**Code Optimization**

**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 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
  - $A = B \times C - D + E \Rightarrow A = B \times C + E - D$
  - may lead to overflow if $B \times C + E$ 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 $\Rightarrow$ 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

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

Control Flow Graph

- Nodes: primitive operations (e.g. quadruples), or basic blocks.
- Edges: control flow transitions

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 = 1 * 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];
```

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

NB: Redefinition 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)
- Example: 
  - $x = y^2 \rightarrow x = y \cdot y$
  - $x = 2.0 \cdot y \rightarrow x = y + y$
  - Concatenation in Snobol string language
    - $L := \text{Length}(S1 || S2) \rightarrow L := \text{Length}(S1) + \text{Length}(S2)$

Some Other Machine-Independent Optimizations
- **Array-references**
  - Elements are beside each other in memory. Ought to be "give me the next element".
- **Inline expansion of code for small routines**
  - $x = \text{sqr}(y) \Rightarrow x = y \cdot y$
- **Short-circuit evaluation of tests**
  - While $(a > b) \text{ and } (c - b < k) \text{ and } ...$
  - If false the rest does not need to be evaluated

More examples of machine-independent optimization
- See for example the OpenModelica Compiler ([https://github.com/OpenModelica/OMCompiler/blob/master/Compiler/FrontEnd/ExpressionSimplify.mo](https://github.com/OpenModelica/OMCompiler/blob/master/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("atan2", e1, DAE.T_REAL_DEFAULT); 
    then DAE.BINARY(DAE.RCONST(1.570796326794896619231321651659751442),
                   DAE.MUL(DAE.T_REAL_DEFAULT, e));
```
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

Example: \{ B2, B4 \} is an SCC with 2 entry points → not a loop in the strict sense...

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

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. 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
  - 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;
    }

Loop Optimization Examples (3)

- Loop interchange
  - To improve data locality, inner loop data access 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;

Faster with consecutive data accesses for inner loop

---

Loop Example

- Removed the 2nd entry point from the previous example
- Example: \{ B2, B4 \} is an SCC with 1 entry points → is a loop!

1: (JEQZ, A, 0, 0)
2: (ASGN, 2, 0, A)
3: (ADD, A, 3, B)
4: (JUMP, 7, 0, 0)
5: (ASGN, 23, 0, A)
6: (JUMP, 10, 0, 0)
7: (MUL, A, B, C)
8: (ADD, C, 1, A)
9: (JNEZ, B, 2, 0)
Loop Optimization Examples (4)

- **Loop fusion**
  - Merge loops with identical headers
  - To improve data locality and reduce number of tests/branches
  - Example:
    ```c
    for (i=0; i<N; i++)
      for (j=0; j<N; j++)
        a[i][j] = ...
    ```
    ```c
    for (i=0; i<N; i++)
      for (j=0; j<N; j++)
        ...
        a[i][j] ...
    ```

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

Exercise 2:
Draw CFG and find possible loops

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:**
  - **Definition:** $A = 5$  
    $A$ is defined
  - **Use:** $B = A \times C$  
    $A$ 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?

Point $p$

- $A = 3$
- $A = 7$
- $B = A$
- $B = 3$
- $B = 2$
- $A \rightarrow B$  
- $B \rightarrow 3$
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.

$$v = A;$$
$$x = 35;$$
$$c = v;$$

First $$v$$ is not live at point $$p$$, since $$v$$ was redefined

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 $$p$$.

$$D = B + C;$$
$$E = 3 + B + C;$$

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 – intraprocedural)
- Interprocedural analysis deals with the whole program

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
ADD 1, R0
ST R0, A
LD A, R0

INC A, R0
(removed)
LD A, R0

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

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…

LD A, R0
ADD 1, R0
ST R0, A
LD A, R0

LD A, R0
ADD 1, R0
ST R0, A
LD A, R0

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