Intermediate Code Optimization

Code optimization – overview

Goal: Faster code and/or smaller code and/or low energy consumption

Source-to-source compiler/optimizer

Intermediate program representation (IR)

Target-level representation

Emit asm code

Target machine independent, language dependent

Mostly target machine independent, language independent

Target machine dependent, language independent

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
  - \( A = B \cdot C - D + E \) \( \Rightarrow \) \( A = B \cdot C + E - D \) 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 \( \Rightarrow \) 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) B3
4: (JUMP, 7, 0, 0)
5: (ASGN, 23, 0, A) B4
6: (SUB, A, 1, B) B5
7: (MUL, A, 1, C) B6
8: (ADD, C, 1, A) B7
9: (JNEZ, B, 2, 0) B8
10: (ADD, C, 0, 1) B9
11: (SUB, A, 1, B) B10
12: (MUL, A, 1, C) B11
13: (ADD, C, 1, A) B12
14: (JNEZ, B, 2, 0) B13
15: (ADD, C, 1, A) B14
16: (SUB, A, 1, B) B15
Control flow graph
- Nodes: primitive operations (e.g., quadruples)
- Edges: control flow transitions

Basic block graph
- Nodes: basic blocks
- Edges: control flow transitions

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

Local optimization (cont.)
- Elimination of common subexpressions
  - Temporary variable introduction

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

Loop optimization
- Minimize time spent in a loop
  - Time of loop body
  - Data locality
  - Loop control overhead

Example: Concatenation in Snobol
  - Snobol: Concatenation (S1 || S2)

Examples:
- x = y ** 2 → x = y * y
- x = 2.0 * y → x = y + y
- L := Length(S1 || S2) → L := Length(S1) + Length(S2)
Loop optimization examples (1)

- Loop-invariant code hoisting

Example:

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

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

Loop optimization examples (2)

- Loop unrolling

Reduces loop overhead (number of branches)

Example:

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

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

Loop optimization examples (3)

- Loop interchange

To improve data locality (reduce cache misses / page faults)

Example:

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

```c
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:

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

```c
for (i=0; i<N; 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:

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

```c
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 \(\rightarrow\) 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

\[\text{See lecture on code generation for RISC and superscalar processors (TDDB44) }\]
\[\text{Much more in TDCC86 Compiler optimizations and code generation}\]

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

Postpass optimizations (2)

- "postpass" = done after target code generation

  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

Postpass instruction (re)scheduling

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