FP8-17: Software Programmable Signal Processing Platform Analysis Exercises for Episode 6

Andrzej Wąsowski

Wednesday, 3 May 2006

I recommend, exercises 6.1, 6.2, 6.4, and 6.6. Exercise 6.7 is likely to be useful in the long run.

Exercise 6.1 The following program (based on Program 17.3 in [Appel]) exhibits opportunities for constant propagation optimization. Compile it with `cl6x`, optimization levels `-O0` and `-O1` . Check whether you can see (relevant) differences in the generated assembly code.

```
int f (void) {
  int a = 5;
  int c = 1;
  while (c <= a)
    c = c + c;
  return c;
}
```

Then insert an assignment `c=0` just before the `return` statement. Try to compile with various optimizations levels. Is `cl6x` able to discover that the whole function could now be rewritten to just `return 0`? How about `gcc`? What is the sequence of analyses and optimizations that would have to be applied to achieve this?

Exercise 6.2 (project task proposal) Take some of your own C code with loops, identify basic expressions, list them and classify each of them whether they can be hoisted or not (the criteria are on p. 418 in [Appel]). Then check the assembly code generated by the compiler, to see what is actually hoisted by it, at various optimization levels.

In `cl6x` hoisting is most likely turned on at optimization level `-O2`. Remember that occasionally the compiler may also hoist expressions that are not present explicitly in the source code (like address computations).

Exercise 6.3* (moderately theoretical) Solve exercise **17.1** p. 408 in [Appel].

Exercise 6.4 Solve exercise **18.1** p. 431 in [Appel].

Exercise 6.5* Solve **18.8** p. 432 in [Appel].

Exercise 6.6 Study the section on **Turning while loops into repeat-until loops** on pp. 418–419 (also Fig. 18.7) in [Appel]. Then go back to exercise 4.2 from episode 4. Can you now explain why the scheme of translation for while loops, studied in 4.2, may be occurring in the compiler? According to section 18.2, what reason is motivating this rendering, other than counting simple instructions, as we discussed two weeks ago?

Exercise 6.7 (project task proposal) This exercises is a continuation of exercise 6.2. Identify the speed critical loop in your project and think how would you optimize it using the techniques presented in the lecture. Ask questions like: what code should be hoisted out? what are the subexpressions that should be reused? what are the opportunities for copy and constant propagation? dead-code elimination? Can loop unrolling help? etc.

Then see whether the code generated by compiler meets your expectations. If the compiler is lacking in certain respects[1], then apply the most essential optimizations one by one on the source code level. It is often sufficient to indicate just one optimization (for example an essential copy/constant propagation) to help the compiler discover possibilities of other.

If you want to extend this task to cover the loop unrolling too, then you may find it useful to study section 2.4.3.4 p. 2-49 in [SPRU198G], The *TMS320C6000 Programmer's Guide*.

[1] It has been formally proven that a perfect compiler will never exist. See Appel, p. 383, *No magic bullet*.