بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۱۲)

کامپایلر

حسین فلسفین

Loop Unrolling

Consider the two pieces of code:

```
i = 0;                      a[0] = 0;
while (i<4) {               a[1] = 0;
   a[i] = 0;                a[2] = 0;
   i = i + 1                a[3] = 0;
}
```

Many programmers would produce the code on the left (or something like it, probably using a `for` *loop), but the code on the right could be more efficient, in speed and also possibly in memory usage. The code on the right is produced by unrolling the loop on the left and it obviously avoids the loop control variable* `i` *and the code that has to be generated at the start and at the end of the loop. This form of loop unrolling is only applicable when the number of iterations is known at compile time.*

Loop unrolling should always reduce the total number of operations performed by the target machine in executing the code. But if the number of copies of the body of the loop is large because the total number of iterations of the loop is large, then there may well be code space issues. There is a clear tradeoff here of code space against execution time and an appropriate compromise has to be reached. As well as removing the loop control code, loop unrolling may have further benefits by creating opportunities for other optimizations. The expanded code can be optimized using the IR optimization techniques and also there may be more opportunities for pipeline optimizations on the target machine. However, depending on target machine code sizes, it could be that there are performance penalties in unrolling because the code no longer fits in the target machine's instruction cache. These are complex interrelated issues requiring careful analysis.

8.7 Peephole Optimization

While most production compilers produce good code through care-ful instruction selection and register allocation, a few use an alter-native strategy: they generate naive code and then improve the quality of the target code by applying "optimizing" transformations to the target program. The term "optimizing" is somewhat mislead-ing because there is no guarantee that the resulting code is opti-mal under any mathematical measure. Nevertheless, many sim-ple transformations can significantly improve the running time or space requirement of the target program.

A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible. Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.

بخش ۸.۷ از کتاب آهو به موضوع بهینه‌سازی پیپ‌هول اختصاص دارد

The peephole is a small, sliding window on a program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements. In general, *repeated passes over the target code are necessary* to get the maximum benefit. In this section (Aho's book, Section 8.7), we shall give the following examples of program transformations that are characteristic of peephole optimizations:
☞ *Redundant-instruction elimination*
☞ *Flow-of-control optimizations*
☞ *Algebraic simplifications*
☞ *Use of machine idioms*

In a simple compiler with no machine-independent code improvement, a code generator can simply walk the abstract syntax tree, producing naive code, either as output to a file or global list, or as annotations in the tree. However, the result is generally of very poor quality. *A relatively simple way to significantly improve the quality of naive code is to run a peephole optimizer over the target code. A peephole optimizer works by sliding a several-instruction window (a peephole) over the target code, looking for suboptimal patterns of instructions. The set of patterns to look for is heuristic;* *generally one creates patterns to match common suboptimal idioms produced by a particular code generator, or to exploit special instructions available on a given machine. Here are a few examples:*

Constant folding: A naive code generator may produce code that performs calculations at run time that could actually be performed at compile time. A peephole optimizer can often recognize such code. For example:

r2 := 3 × 2   becomes   r2 := 6   ∎

Common subexpression elimination: When the same calculation occurs twice within the peephole of the optimizer, we can often eliminate the second calculation:

r2 := r1 × 5		r4 := r1 × 5
r2 := r2 + r3	becomes	r2 := r4 + r3
r3 := r1 × 5		r3 := r4

Often, as shown here, an extra register will be needed to hold the common value.   ∎

Copy propagation: Even when we cannot tell that the contents of register b will be constant, we may sometimes be able to tell that register b will contain the same value as register a. We can then replace uses of b with uses of a, so long as neither a nor b is modified:

r2 := r1		r2 := r1			r3 := r1 + r1
r3 := r1 + r2	becomes	r3 := r1 + r1	and then		r2 := 5
r2 := 5		r2 := 5			

Performed early in code improvement, copy propagation can serve to decrease register pressure. In a peephole optimizer it may allow us (as in this case, in which the copy of r1 in r2 is dead) to eliminate one or more instructions. ∎

Strength reduction: Numeric identities can sometimes be used to replace a comparatively expensive instruction with a cheaper one. In particular, multiplication or division by powers of two can be replaced with adds or shifts:

r1 := r2 × 2	becomes	r1 := r2 + r2	or		r1 := r2 << 1
r1 := r2 / 2	becomes	r1 := r2 >> 1			

(This last replacement may not be correct when r2 is negative; see Exercise C-17.1.) In a similar vein, algebraic identities allow us to perform simplifications like the following:

r1 := r2 × 0	becomes	r1 := 0

∎

*Elimination of useless instructions:* Instructions like the following can be dropped entirely:

```
r1 := r1 + 0
r1 := r1 × 1
```

■

Because they use a small, fixed-size window, peephole optimizers tend to be very fast: they impose a small, constant amount of overhead per instruction. They are also relatively easy to write and, when used on naive code, can yield dramatic performance improvements.

