Compiler Principle

Prof. Dongming LU

Apr. 15th, 2024

Content

1. INTRODUCTION

2. LEXICAL ANALYSIS
3. PARSING
4. ABSTRACT SYNTAX
5. SEMANTIC ANALYSIS

6. ACTIVATION RECORD
7. TRANSLATING INTO INTERMEDIATE CODE

8. BASIC BLOCKS AND TRACES

8 Basic Blocks and Traces

8.2 Taming conditional branches

To make the trees easy to translate into machine instructions, they are to be rearrange.

- The transformation in two stages:
  - ✓First, we take the list of canonical trees and form them into *basic blocks*.
  - ✓Then we order the basic blocks into a *trace*.

Basic Blocks

- **Control flow** is the sequencing of instructions in a program.
  - ✓ Ignoring the data values in registers and memory.
  - ✓ Ignoring the arithmetic calculations.

- **Lump together** any sequence of non-branch instructions into a basic block
- **Analyze the control flow** between basic blocks.

Basic Blocks

- A *basic block* is a sequence of statements that is always entered at the beginning and exited at the end
  - ✓ **The first** statement is a LABEL.
  - ✓ **The last** statement is a JUMP or CJUMP.
  - ✓ There are no other LABELs, JUMPs, or CJUMPs.

Basic Blocks

- **The algorithm is quite simple**
  - ✓ The sequence is scanned from beginning to end;
  - ✓ Whenever a LABEL is found, a new block is started (and the previous block is ended);
  - ✓ Whenever a JUMP or CJUMP is found, a block is ended (and the next block is started).
  - ✓ If this leaves any block not ending with a JUMP or CJUMP, then a JUMP to the next block's label is appended to the block.
  - ✓ If any block has been left without a LABEL at the beginning, a new label is invented and stuck there.

Basic Blocks

```
(1)  prod := 0
(2)  i:= 1
(3)  t1 := 4*i
(4)  t2 := a[t1]
(5)  t3 := 4*i
(6)  t4 := b[t3]
(7)  t5 := t2 * t4
(8)  t6 := prod +t5
(9)  prod := t6
(10) t7 := i+1
(11) i := t7
(12) if  i<= 20 goto (3)
```

```
prod := 0
 i:= 1
```

```
t1  := 4*i
t2  := a[t1]
t3  := 4*i
t4  := b[t3]
t5  := t2 * t4
t6  := prod +t5
prod := t6
t7  := i+1
 i := t7
if  i<= 20 goto (3)
```

Traces

- The basic blocks can be arranged in any order, and the result of executing the program will be the same.
  - ✓ Take advantage of this to choose an ordering of the blocks satisfying the condition that each CJUMP is followed by its false label.
  - ✓ Arrange that many of the unconditional JUMPs are immediately followed by their target label.
  - ✓ Allow the deletion of these jumps, which will make the compiled program run a bit faster.

Traces

- A *trace* is a sequence of statements that could be consecutively executed during the execution of the program.
  - ✓ It can include conditional branches.
  - ✓ To make a set of traces that exactly covers the program: Each block must be in exactly one trace.
  - ✓ To have as few traces as possible in our covering set.

Traces

- The idea is to start with some block: the beginning of a trace - and follow a possible execution path - the rest of the trace.
- Suppose block *b*1 ends with a JUMP to *b*4, and *b*4 has a JUMP to *b*6. Then make **the trace *b*1, *b*4, *b*6.**

- Suppose *b*6 ends with a conditional jump CJUMP(*cond,* *b*7, *b*3). Append *b*3 to our trace and continue with the rest of the trace after *b*3. The block *b*7 will be in some other trace.

Traces

- **Algorithm 8.3** (make a set of traces that covers the program and each block is in one trace):

  ✓It starts with some block and follows a chain of jumps,
  ✓Marking each block and appending it to the current trace.

  ✓Eventually it comes to a block whose successors are all marked,
  ✓So it ends the trace and picks an unmarked block to start the next trace.

Finishing Up

- Flatten the ordered list of traces back into one long list of statements
  - ✓ **Any CJUMP immediately followed by its false label we let alone** (there will be many of these).
  - ✓ For any CJUMP followed by its true label, we **switch** the true and false labels and negate the condition.
  - ✓ For any CJUMP(*cond, a, b, lt, lf*) followed by neither label, we **invent a new false label *lf'*** and rewrite the single CJUMP statement as three statements, just to achieve the condition that the CJUMP is followed by its false label:

CJUMP(*cond, a, b, lt, lf'*)
LABEL *lf'*
JUMP(NAME *lf*)

Optimal Traces

- Any frequently executed sequence of instructions (such as the body of a loop) should occupy its own trace.
  - ✓This helps to minimize the number of unconditional jumps.
  - ✓This helps with other kinds of optimizations.
    - ➢ Register allocation
    - ➢ Instruction scheduling
    - ➢ …

Optimal Traces

prologue statements
~~JUMP(NAME *test*)~~
LABEL(*test*)
CJUMP($>$, i, N, *done*, *body*)
LABEL(*body*)
loop body statements
JUMP(NAME *test*)
LABEL(*done*)
epilogue statements

(a)

prologue statements
~~JUMP(NAME *test*)~~
LABEL(*test*)
CJUMP($\leq$, i, N, *body*, *done*)
LABEL(*done*)
epilogue statements
LABEL(*body*)
loop body statements
JUMP(NAME *test*)

(b)

prologue statements
JUMP(NAME *test*)
LABEL(*body*)
loop body statements
~~JUMP(NAME *test*)~~
LABEL(*test*)
CJUMP($\leq$, i, N, *body*, *done*)
LABEL(*done*)
epilogue statements

(c)

- **Which is better?**

The end of Chapter 8(2)