Unit Project

CS 426 — Compiler Construction
Fall Semester 2019

Due: 5:00pm, Friday, 13 December 2019

1 Goals

This assignment deals with implementing part of a native code generator (i.e., a back end). Our goals in this project are for you to gain some understanding of writing a native code generator in a commercial-quality compiler for a modern architecture, and also for you to learn about dealing with machine-specific details within a compiler. To make this practical, the project is (a) limited to one pass, Register Allocation; (b) focuses on the well-known x86 architecture, which is well-supported in LLVM; (c) requires only a straightforward implementation of the classical Chaitin-Briggs register allocation algorithm, which we study in class; and (d) is to be done in groups of two. You are not required to implement key optimizations like live range splitting, copy coalescing and rematerialization for this project.

You will evaluate your algorithm's performance on either the EWS Linux machines or on a personal x86-based machine, using 64-bit mode in either case. You only need to handle the general purpose registers, and so you will only test and evaluate your algorithm with programs that do not use floating point types. You will use both C and COOL programs for your tests, but focus on C programs for the performance tests. You must turn in a final report describing the algorithm, the organization of your code, the status, and your results. The report format is described in detail in Section 7, below. You must also turn in your code.

LLVM uses a target-independent code generation system, where target-independent algorithms for instruction selection, register allocation, instruction scheduling, and others are parameterized by a detailed target description. This means that the algorithms have to deal with the union of all possible complexities of all possible target architectures. For this project, you can ignore these issues. You *should* use the Target register description (class `TargetRegisterInfo`) but you should limit your algorithm to the features that are required for the X86-64 general purpose registers. You can ignore the floating point register stack, MMX/SSE/AVX registers, and any non-X86 register features.

2 Working in Teams

Although you will be working in teams of two, both members of a team will get the same grade, *with no exceptions*. You are responsible for deciding how to divide up the work between the two team members, but try to divide it as evenly as possible. Your report should describe clearly what code was written by each person. Regardless, you should *both* familiarize yourself with the algorithmic concepts and the infrastructure enough so that you could do the entire project on your own. Also, have each team member understand and check the other person's code. I strongly recommend trying an approach called "pair programming" (look it up on *Wikipedia*) for the more important parts of such a project, e.g., for developing the overall structure, designing major interfaces, choosing the data structures, etc.

3 Preparation

Here is a recommended order for tackling the reading you need for this project. *Learn to read only the relevant parts of such documents, and to do so as you need them.* In any project where you must work within the context of a large system, there will be too much total documentation to absorb it all before you start. Those who optimize their initial reading in this way will make progress most efficiently. *Begin this reading immediately.*

- Read the lecture notes on Register Allocation. Come by and see me if you have any questions.
- Read the *Intel 64 / IA-32 Software Developer's Manual, Volume 1*, initially focusing on Chapters 1, 3, 4.1–4.3, and 5.1. You can refer to other chapters as needed. This manual is now available on the class *Resources* page, along with Volume 2, which includes the *Instruction Set Reference*. You should not need to read the latter much: use it mainly as a reference to understand the details of specific instructions, e.g., during debugging. In particular, most of the details of machine instructions needed for register allocation, such as lists of operands that are definitions and uses, are accessible via generic interfaces and data structures: it would not be possible to write machine-independent allocation algorithms otherwise.
- Read and understand the following items, in order to understand the LLVM code generation system:
  - The *LLVM Target-Independent Code Generator* document, the Sections called *Introduction, The TargetRegisterInfo class, Machine code description classes, Live Intervals* and *Register Allocation*.
  - The document *Writing an LLVM Pass*, Section called *Registering dynamically loaded passes*, and in particular, the information on using `RegisterRegAlloc`. This conveniently shows you how to create a new register allocation pass and register it so it can be invoked using the standard command line option: `llc -regalloc=name`.
  - The X86 register description file, `llvm/lib/Target/X86/X86RegisterInfo.td`. Use the document, *TableGen Fundamentals*, *as a reference* to understand this file. Again, using this as a reference means that you look at it mainly to answer specific questions you may have when reading the X86 register description file.