توضیحی اجمالی دربارهٔ بکاِند کامپایلر
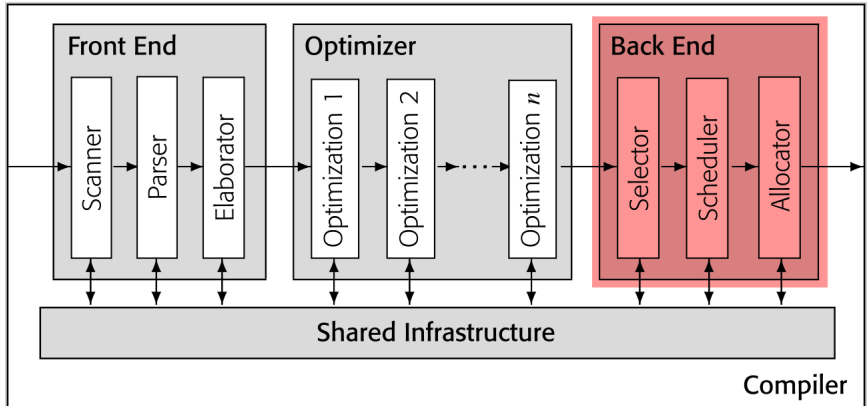
The final phase in our compiler model is the *code generator**. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program, as shown in Fig. 8.1.*
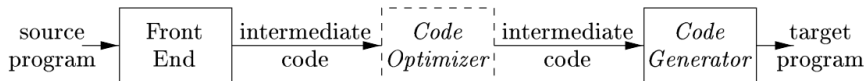

Figure 8.1: Position of code generator

The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.

The Back End

The compiler's back end traverses the IR and emits code for the target machine. The back end solves at least three major problems.

✳ *Instruction Selection*

✳ *Instruction Scheduling*

✳ *Register Allocation*

> *A code generator has **three primary tasks**: instruction selection, register allocation and assignment, and instruction ordering.*

The process of code generation takes the intermediate representation generated by the front-end of the compiler, with or without any optimization performed by the machine-independent optimization phase, and generates code for the target machine.

☞ *Instruction selection* involves choosing appropriate target-machine instructions to implement the IR statements.

☞ *Register allocation and assignment* involves deciding what values to keep in which registers.

☞ *Instruction ordering* involves deciding in what order to schedule the execution of instructions.

While the details are dependent on the specifics of the inter-mediate representation, the target language, and the run-time system, tasks such as instruction selection, register allocation and assignment, and instruction ordering are encountered in the design of almost all code generators.

مجبور به استفاده از هیوریستیک‌ها هستیم

The challenge is that, mathematically, the problem of generating an optimal target program for a given source program is *undecidable**; many of the subproblems encountered in code generation such as register allocation are* *computationally intractable**. In practice, we must be content with* *heuristic techniques* *that generate good, but not necessarily optimal, code. Fortunately, heuristics have matured enough that a carefully designed code generator can produce code that is several times faster than code produced by a naive one.*

مجبور به استفاده از هیوریستیک‌ها هستیم

Each of these three problems is, on its own, a computationally hard problem. While it is not clear how to define optimal instruction selection, the problem of generating the fastest code sequence for a procedure on a given ISA involves considering a huge number of alternatives. Instruction scheduling is NP-complete for a basic block under most realistic execution models; moving to larger regions of code does not simplify the problem. Register allocation is, in its general form, also NP-complete in procedures with control flow.

Instruction Selection

We must convert the IR operations into equivalent operations in the target processor's ISA, a process called instruction selection. The first stage of code generation rewrites the IR operations into target machine operations, a process called instruction selection. Instruction selection maps each IR operation, in its context, into one or more target machine operations.

The instructions available on a given machine, and their encoding in machine language, are referred to as the *instruction set architecture* (ISA). Existing ISAs vary quite a lot, but all include instructions for:

Computation — arithmetic and logical operations, tests, and comparisons on values held in registers (and possibly in memory)

Data movement — loads from memory to registers, stores from registers to memory, copies from one register (or memory location) to another

Control flow — conditional and unconditional branches (gotos), subroutine calls and returns, traps into the operating system

The compiler's front end and optimizer both operate on the code in its IR form. To create executable code, the compiler must rewrite the IR into the processor's instruction set. This process is called instruction selection. Because a typical processor provides multiple ways to express most computations, the selector must choose the best sequence from among multiple implementations.

The aim is to take the IR generated by the front-end and replace the IR instructions by functionally equivalent target machine instructions.

