THE UNIVERSITY OF ARIZONA®
DEPARTMENT OF COMPUTER SCIENCE

CSc 553 : Principles of Compilation

Programming Assignment 2 (Machine-independent Optimization)

Start: Fri Oct 6, 2023
<u>Due</u>: 11:59 PM, Fri Oct 20

1. General

This assignment involves the implementation of dataflow analysis and machine-independent optimizations along with their evaluation, as specified below:

(a) For each function in the input program, your compiler should do the following:

1. If the command-line options `-Olocal` and/or `-Oglobal` are specified, construct a control flow graph for the function.
2. If the command-line option `-Olocal` is specified: carry out (at least) the following set of local (i.e., per-basic-block level) optimizations:
    - Some machine-indpendent peephole optimizations.
    - Intra-block copy propagation.

   These are described in more detail below.
3. If the command-line option `-Oglobal` is specified: carry out (at least) the following set of global (i.e., per-function level) optimizations:
    - Global (intra-procedural) dead code elimination.

(b) You should test the effectiveness of your optimizations by testing five C-- programs of your choice, compiled with and without optimization (the four combinations of optimization flags mentioned in Section 2.3 below). Use 'spim -keepstats' to measure the number of instructions executed, and report the results in a file EVALUATION.txt that you upload along with your code.

2. Optimizations

2.1. Local (i.e., basic block level) optimization

Your compiler should implement (at least) the following intra-block optimizations, to be performed if the command-line option `-Olocal` is specified:

1. *Simple machine-independent peephole optimizations*. To determine what patterns to use for your peephole optimizations, examine the code generated by your compiler on several programs of reasonable size, of your choosing, to see how their efficiency could be improved using simple peephole optimizations. In the EVALUATION.txt file reporting your experiments, describe the peephole optimization patterns you used and their impact on the number and type of instructions

executed.

3. *Intra-block copy propagation*. Given a copy instruction

   d: **x = y**

   and a subsequent use *u* of **x** within a basic block, if it can be guaranteed that neither **x** nor **y** is redefined between *d* and *u*, then the reference to **x** at *u* can be replaced by a reference to **y**, as illustrated by the following:

Before	After

x = z
u = x+1
	x = z
	u = z+1 /* use of x replaced by z */

2.2. Global (i.e., function-level) analysis and optimization

Your compiler should implement (at least) the following global (i.e., function-level) analysis and optimizations, to be performed if the command-line option `-Oglobal` is specified.

2.2.1. Intra-procedural liveness analysis

You should carry out liveness analysis at the intra-procedural level. To simplify the problem, you can assume that:

1. All global variables are always live. This means that the liveness analysis does not have to concern itself with reasoning about global variables. (However, it should still be able to handle programs containing global variables.)
2. All elements of an array are live if any of its elements is (i.e., treat the array as a single variable). This means that if *any* element of an array is used at some point in a program, your liveness analysis can treat *all* elements of the array to be live at that point.

Also, you are not *required* to use bit vectors for your implementation of this analysis. It is OK to implement sets using whichever data structure you are most comfortable with.

2.2.2. Intra-Procedural Dead Code Elimination

Dead code refers to code whose results are never used. You are to use the information obtained from your liveness analysis to identify and eliminate dead code. I suggest using the algorithm shown in '**Dead Code Elimination: Algorithm 2**' (slide 23 in slide deck **08. Code Optimization**).

2.3. Order of optimization

The optimizations performed by the `-Olocal` and `-Oglobal` flags should be independent: i.e., it should be possible to specify each of the following combinations to study the impact of the local and global optimizations as well as their interactions:

- neither `-Olocal` nor `-Oglobal` specified (no optimization);
- only `-Olocal`;
- only `-Oglobal`;
- both `-Olocal` and `-Oglobal`. In this case, you should experiment with your compiler to evaluate how the generated code is affected by the order in which the local and global optimizations are performed as well as the number of times they are applied. **Note**: the order in which these flags are specified on the command line should not matter.

