FORMAL LANGUAGES AND COMPILERS

Code generation and optimization

Floriano Scioscia
• The final phase of a compiler is code generation and optimization. It takes in input the intermediate representation (IR) code, along with information stored into the symbol table, and produces a semantically equivalent target program.

• The most advanced compilers execute more than one processing step on the IR. This is because it is easier to apply optimization algorithms one at a time and also because the input of an optimization may depend on the output of another optimization.

• Furthermore, this design for the code generation process facilitates the creation of a single compiler for more architectures, because only the last code generation step requires changes depending on the target machine.
• Requirements of a code generator are very strict:
  – **preserving the meaning** (semantics) of the source program;
  – **using resources** of the target machine **effectively**;
  – **being efficient** itself.

• The challenge stems from the fact that the **mathematical problem** of generating an optimal target code for a given program is **undecidable**: many sub-problems of code generation are computationally intractable, such as register allocation.

• In practice one must resort to **heuristic techniques** to produce good, though non optimal, code. Fortunately, **heuristics have been improved** very much in the course of time, ensuring today a well-designed code generator can produce code many times more efficient than a naïve one.

• Compilers needing to produce particularly efficient target programs include an **optimization phase already on the IR code**.
Code optimization: when is it done?

- Optimization is not a single process or procedure. It is rather a set of strategies to improve programs, which can be applied in various stages of compilation.

- There is little room for optimization in the front end.
  - Precomputation of arithmetic expressions, simplification of logical expressions.
  - The work done here depends on parsing, i.e. it is syntax-directed.
  - Semantic analysis starts collecting information which can help optimization.

- More substantial optimization occurs in the back end
  - Adapt the code to use actually available registers and to exploit as much as possible the functional units of the target architecture.

- Typical back-end optimizations:
  - Parallel instruction execution (pipeline);
  - Instruction reordering (to reduce latency);
  - Change data allocation (registers, cache);
  - Reduce energy consumption.
• In addition to the basic conversion from IR to a linear sequence of machine instructions, a typical code generator seeks to improve the generated code: using **faster instructions from the instruction set**, reducing the **number of instructions**, exploiting **all available registers** and avoiding **redundant computations**.

• **Conventional two-pass compiler**

![Diagram](source-code-front-end-back-end-target-code-error-messages)

• Typically the front end has computational complexity in $O(n)$ or in $O(n \log n)$, while the back end is NP-complete.
• In compilers case, the noun «optimization» is a clear misnomer!

• Even for simple programs, it is impossible to prove a particular version is optimal (i.e. the absolute best) for a specific computer, and even more so it is to create a compiler to generate that optimal version.

• Properly speaking:
  – the goal is to produce «improved» code, not «optimal» code;
  – worse code may be produced sometimes;
  – realistic speed-up usually falls between 1,01 and 4 times (sometimes more, in specific cases).
• An **optimization** is a program transformation from which to expect:

  either

  1. a decrease in **running time**,

  or

  2. a decrease in **memory requirements**.
• Classical architecture-independent optimizations
  – Reduce the number of instructions → reduce the cost of instructions

• Propagation of constants:
  
  $$X := 3; \quad \Rightarrow \quad X := 3;$$

  $$A := B + X; \quad \Rightarrow \quad A := B + 3;$$

  avoiding a memory access.

• Elimination of shared sub-expressions:

  $$A := B \times C; \quad T := B \times C;$$

  $$D := B \times C; \quad \Rightarrow \quad A := T;$$

  $$D := T;$$
• Computer architectures have evolved very much since the 1980s.

• Changes in architectures require changes in compilers:
  – New features introduce new issues;
  – Cost changes leads to change heuristic weighted evaluations;
  – Known solutions must be re-engineered.

• Architecture-dependent optimizations can improve performance significantly:
  – Register allocation: the goal is to minimize CPU wait times for data (typically, about 60% of the overall running time of a program is spent waiting for data);
  – Instruction selection;
  – Instruction scheduling.
The middle end is devoted to improve the code.

