TDDB29 Compilers and Interpreters

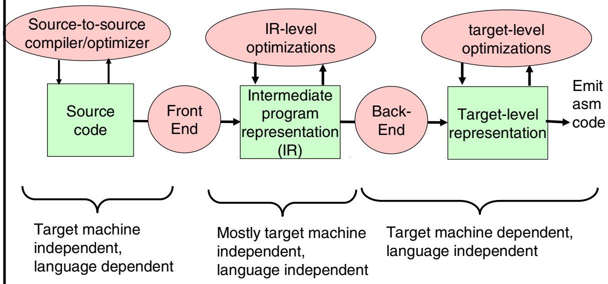
TDDB44 Compiler Construction

Intermediate Code Optimization

Christoph Kessler, IDA,
Linköpings universitet, 2007.

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

Optimization examples

Source-level optimization - 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 - mostly target machine independent

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

Target-level code optimization - target machine dependent

- Instruction selection, register allocation, instruction scheduling, predication
- Peephole optimization

Basic block

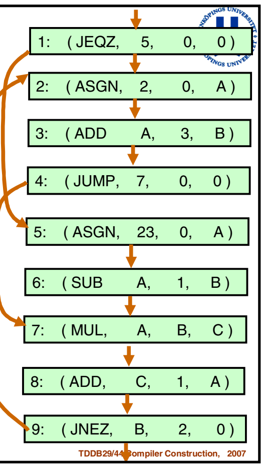
- A **basic block** is a sequence of textually consecutive operations (e.g. quadruples)
  that contains no branches (except perhaps its last operation)
  and no branch targets (except perhaps its first operation).
  - Always executed in same order from entry to exit
  - A.k.a. *straight-line code*

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)
- Edges: control flow transitions

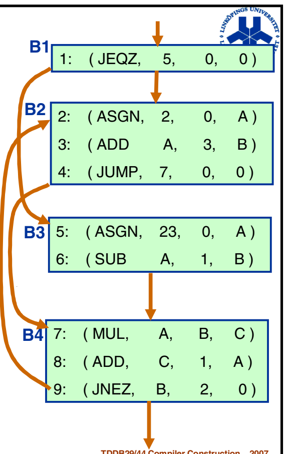
```
1:  ( JEQZ,   5,    0,   0 )   B1
2:  ( ASGN,   2,    0,   A )   B2
3:  ( ADD     A,    3,   B )
4:  ( JUMP,   7,    0,   0 )
5:  ( ASGN,   23,   0,   A )   B3
6:  ( SUB     A,    1,   B )
7:  ( MUL,    A,    B,   C )   B4
8:  ( ADD,    C,    1,   A )
9:  ( JNEZ,   B,    2,   0 )
```



Basic block 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 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)

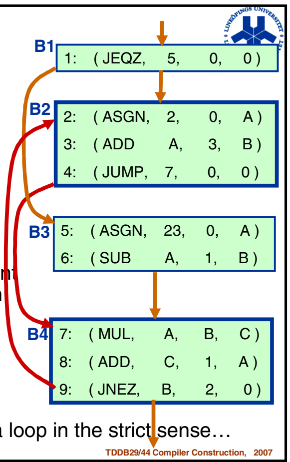
Loop optimization

Minimize time spent in a loop

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

What is a **loop**?

- A strongly connected component (cycle) in the control flow graph resp. basic block graph
- that has a **unique** entry
- Example: { B2, B4 }
  is a SCC with 2 entries → not a loop in the strict sense…



Loop optimization examples (1)

- **Loop-invariant code hoisting**
  - 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 branches)
  - 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;
}
```

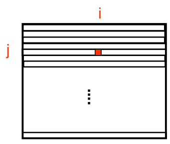

Loop optimization examples (3)

- **Loop interchange**
  - To improve data locality (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 number of branches
  - Example:

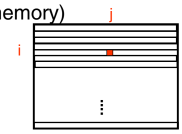
```
for (i=0;  i<N;  i++)
   a[ i ] = … ;
for (i=0;  i<N;  i++)
   … = … a[ i ] … ;
```

⇒

```
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 ] … ;
}
```


Remarks

- Need to analyze **data dependences** to make sure that transformations do not change the semantics of the code
- **Global transformations** (within a procedure – intraprocedural) need control and data flow analysis
- **Interprocedural analysis** deals with the whole program
- Will be covered in TDDC86 Compiler optimizations and code generation

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