3. Execution behavior

Your program will be called **compile**. It will take two optional command-line flags, `-Olocal` and `-Oglobal`, that will control code optimization as discussed in Section 2.3 above. It will take its input from **stdin** and generate all output on **stdout**.

4. Evaluating your optimizations

In addition to the source code for your compiler, you should additionally submit a file EVALUATION.txt that describes your experiments with your optimizations for at least five programs of your choice of reasonable size. At a minimum, this should contain the following:

1. The *cost* and *benefit* of the optimization.

   The *cost* of your optimizations can be evaluated by measuring the additional runtime overhead of the optimizations themselves together with the program analyses needed to support the optimizations. Students sometimes use worst-case asymptotic analysis (big-O) to characterize the cost of an analysis. Unfortunately, this does not give a concrete idea of just how much time the analysis and optimization take on a given program. Please measure the actual execution time cost using something like the `time` command.

   The benefits of optimizations can be estimated by the percentage reductions in the number of different types of instructions executed. See Appendix A.1 for more information.
2. The peephole optimization patterns you used, how you came up with those patterns, and their impact on the number and type of instructions executed.
3. Interactions between local and global optimizations when both `-Olocal` and `-Oglobal` are specified. In this scenario, does the order in which they are specified make a difference to code quality? Does executing them multiple times make a difference? Why or why not?

5. Grading

In order to get credit, your optimizations must satisfy the following criteria:

1. It must be *correct*, i.e., it must not change the behavior of any program.
2. It must be *effective*, i.e., for programs that contain code instances to which the optimization is applicable, it should in fact optimize those code instances.

5.1. Evaluating correctness

Correctness will be assessed using some subset of the test inputs used for Assignment 1.

5.2. Evaluating effectiveness

The effectiveness of an optimization will be evaluated by looking at instruction execution counts (obtained using `spim -keepstats`) and considering the extent to which the optimization is effective in reducing the execution counts for different kinds of MIPS instructions. See the appendix for more details.

5.3. Point distribution

At this point, you have completed Assignment 1 and have access to all of the test inputs used for that assignment (they were released separately for each milestone of assignment 1, and are also available together under the directory **/home/cs553/fall23/TestCases/all-codegen/** on lectura). It is expected that your compiler will generate correct code for all of these programs for all combinations of `-Olocal` and `-Oglobal`. Generating correct code for these programs will not, in itself, earn you points for this assignment; however you will lose points if your compiler fails any of the correctness tests.

Assuming that your compiler passes the correctness requirement, I will use a selection of test inputs, as discussed in Section 5.2 above, to evaluate each optimization specified above. Points will be awarded based on whether these optimizations are being carried out as expected.

Finally, your evaluation of your optimizations, as given in your EVALUATION.txt file, will count for 10% of the grade for this assignment.

5.4. Partial Credit

You may be eligible for partial credit if your program works for some inputs but not others: see the **Grading Procedure** described in the course syllabus for details of how to go about this. Note that *you* have to initiate the partial credit process by submitting the necessary information (described in the syllabus).

6. Submitting your work

Submit your work in GradeScope in the submission area created for this assignment. You should submit the following files:

- All files needed to build an executable of your compiler from scratch;
- a Makefile that provides (at least) the following targets:

  make clean:
  Deletes any object files (*.o) as well as the file 'compile'

  make compile:
  Compiles all the files from scratch and creates an executable file named 'compile'; and
- a file EVALUATION.txt evaluating your optimizations as discussed in Section 4.

7. Gotchas to watch out for

7.1. Copy propagation involving values of different types

Copy propagation can interact with implicit type conversion in unexpected ways. For example, consider the following program:

```
int i0;
char c0;

void main(void) {
    c0 = 255;
    i0 = c0;  /* the RHS value (== 255) is sign-extended to -1 */
    println(i0);
}
```

In this program, the assignment '`i0 = c0`' converts the char value `c0` to an int value `i0`, and in the process performs sign-extension. If we perform copy propagation without paying attention to this, we get:

```
int i0;
char c0;

void main(void) {
    c0 = 255;
    i0 = 255;  /* the RHS value (== 255) is not sign-extended */
```

```
        println(i0);
    }
```

In this program, the fact that the assignment to `i0` needs to sign-extend its operand is lost, resulting in incorrect code.