To translate a program from an intermediate representation, such as an abstract syntax tree or a low-level linear code, into executable form, the compiler must map each IR construct into an equivalent construct in the target processor's instruction set. *Depending on the relative levels of abstraction in the IR and the target machine's instruction set architecture (ISA), this translation can involve elaborating details that are hidden in the IR program or it can involve combining multiple IR operations into a single machine instruction.* The specific choices that the compiler makes have a direct impact on the overall efficiency of the compiled code.

Instruction Scheduling

We must select an execution order for the operations, a process called instruction scheduling. The instruction scheduler reorders the operations in the code. It attempts to minimize the number of cycles wasted waiting for operands.

The elapsed running time of a set of operations depends heavily on the order in which those operations are presented for execution. Instruction scheduling reorders the operations in a procedure to reduce the code's execution time. The set of legal orders for the operations is constrained by the need to preserve the flow of values from the original code. The goal is to reduce the number of cycles, start to finish, required to execute the code.

The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. **However, picking a best order in the general case is a difficult NP-complete problem.**

The instruction scheduler takes target machine code as input and it produces code where the instructions are **reordered to make best use of the processor's instruction-level parallelism**. *It must, of course, maintain the semantics of the code.*

Register Allocation

A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but *we usually do not have enough of them to hold all values*. Values not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so *efficient utilization of registers is particularly important*. The use of registers is often subdivided into two subproblems:

1. Register allocation, during which we select the set of variables that will reside in registers at each point in the program.

2. Register assignment, during which we pick the specific register that a variable will reside in.

یکی از مواضع مواجهه با مسائل ماهیتاً سخت

Finding an optimal assignment of registers to variables is diffi-cult, even with single-register machines. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register-usage conven-tions be observed.

Register allocation must decide which values should reside in registers at each point in the code, a process called register allocation. During instruction selection, the compiler deliberately ignores the fact that the target machine has a limited set of physical registers. Instead, it uses virtual registers and assumes that "enough" registers exist. In practice, the earlier stages of compilation may create demand for more physical registers than the hardware provides. The register allocator must map those virtual registers onto physical registers. Thus, the allocator must decide, at each point in the code, which values will reside in physical registers.

The instruction selector relies on the fact that the register allocator will map the virtual registers into physical registers on the target machine.

The code generated by a compiler must make effective use of the limited resources of the target processor. Among the most constrained resources is the set of hardware registers. Register use plays a major role in determining the performance of compiled code. At the same time, register allocation—the process of deciding which values to keep in registers—is a combinatorially hard problem. Most compilers decouple decisions about register allocation from other optimization decisions. Thus, most compilers include a separate pass for register allocation.

Graph Coloring

The live range of a variable starts at its definition (i.e. when it is first initialized) and ends with its last use. If a variable is redefined (i.e. it has a new value assigned to it) a new live range is started.

The analysis of live ranges can be used to form the basis of register allocation. Each virtual register in the code being analyzed has to be allocated to a physical register but there are likely to be many more virtual registers than physical registers.

It is live ranges rather than variables that are central to the register allocation problem.

The register allocation analysis requires the construction of a register interference graph, where the nodes are the virtual registers and an edge between two nodes indicates that the two virtual registers have overlapping live ranges where one of the virtual registers is live at the point at which the other is defined.

If live ranges overlap, the virtual registers cannot share the same physical register. After the interference graph has been constructed, the allocation problem becomes one of coloring the nodes of the graph so that no two connected nodes share the same color. If the graph can be colored using n different colors and there are n or more physical registers available, the allocation problem succeeds. A color represents a particular physical register. This is the graph coloring problem, a well-known problem of graph theory, arising originally from coloring maps so that countries sharing a boundary have different colors.
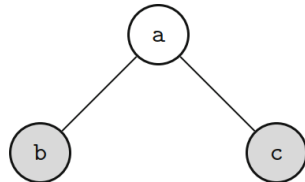
Live ranges and register interference graph

اینجا یکی از مواضعی است که مجبور به استفاده از هیوریستیک‌ها هستیم

The live ranges for the three variables a, b *and* c *are marked on the code on the left. Live range* a *overlaps with both live ranges* b *and* c, *and there is no overlap between live ranges* b *and* c. *This information allows the register interference graph on the right to be drawn. This graph can be colored using just two colors (white for node* a *and gray for nodes* b *and* c). *So two physical registers are required,* b *and* c *can share a register.* *Unfortunately, graph coloring is an NP-complete problem (no polynomial-time solutions are known) and exhaustive search for minimum register solutions is not practically feasible as the size of the interference graph increases to that found from even average programs. Heuristic approaches have to be adopted.*