TDDD16 Compilers and Interpreters

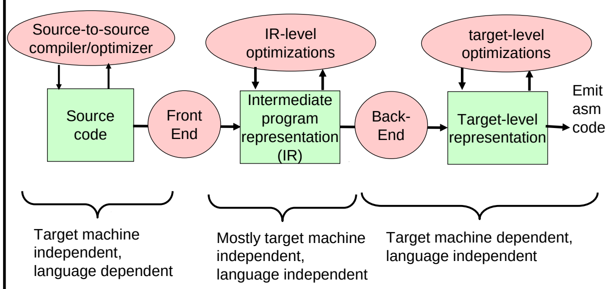
TDDB44 Compiler Construction

Code Optimization

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2008.

Code Optimization – Overview

Goal: Faster code and/or smaller code 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 on 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 compiling)

Disadvantages of Compiler Optimizations

- Debugging made difficult
  - Code moves around or disappears
  - Important to be able to switch off optimization
- Increases compilation time
- May even affect program semantics
  - A = B*C – D + E ➔ A = B*C + E – D

    may lead to overflow if B*C+E is a too large 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 ➔ Quick sort
- Poor algorithms are the main source of inefficiency but difficult to optimize
- Needs pattern matching, e.g. [K.’96] [di Martino, K. 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

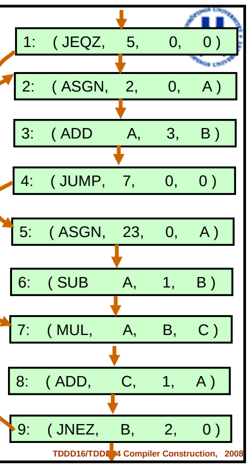
- A **basic block** is a sequence of textually consecutive operations (e.g. quadruples) that contains no branches (except perhaps its last operation) and no branch targets (except perhaps its first operation).
  - Always executed in same order from entry to exit
  - A.k.a. *straight-line code*

```
1:  ( JEQZ,   5,    0,   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

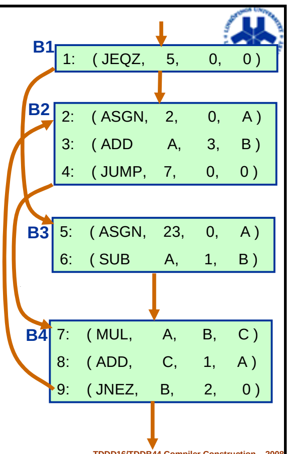
```
1:  ( JEQZ,   5,    0,   0 )   B1
-------------------------------
2:  ( ASGN,   2,    0,   A )   B2
3:  ( ADD     A,    3,   B )
4:  ( JUMP,   7,    0,   0 )
-------------------------------
5:  ( ASGN,   23,   0,   A )   B3
6:  ( SUB     A,    1,   B )
-------------------------------
7:  ( MUL, 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



Basic Block Control Flow Graph

- Nodes: basic blocks
- Edges: control flow transitions

```
1:  ( JEQZ,   5,    0,   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 )
```



Local Optimization

(within single Basic Block)

Local Optimization

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

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

→

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


Local Optimization (cont.)

- **Elimination of common subexpressions**

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

→

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

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

→

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

NB: Redefinition of D → D+T is *not* a common subexpression! (does not refer to the same *value*)

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

Local Optimization (cont.)

- **Reduction in operator strength**
  - Replace an expensive operation by a cheaper one (on the given target machine)

Example: x = y ** 2 → x = y * y

Example: x = 2.0 * y → x = y + y

Example: Concatenation in Snobol

L := Length(S1 || S2) → L := Length(S1) + Length(S2)

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*".
- Expand the code for small routines
  - `x = sqr(y)` ⇒ `x := y * y`
- Short-circuit evaluation of tests
  - `while (a > b) and (c-b < k) and ...`
  - If **false** the rest does not need to be evaluated

Loop Optimization

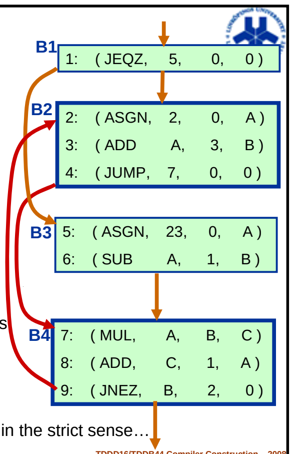
Loop Optimization

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
- Example: { B2, B4 } is an SCC with 2 entries → not a loop in the strict sense…