Recommendation: For this assignment, it's OK to not perform copy propagation when doing so would result in a type conversion problem such as that shown above.

7.2. Liveness analysis and array references

An array reference of the form '`A[i] = ...`' in the source code translates to three-address code of the following form:

```
... compute the address of A[i] into tmp0 ...
deref(tmp0) = ...
```

In such cases, note that even though tmp0 occurs as part of the destination operand of this instruction, its value is nevertheless "used" by the instruction, so tmp0 should be considered live at the point immediately before this instruction.

Appendix A: Evaluating Optimizations

A.1. Obtaining instruction execution counts

Instruction execution counts can be obtained using **spim-stats**, a version of SPIM that provides statistics on different kinds of instructions executed by a program.

The source code for **spim-stats** is available here (as a zipped file) and also on the CS Department server **lectura** in the directory **/home/cs553/fall22/spim-stats**; the executable for this version of SPIM is on **lectura** in the file **/home/cs553/fall22/bin/spim.**

To run SPIM on a file **foo.s** containing MIPS assembly code, use: **spim -file foo.s**

To get execution count statistics, use: **spim -keepstats -file foo.s**

For example: The following two files are the source and MIPS assembly code for a program to compute and print out the value factorial(7): fact.c; fact.s

When we run it on SPIM using the command "`spim -keepstats -file fact.s`" the generated output is

```
5040
Stats -- #instructions : 307
         #reads : 84  #writes 69  #branches 31  #other 123
```

The first line is the output from the program; the last two lines are statistics about the executed instructions.

A.2. Evaluating the impact of optimizations

This section describes how the optimizations implemented in your compilers will be evaluated in the GradeScope grading script, so that you can more easily understand the feedback you get from the script.

Note: A script is just a piece of not-too-bright software, and it is possible that it may erroneously consider the optimizations you have implemented to be "not good enough." If you disagree with the feedback you receive from the grading script (and there is no penalty if you do), please let me know so that I can take a look and decide whether the optimization is effective.

I will evaluate the effectiveness of optimizations using the two measures described below, which I will apply to a set of programs constructed specifically to check how they are affected by the particular optimizations this assignment focuses on. These measures are:

1. **Baseline instruction execution counts.** The evaluation criterion is that the baseline instruction count, i.e., the number of instructions executed when no optimizations are performed, is not too high, i.e., the code generator generates "reasonable" code. For example, it would reject a sloppy code generator that generates a ton of unnecessary code that is then "optimized" away, thereby artificially inflating the impact of the optimizer.

   I will set the acceptable threshold for the baseline instruction execution counts at 120% of what we would get using the code generation scheme discussed in class.

2. **Instruction execution counts with optimizations enabled.** The evaluation criterion is that the reduction in instruction execution counts should be high enough to indicate that the optimizations (and any analyses necessary to support them) have been properly implemented.

   I will set the acceptable thresholds for this as follows:

   - **`-Olocal`** : the reduction in the number of memory reads executed ('**`#reads`**' in **`spim -keepstats`**) compared to the baseline should be at least 80% of what we would get using the copy propagation algorithm discussed in class.
   - **`-Oglobal`** : the reduction in the total number of instructions executed ('**`#instructions`**' in **`spim -keepstats`**) compared to the baseline should be at least 80% of what we would get using the dead code elimination algorithm discussed in class.

