CHAPTER 1

Introduction

M. A. Ertl

Progress: 5%

Material gathering in progress

Code generation transforms the program from a machine-independent intermediate representation (IR) into some representation of the code for the target machine. Traditionally code generation is divided into the following subtasks:

Instruction selection
: combines the operations of the IR into target machine instructions. Instruction selection aims for selecting a set of instructions with the lowest combined cost; the cost of an instruction is its code size and/or its execution time.

Instruction Scheduling
: orders the instructions, resulting in a total order; among the many possible orders, instruction scheduling usually aims for one that exploits instruction-level parallelism, i.e., it tries to avoid dependences between instructions that are close together. Another (conflicting) goal is to reduce register pressure.

Register allocation
: assigns target-machine registers to IR values (aka live ranges or pseudo-registers) and may insert code to spill registers to and refill registers from memory if there are too many values competing for the limited number of registers. The goals in register allocation are to avoid spill code and moves between registers.

The ordering of these phases determines the quality of the resulting code. Usually the phases are performed in the order given above, with another scheduling pass afterwards to schedule spill code; however, there are phase ordering conflicts between these phases as long as the phases are treated separately: e.g., instruction scheduling for instruction-level parallelism generally increases register pressure, while register allocation reduces the exploitable instruction-level parallelism.

Traditionally, instruction selection is performed on IR trees or at most at the basic block level. Plain instruction scheduling is performed at the basic block level, but

there are techniques for larger control structures: Trace scheduling and superblock scheduling for sequences of basic blocks, and software pipelining for simple loops. Register allocation is usually performed at the function level (global register allocation), but in its traditional form this is an NP-complete problem, whereas register allocation for (already-scheduled) basic blocks can be solved in linear time.

Actually not always (eg in the presence of register constraints or aliasing)

SSA form can be useful when extending work from basic blocks (and superblocks and extended basic blocks[1]) to control structures that include control flow joins. In particular, the three phases of code generation interact with SSA form as follows:

Instruction selection: When extending instruction selection to larger control structures, SSA form has the usual benefit of being a clean representation of data flow; in particular, it avoids false dependencies. Such an extension is the topic of Chapter **??**.

Instruction scheduling: The input to instruction scheduling is a data dependence graph; these data dependences include data flow dependences (represented in SSA form), but also other dependences such as write-after-read dependences (aka antidependences) and write-after-write (aka output) dependences through memory (not represented in SSA form at the low level useful as input for code generation). These other dependencies also have to be taken into account in instruction selection. They have to be maintained during various transformations on the SSA form. As a (small) benefit, there is then no need to generate the data dependence graph from the sequence of instructions.

SSA-based register allocation requires that the conflict graphs for basic blocks are interval graphs; therefore the code has to be in a total order for SSA-based register allocation (whereas the data flow graph represented in SSA form, and the data dependence graph are only partial orders). Running the scheduler before SSA-based register allocation is therefore necessary if the compiler does not have a totally-ordered IR in addition to SSA form; running the scheduler before register allocation is usually a good idea anyway, because register allocation introduces write-after-read dependences between registers which hinder scheduling.

Apart from these considerations, SSA form does not particularly affect instruction scheduling, so there is no chapter in this book about instruction scheduling.

This is strange to see if-conversion as a subcase of scheduling.

If-conversion transforms if-statements into predicated code, where each instruction is executed only when some condition holds; the condition is an additional input to the instruction. I.e., if-conversion converts control flow into data flow. One benefit is that this can avoid branch mispredictions (especially if the condition is hard to predict); another benefit is that the applicability of instruction scheduling techniques for limited control structures (such as software pipelining for simple loops) can be extended (for software pipelining, to loops containing ifs). To achieve the latter benefit even when predicated code is not profitable (because the if-statement is too big and too predictable), one can use reverse if-

[1] Both are single-entry multiple-exit control structures; superblocks are limited to sequences of basic blocks, whereas extended basic blocks allow control-flow trees of basic blocks.

conversion after scheduling. Chapter **??** describes an if-conversion algorithm for SSA form.

Register allocation: The classical view of live ranges leads to arbitrary conflict graphs, for which register allocation is equivalent to graph colouring, which is NP-complete. The SSA-based view introduces register renaming at control-flow joins. As a result, in the conflict-graph view we get chordal graphs, which can be coloured in linear time (after spilling, which is still hard). Or, in the direct view, the control flow that register allocation has to consider is an extended basic block, which can be solved through an extension of the linear basic block scheduling algorithm. This topic is discussed in Chapter **??**.

This citation is to make chapters without citations build without error. Please ignore it: [**?**].

We do not see the advantages of using SSA.

coalescing is still hard. The real contribution is the decoupling at low cost between spill and coalescing/coloring. This leads to efficient and simple scan based allocators also.

I do not understand this last sentence

References