Loop Optimization Examples (1)

- **Loop-invariant code hoisting**
  - **Move loop-invariant code out of the loop**
  - Example:

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

⇒

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


Loop Optimization Examples (2)

- **Loop unrolling**
  - Reduces loop overhead (number of tests/branches) by duplicating loop body. However, 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
  - Example:

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

⇒

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

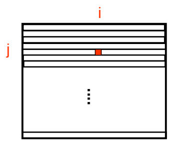

Loop Optimization Examples (3)

- **Loop interchange**
  - To improve data locality, inner loop within a cache block (reduce cache misses / page faults)
  - Example:

```
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 ;
```



Loop Optimization Examples (4)

- **Loop fusion**
  - Merge loops with identical headers
  - To improve data locality and reduce number of tests/branches
  - Example:

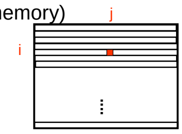
```
for (i=0;  i<N;  i++)
   a[ i ] = … ;
for (i=0;  i<N;  i++)
   … = … a[ i ] … ;
```

⇒

```
for (i=0;  i<N;  i++) {
   a[ i ] = … ;
   … = … a[ i ] … ;
}
```


Loop Optimization Examples (5)

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




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

⇒

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


Global Optimization

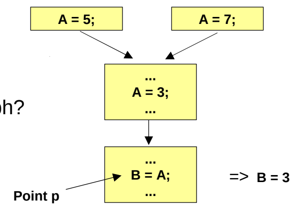
(within a single procedure)

Global Optimization

- More optimization can be achieved if a *whole procedure* is analyzed
  (Whole program analysis is called 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)

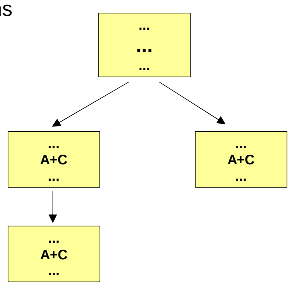
- Concepts: Data is flowing from definition to use
  - *Definition*: A := 5 *A is defined*
  - *Use*: B := A*C *A is used*
- The flow analysis is performed in two phases, forwards and backwards
- **Forward analysis:**
  - *Reaching definitions*
  - Which definitions apply at a point *p* in a flow graph?



Data Flow Analysis (2), Forward

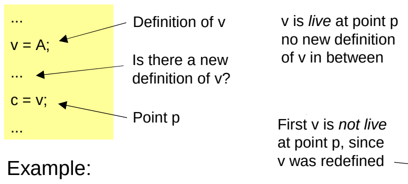
- *Available expressions*
  - Used to eliminate common subexpressions over block boundaries

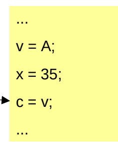
Example:
An available expression
A+C

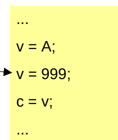


Data Flow Analysis (3), Backward

- *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.



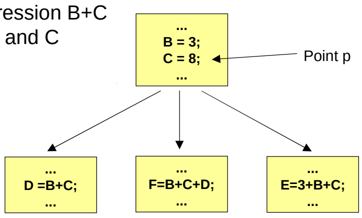




- Example:
  - If variable A is in a register and is dead (not live, will not be referenced) the register can be released

Data Flow Analysis (4), Backward

- *Very-Busy Expressions* 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 dependences** 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 TDDC86 Compiler optimizations and code generation

Target Optimizations on Target Binary Code

Target-level Optimizations

Often included in main code generation step of back end:

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

→ See lecture on code generation for RISC and superscalar processors (TDDB44)
→ Much more in TDDC86 Compiler optimizations and code generation

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     INC  A         INC  A, R0
ADD  1, R0     (removed)      (removed)
ST   R0, A     (removed       (removed)
LD   A, R0     LD   A, R0     LD   A, R0
…              …              …
```


Postpass Optimizations (2)

- **”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…

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

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


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