiler optimizations and code generation
  - (9 hp Ph.D. student level) DF00100 Advanced Compiler Construction

Part V

Target Optimizations
on
Target Binary Code

Target-level Optimizations

Often included in main code generation step of back end:

- Register allocation
  - Better register use $\rightarrow$ less memory accesses, less energy
- Instruction selection
  - Choice of more powerful instructions for same code $\rightarrow$ faster + shorter code, possibly using fewer registers too
- Instruction scheduling $\rightarrow$ reorder instructions for faster code
- Branch prediction (e.g. guided by profiling data)
- Predication of conditionally executed code

$\rightarrow$ See lecture on code generation for RISC and superscalar processors (TDDB44)

$\rightarrow$ (Much more in the discontinued course)

Postpass Optimizations (1)

"postpass" = done after target code generation

Peephole optimization

- Very simple and limited
- Cleanup after code generation or other transformation
- Use a window of very few consecutive instructions
- Could be done in hardware by superscalar processors

```
; ...
LD    A, R0
ADD   1, R0
ST   R0,  A
LD    A, R0
; ...
```

```
; ...
INC   A, R0
; (removed)
; (removed)
LD    A, R0
; ...
```

```
; ...
INC   A, R0
; (removed)
; (removed)
LD    A, R0
; ...
```

Could not remove LD instruction since the peephole context is too small (3 instructions). The INC instruction which also loaded A is not visible.

Postpass Optimizations (1)

"postpass" = done after target code generation

Peephole optimization

- Very simple and limited
- Cleanup after code generation or other transformation
- Use a window of very few consecutive instructions
- Could be done in hardware by superscalar processors

```
; ...
LD    A, R0
ADD   1, R0
ST   R0,  A
LD    A, R0
; ...
```

```
; ...
LD    A, R0
ADD   1, R0
ST    R0, A
; (load removed)
; ...
```

```
; ...
LD    A, R0
ADD   1, R0
ST    R0, A
; (load removed)
; ...
```

Greedy peephole optimization (as on previous slide) may miss a more profitable alternative optimization (here, removal of a load instruction)

Postpass Optimizations (2) – Postpass instruction (re)scheduling

- Reconstruct control flow, data dependences from binary code
- Reorder instructions to improve execution time
- Works even if no source code available
- Can be retargetable (parameterized in processor architecture specification)
- E.g., aiPop™ tool by AbsInt GmbH, Saarbrücken

References

Beniamino Di Martino and Christoph Kessler. "Two program comprehension tools for automatic parallelization". In: *IEEE Concurrency* 8.1 (2000), pp. 37–47. DOI: 10.1109/4434.824311.

Christoph Kessler. "Pattern-Driven Automatic Parallelization". In: *Sci. Program.* 5.3 (Aug. 1996), pp. 251–274. DOI: 10.1155/1996/406379.

www.liu.se