- Analyze and rewrite (transform) the intermediate representation of the code.
- The decrease in running time of compiled code is usually assumed as the main goal.
- Also memory and energy usage can be improved.
- Improvements must preserve the «meaning» of the code in a provable way.
A code generator (be it for IR or for machine code) always includes three main tasks:

- **instruction selection**: mapping the program from IR to instruction sequences executable by the target machine; the complexity of such mapping is due to factors such as the level of the IR, the nature of the target machine instruction set and the desired quality of generated code.

- **register allocation and assignment**, a key requirement of code generation: registers are the memory unit with fastest access in an architecture, but unfortunately their number is constrained with respect to program needs. Therefore values which cannot stay in registers must be stored in memory and processed with different instructions, which use more space and are very much slower.

- **instruction ordering**, another NP-complete problem, implying the choice of the order in which computations will be executed: some computations, in fact, require fewer registers, so impacting code efficiency.
Instruction selection (1/3)

• IR level
  – If the IR is of high level, the code generator can translate each IR statement in a sequence of machine instructions through predefined encoding schemes (code templates) which, however, need a further optimization phase.
  – On the other hand, if the IR reflects some of the low level details of the target architecture, then the code generator can exploit this information to generate more efficient sequences of machine instructions.
• If one does not care the efficiency of the target program, instruction selection can be direct: for example, for each three-address instruction a predefined target code template can be used.

• For instance, a statement such as

\[ x = y + z \]

is equivalent to a three-address instruction of the form

\[ \text{op } y, z, x \]

where \( x, y \) and \( z \) are statically allocated.

It can be translated into this sequence of machine instructions

- \( \text{LD } R0, y \) – load \( y \) into register \( R0 \)
- \( \text{ADD } R0, R0 z \) – add \( z \) to \( R0 \)
- \( \text{ST } x, R0 \) – store \( R0 \) into \( x \)

• This strategy often causes redundancy of load and store operations.
Instruction selection (3/3)

- With template matching, the instruction sequence
  \[
  \begin{align*}
  x &= y + z \\
  w &= x + v
  \end{align*}
  \]
  is translated to
  \[
  \begin{align*}
  &LD \ R0, \ y \\
  &ADD \ R0, \ R0 \ z \\
  &ST \ x, \ R0 \\
  &LD \ R0, \ x \\
  &ADD \ R0, \ R0 \ v \\
  &ST \ w, \ R0
  \end{align*}
  \]

- As easily noticed, the fourth instruction is redundant, because it loads a value stored by the instruction immediately before it. Moreover, the third instruction is redundant too, if the value of \( x \) is not used afterwards.

- Analogously, if the target machine has an increment instruction \texttt{INC}, the three-address instruction \( a = a + 1 \) can be translated more effectively with the single instruction \texttt{INC a} than with the sequence
  \[
  \begin{align*}
  &LD \ R0, \ a \\
  &ADD \ R0, \ R0 \ #1 \\
  &ST \ a, \ R0
  \end{align*}
  \]

- **Strength Reduction**: replace a (computationally) costly instruction with an equivalent cheaper one. For example, a division can be replaced with a multiplication of the reciprocal number.
Register allocation and assignment

• Register usage implies two kinds of problems:
  – register allocation: selecting the set of variables which will stay in registers in the various sections of the program;
  – register assignment: selecting the particular register a variable will stay in; it is an NP-complete problem, difficult to solve even for the simplest target machines, due to the “interference” of the operating system.
Instruction ordering

• Instruction ordering is based on the division of the IR code into a graph of elementary blocks.

• An elementary block is defined as a sequence of instructions which will be always executed consecutively without jumps.

• This technique enables several optimizations:
  – removal of unreachable code (dead code);
  – evaluation of alive variables (the ones which may be used again) and dead ones (which surely will not be used anymore): alive variables must be allocated to registers, dead ones can be stored to memory;
  – assignment of registers to the variables which produce the highest saving of memory accesses (i.e. the most used ones);
  – checking that the meaning of the code is not altered.

• In order to implement these optimization strategies on graphs, pattern matching, dynamic programming or data flow analysis techniques can be used.
Modern compilers

- Modern compilers are usually very large and sophisticated software systems, often developed in the course of years or decades.
- A full industry-standard compiler can have several million lines of code.
- Nobody understands the whole system.
- This very course provides just an introduction to compilers. There is much more depth to fathom.