Example 1: Copy Propagation

Consider the following program:

```
void main(void) {
  int x, y, z, w, i;
  x = 0;
  for (i = 1; i < 1000; i = i+1) {
        y = 10;
        z = y;
        w = z;
        x = w;
        y = 20;
        z = y;
        x = z;
    }
    println(z);
}
```

Baseline execution count: Using the code generation approach discussed in class, the three-address code generated for this function should be something like this:

```
enter main
tmp$0 := 0
x := tmp$0
tmp$1 := 1
i := tmp$1

label Lbl0   /* top of for loop */
tmp$2 := 1000
if i < tmp$2 goto Lbl1
goto Lbl2
label Lbl1
tmp$5 := 1
y := tmp$5
z := y
w := z
x := w
tmp$6 := 20
y := tmp$6
z := y
x := z
tmp$4 := 1
tmp$3 := i + tmp$4
i := tmp$3
goto Lbl0   /* bottom of for loop */

label Lbl2
param z
call println, 1
leave main
ret
leave main
ret
```

If we count the total number of MIPS instructions generated from this, we get something like 33 MIPS instructions from the top of the for loop to the bottom of the for loop, which works out to a total of 33,000 instructions executed over 1000 iterations. There are also some instructions at entry to and exit from the function and in setting up the call to println(), adding up to a total of roughly 33,020 instructions.

As mentioned above, the threshold for a "reasonable" baseline execution count is 120% of this value. So for this program I expect the code generated by your compiler to execute no more than 1.2 x 33020 = 39,624 instructions.

Impact of local optimization: The body of the for loop consists of a single basic block containing 5 copy operations, i.e., statements where one variable is being assigned (copied) to another. Using the code generation approach discussed in class, the effect of copy propagation should be something like the following (obviously, the stack offsets may be different):

optimization	three-address code	MIPS asm code

none	`z = y` (original)	`lw  $t1, -8($fp)`
		`sw  $t1, -12($fp)`

`-Olocal`	`z = 10` (due to copy propagation)	`li  $t1, 10`
		`sw  $t1, -12($fp)`

So the impact of copy propagation should be:

- a memory read (`lw`) is replaced by a constant operation (`li`); and
- the number of memory writes (`sw`) and the total number of instructions are not affected.

There are five copy instructions in the body of the loop, and copy propagation should remove one memory read for each such copy instruction on each iteration. Since the for loop iterates 1000 times, I would expect that basic-block-level copy propagation should therefore reduce the number of load instructions (what **`spim -keepstats`** reports as "#reads") by at least 1000 x 5 = 5000. (The introduction of `li` instructions in the optimized code will actually lead to an increase in the number of operations reported as '#other' by **`spim -keepstats`**. For grading purposes, this is OK.)

Since **`-Olocal`** also enables peephole optimizations, the actual reduction in the number of read operations may be higher than 5000. Therefore the grading script will be looking for a reduction in the number of *memory reads* of *at least* 5000; the number of memory writes, branches, and "other" instructions will not be checked.

Example 2: Dead code elimination

Consider the following program:

```
void main(void) {
  int x, y;
  for (x = 0; x < 1000; x = x+1) {
    y = 1;
    y = y + 2;
    if (x >= 0) {
        y = y + 3;
        y = y + 4;
    }
    else {
        y = y + 5;
        y = y + 6;
    }
    y = y + 7;
  }
}
```

For this program, all of the assignments to the variable y should be identified as dead by liveness analysis and removed by global dead code elimination (enabled when **`-Oglobal`** is specified). Each such assignment in the body of the loop, e.g., 'y = y + 1', works out to at least six MIPS instructions:

1. Compute y+1 into a temporary variable, say tmp$1:
   - load the variable y into a register;
   - load the integer constant 1 into a register;
   - perform addition;
   - store the result back into the stack slot for tmp$1.
2. Copy the value of tmp$1 into y:
   - load tmp$1 into a register
   - store the register into the stack slot for y.

Each iteration of the loop executes five such assignments, so dead code elimination should get rid of 5 x 6 = 30 instructions for each loop iteration. Since the loop executes 1000 times, the total reduction in the number of instructions executed as a result of **`-Oglobal`** should be at least 30 x 1000 = 30,000.