4 Setting Up Your Software Environment

For this project, unlike MPs 2-4, you will need to work within the LLVM source tree. Follow the *LLVM Getting Started Guide* to download the tarball containing the source files for LLVM 9.0.0, and configure and compile the code for debugging. Add the `LLVM_OBJ_DIR/bin` directory to your path instead of (or ahead of) the ones in the cs426 class directory on EWS (where `LLVM_OBJ_DIR` is the directory in which you build LLVM).

Now, create an empty register allocation pass, by copying and gutting the file `llvm/lib/CodeGen/RegAllocBasic.cpp`. Use the `RegisterRegAlloc` mechanisms above for registering your new register allocation pass. Place the new `*.cpp` file(s) in the same directory, and add them to the file `CMakeLists.txt` so that they automatically get included in the `libLLVMCodeGen.a` library, which is linked in to `llc` and `lli`. Recompile and run `llc -help` to make sure that your new regalloc option is included in the list of register allocation choices.

The following command line will both print out the various passes being executed by the back end (-debug-pass=Executions), and print out the internal code at several points during the process (-print-machineinstrs):

```
llc -debug-pass=Executions -print-machineinstrs somefile.bc -o somefile.s
```

The `-debug-pass=Executions` option is very informative: look through and try to understand (in simple terms) the various steps taken by the LLVM code generator. The `-print-machineinstrs` is extremely verbose, so use it sparingly. Note that you do not need the `-march` option of `llc` to invoke the X86 back end code generator when you are on an x86-based system: the target architecture defaults to be the same as the host system on which the compiler is run.

Make yourself familiar with the debugging output printed out by the LLVM backend: it will be essential to you. Use very small functions with simple operations to understand this code, because this will produce a *lot* of output for each function. Later, you can write small C test cases similar to the smaller ones in `llvm/projects/test-suite/SingleSource/UnitTests`, but focus on simple instructions you expect to encounter first.

5 Tasks and Timeline

Here is a suggested order in which to approach the problem, along with rough time estimates for the major tasks. Note that you have roughly 3.5 weeks for the project, but overlapping with other assignments.

1. (**Roughly 1 day**) Set up your software environment, as described above.
2. (**Roughly 1 week**) Begin by writing a trivial pass that spills every LLVM virtual register at every instruction. This pass should allocate a stack slot for each LLVM register, store the register to the stack slot after every definition, and load it before every use. You will have to check if an operand is a register (because there are non-register operands!) and is not a physical register (because some special cases like formal parameters and return values may already be assigned to physical registers). **Test your code on several simple examples.**
3. (**Roughly 1-1.5 weeks**) Now, extend your algorithm to reduce spilling. You can use the existing `LiveIntervalAnalysis` pass in `lib/CodeGen/LiveIntervalAnalysis.{h,cpp}` as is: you don't need to implement this analysis yourself. This pass will compute live intervals for individual LLVM registers and also for physical registers. You can use the results of this pass to compute interferences between the live intervals and representing them in an interference graph. Finally, you can assign physical registers to virtual registers using the Chaitin-Briggs deferred spilling algorithm. Use the `VirtRegMap` class to record register assignment and spill decisions, and the `Spiller` class to insert load/store instructions for spills.[1] **Test your code again, on several simple examples.**
4. (**Roughly 2-3 days.**) Now, **test your code thoroughly on larger programs**, including (at least) five medium-to-large C programs from the `test-suite` project, which I will specify later.
5. (**Roughly 1-2 days**) Evaluate the performance of the two versions of your algorithm (from the previous three steps) as described in Section 4, and compare them to four existing LLVM allocators –

[1] The *LLVM Target-Independent Code Generator* document says to see the file `lib/CodeGen/RegAllocLinearScan.cpp` for an example of how to use the spiller, but that file no longer exists. Instead, see `lib/CodeGen/RegAllocGreedy.cpp` or `lib/CodeGen/RegAllocPBQP.cpp`.

`basic, fast, greedy` and `pbqp`. For this evaluation, maintain a count in your code of the number of spill instructions inserted (counting loads and stores separately) for each of your allocators. Use the class `LLVMStatistic` for the counter; you can then use -stats to report this number.

6. (**Roughly 1 day**) Write your report as described in Section 8. Submit your work as described in Section 9.

The time estimates above are highly approximate, and I have calculated them knowing that you have other responsibilities. Use these estimates as a rough guide to track your progress.

