L25: Modern Compiler Design Exercises

David Chisnall

Deadlines: October 29^{th}, November 5^{th}, November 19^{th}

These simple exercises account for 20% of the course marks. They are intended to provide practice with the techniques covered in the course and are marked on a simple pass/fail basis. The deadlines for each of these are 1pm on the date noted along with each exercise. The 12-1pm slot on most Wednesdays during full term will be a lab class when you can receive help working on the exercises and have them assessed. The deadlines for completing each of the exercises are 12 noon on the date given alongside each exercise. They can be ticekd in the immediately following lab class *or at any lab class earlier in the term*.

The remaining 80% of the course marks are awarded for the miniproject. You must also submit an approved miniproject proposal to the Graduate Education Office by noon on Friday **November** 14^{th} and must submit the write-up by **January** 13^{th}. Further details will be given in lectures.

1 Exercise goals

These exercises have three purposes:

- To ensure that you are comfortable modifying a large existing compiler codebase.
- To check that you have understood the material covered in the lectures.
- To ensure that you understand the evaluation requirements of a systems-research paper, in preparation for the miniproject.

2 Setting up

The ACS lab machines have a debug build of LLVM available and two scripts that will set up the environment for you. These are both located in `/auto/groups/acs-software/L25`. The `setup.sh` script is intended to be sourced into your current shell and will set up paths for you. The `get-examples.sh` script will download and build the examples for you.

Make sure that you run the `get-examples.sh` script before proceeding! After running this script, you should have a `Debug` and a `Release` subdirectory in each

of example programs' directories. The first contains a debug build and should be used whenever you are trying to find errors in your code, as it is built with a large number of assertions in LLVM and with all debugging symbols. The `Release` directory contains an optimised build, which is expected to run approximately an order of magnitude faster. *Make sure that you run any benchmarks that you use for evaluating your changes with the release build!* You can regenerate either after modifying the program by simply typing `make` in the relevant directory.

If you are using the ACS lab machines then you can ignore the build instructions in each `README.md` file. The `get-examples.sh` script will have created build directories for you and you can just type `make` in the `Release` or `Debug` directory to rebuild after making changes. If you wish to do the coursework on your own machine, then you will need to make sure that you have a C++11 compiler and standard library, LLVM 3.5, and CMake 2.8.8 installed before fetching any of the examples.

The example projects are all local git clones. Checkpoint your work with `git commit -a` periodically so that you can later undo any mistakes you might make. If any errors are found in the example code, they will be fixed in the central repository and you can pull in a new version with `git pull --rebase`.

3 Exercise 1: Writing a simple pass (5%)

The SimplePass example shows a trivial LLVM pass that just dumps `alloca` instructions. Read the `README.md` file accompanying this example and make sure that you can build the pass and that it runs when you instruct clang to use it when compiling a C or C++ source file. If you are using the ACS lab machines then you do not need to follow the build instructions, only the usage instructions. You will already have debug and release builds in the `Debug` and `Release` subdirectories of the pass directory and can just rebuild by typing `make`.

Note that the release and debug builds of this pass can both be used with either a release or debug build of clang. It is generally easier to find bugs if you can see debugging symbols in the whole program, so I strongly recommend that you use the debug build of clang when testing. This can be found in `/auto/groups/acs-software/L25/llvm/bin`. The release build of clang is in `/auto/groups/acs-software/L25/llvm-release/bin` and should already be in your path (so can be invoked as simply `clang`) if you have sourced the `setup.sh` script.

A common heuristic for code compiled from C/C++ is that it contains one branch every 7 instructions on average. Modify the SimplePass example to investigate each basic block and count the instructions. Exclude `inttoptr`, `ptrtoint` and `bitcast` instructions, which will not expand to any instructions in a target.

Compile a program or library that you use often with this pass and plot a graph showing the frequency distribution of the block sizes. How far off is the assumption that there's one branch every 7 IR instructions? Does this

change if you discount `GetElementPtr` instructions? What about if you count instructions in basic blocks that end with unconditional branches as if they were part of the next basic block? What happens if you treat `call` instructions as ending a contiguous range?

Deadline: October 29^{th}

3.1 Evaluation criteria

- The SimplePass example must be modified to count instructions per basic block.
- You must present a justification of your choice of software to test.
- You must present a graph of branch frequencies and draw some conclusions about what this means for compilers

4 Exercise 2: MysoreScript (7.5%)

MysoreScript is a very simple language that provides a JavaScript-like model. The implementation is limited in a number of ways, including:

- It lacks any type feedback mechanism
- Method lookups are $O(n)$ in terms of the number of methods in a class
- There is no caching or speculative inlining of methods.

You should improve the system by adding one of the following:

Inline caching, so that the JIT-compiled code has a hard-coded address for a direct jump if the class of an object at a call site is the expected one. This will require modifying the interpreter to record possible classes. Make sure that you only insert inline caches when there's a good chance that they'll be hit!

Type specialisation for arithmetic, so that the compiled code will jump back to the interpreter if the argument values are not integers, but will then proceed without additional run-time checks if they are.

Improved dispatch tables, replacing the linked list. Try adding either a sparse tree or inverted dispatch tables (where each selector has a class-to-method mapping, rather than each class having a selector-to-method mapping) and modify the compiler to do lookups inline, rather than calling out to C code.

Whichever option you pick, show some example code where it gives a performance increase and be prepared to justify whether this is representative.

Deadline: November 5^{th}

4.1 Evaluation criteria

- You must make one of the proposed changes to the MysoreScript example.
- You must present a justification of your choice of benchmark programs and whether they are representative.
- You must either show that your modification has given a statistically significant speedup or explain why it does not.

5 Exercise 3: CellularAutomata (7.5%)

This is a simple compiler for a domain-specific language for generating cellular automata. The language itself is intrinsically parallel—you define a rule for updating each cell based on its existing value and neighbours—but the compiler executes each iteration entirely sequentially, one cell at a time.

There are lots of opportunities for introducing parallelism into this system. Pick one of the following:

Vectorised implementation. The current version is not amenable to automatic vectorisation because the edge and corner implementations are not the same as the values in the middle. Modify the compiler to generate three versions of the program: one for edges, one for corners, and one for the middle. Make the edge and middle implementations simultaneously operate on 4 (or more) cells by using vector types in the IR. Be careful with the global registers!

Parallel implementation. Divide the execution between two or more threads, extending the operations on globals so that they provide a guaranteed ordering. Each operation that modifies a global register should become a barrier, ensuring that the current iteration does not proceed until all previous iterations (in grid order) have reached that point. Note that an efficient implementation of this will require modifying the compiled program to automatically jump to the next element in the queue when this happens.

Deadline: November 19^{th}

5.1 Evaluation criteria

- You must make one of the proposed changes to the CellularAutomata example.
- You must present a justification of your choice of benchmark programs and whether they are representative.
- You must either show that your modification has given a statistically significant speedup or explain why it does not.