6 Experiments

Evaluate the performance of the two versions of your algorithm – let's call them `SpillAll` and `Full` – against each other, and against the four LLVM allocators mentioned above: `basic`, `fast`, `greedy` and `pbqp`. For each program, report the following two numbers *in two separate tables*: (i) the total running time of the program for each of the six allocators; (ii) the total number of spills, as reported by -stats for each of the six allocators. In other words, each table should have one row for each program, one column for the size of the program, and one column for each of the six allocators. To measure the running times for each benchmark program you compile and each allocation algorithm, execute the generated code five times and report the average of the runs (check manually for outliers: if there are noticeable outliers, you may need to re-run the experiment).

The first version of your algorithm and the `Basic` and `Fast` LLVM allocators at least should produce significant numbers of spills. Use at least 10 C programs to carry out these comparisons, including at least 5 medium-to-large programs, as noted earlier. You can use COOL programs to test your code, but use C programs for the performance tests. A number of C programs are available from the LLVM test suite:

`http://llvm.org/docs/TestSuiteMakefileGuide.html`.

Ignore the LNT framework mentioned there: it is too heavyweight for your needs. Note that I do *not* expect you to beat the best LLVM allocators.

Detailed Analysis of Two Programs: Finally, and most importantly, choose two programs for which you see the biggest difference in performance across your allocators. For these programs, analyze why the difference in performance occurs. To do this, *you will need to read assembly code to understand what spill decisions the allocator is making and why they lead to different numbers of spills.* Reading and understanding the performance of assembly code is a very important part of machine code generation for any architecture. This step accounts for a substantial part of the grade for this project, so try to be thorough in your analysis and detailed in your report.

7 Report

Your report should be about (and no more than) 5 pages long, 11-on-14 pt. (i.e., 11pt. font with 14pt. line spacing) with 1 in. margins, formatted as a PDF file. It should include:

1. A brief description (in one page or less) of the Chaitin-Briggs graph coloring register allocation algorithm *in your own words.* Use one or more diagrams to make it easier to understand.

2. The heuristic you use for choosing what to spill, along with a brief rationale for it, again *in your own words.*

3. A brief description of the status of your code: what works, what doesn't, and especially any key features that you would have liked to implement but could not (if any).

4. A clear summary of the division of work, listing what major classes or functions were written by each member of the team: *this should describe what actually happened, not the initial plan*!

5. An experimental section, including:

   - The two performance tables described above, along with a concrete discussion of the important conclusions to be drawn from them. *Don't just repeat the numbers in the tables.*
   - The analysis of the performance differences of the two selected programs. Be systematic and clear in explaining the differences you observed. Include code segments in an Appendix, if necessary; there is no page limit for this Appendix (but don't go overboard!).

6. References. Never write a technical document without them!

8 Grading

You will be graded on the performance of the allocators you write, on your code, on your performance experiments, and on the contents of the report. Specifically the grade for this project will comprise:

- 30% for the the status of your work, in particular, the extent to which both versions of your allocator have been completed and tested.
- 10% for the quality of your code implementing the allocators. Factors I will look at are (i) appropriateness of data structures and algorithms; (ii) modularity; (iii) reuse; (iv) documentation; and (v) your unit tests.
- 30% for the results of the performance experiments, including the range of benchmarks tested, the analysis of performance differences between your allocators for two programs, and the relative performance of your best allocator compared with the best LLVM allocator for each program. I do *not* expect you to beat the best LLVM allocators; this is just to see how close you can come.
- 20% for the presentation in the report, including the description of the algorithm and the presentation of the experimental results.
- 10% for the effort you put into the project, as shown by the heuristics, experiments and report. This should be a free 10% for you!

9 What and how to hand in

Hand in your files as a compressed tar file holding a directory named `netid1-netid2`, containing:

- *`netid1-netid2`*`-report.pdf`: **Please follow this naming convention.**
- All the new source files you have created. All these files should have been in `llvm/lib/CodeGen`. No existing source files or Makefiles (except any CMakeLists.txt files) should be modified.
- The unit tests you wrote to test your algorithms. Provide a Makefile, along with instructions to compile and run the tests. If any tests were only used for certain algorithms, make that clear. Also, list any tests that are not expected to pass (this is quite common).