L25: Modern Compiler Design Exercises

David Chisnall

Deadlines: November 1st, November 15th, November 29th

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. They can be ticked in the lab class immediately before the deadline *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 Wednesday **November** 2nd and must submit the write-up by **January** 17th. Further details will be given in lectures.

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.

Setting up

You may either use the ACS lab machines or your own computer for the coursework assignments. If you are using your own machine, then please use the scripts in `https://github.com/compilerteaching/Scripts` to set up LLVM and the examples. You will need to modify the line that sets `INSTALL_PREFIX` in the `setup_llvm.sh` script. Note that using your own machine will require around 30GB of free disk space for debug and release builds of LLVM (the lab machines have these in a shared NFS space). **If you wish to use your own machine then please ensure that you have set up everything before the first**

lab class. You will need to have `libedit` and the Boehm Garbage Collector (including associated headers), and a working C++14 toolchain, installed before running the scripts.

If you are building on macOS, then you may need to edit the `setup_llvm.sh` script to pass a `-DCMAKE_OSX_SYSROOT=` flag to `cmake`. The exact value of this flag depends on your XCode installation.

The ACS lab machines are set up with debug and release builds of LLVM, located in the `/auto/groups/acs-software/L25/llvm` and `/auto/groups/acs-software/L25/llvm` directories, respectively.

To set up the examples run:

```
$ /auto/groups/acs-software/L25/get-examples.sh
```

You should now have three directories (`SimplePass`, `CellularAutomata`, `MysoreScript`), one for each example.

In each of these, you will find two build directories:

Debug contains a build with all debugging symbols and with assertions. Use this when developing the code for your assignments. The assertions will help catch bugs in our code (usually close to where they occur).

Release contains an optimised build with assertions disabled. *Use this when benchmarking!*

You can regenerate either build after modifying the program by simply typing `ninja` in the `Debug` or `Release` directory.

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`. If you have any uncommitted local changes, then you should run the following commands:

```
$ git stash
$ git pull --rebase
$ git stash pop
```

This will put your changes to one side, apply the remote changes, and then reapply your changes on top. Note that you will need to merge any conflicts with the remote bug fixes.

Tests

The MysoreScript and CellularAutomata compilers (for exercises 2 and 3) have test suites accompanying them. You can run the test suite by running `ctest` in one of your build directories. For example:

```
MysoreScript/Debug$ ctest --output-on-failure
Test project /home/dc552/L25/MysoreScript/Debug
    Start 1: binding
1/8 Test #1: binding .........................   Passed    0.31 sec
    Start 2: binding_jit
2/8 Test #2: binding_jit .....................   Passed    0.43 sec
    Start 3: inheritance
3/8 Test #3: inheritance .....................   Passed    0.32 sec
    Start 4: inheritance_jit
4/8 Test #4: inheritance_jit .................   Passed    0.42 sec
    Start 5: method_inheritance
5/8 Test #5: method_inheritance ..............   Passed    0.32 sec
    Start 6: method_inheritance_jit
6/8 Test #6: method_inheritance_jit ..........   Passed    0.45 sec
    Start 7: operator
7/8 Test #7: operator ........................   Passed    0.33 sec
    Start 8: operator_jit
8/8 Test #8: operator_jit ....................   Passed    0.46 sec

100% tests passed, 0 tests failed out of 8

Total Test time (real) =   3.06 sec
```

Note the `--output-on-failure` flag, which instructs `ctest` to tell you *why* a test failed, if it did. This will help when debugging. Running the tests on the debug build will ensure that you trigger any assertions that may help you find the reason for your bugs early.

If you are finding that the tests take too long to run, you can try using the `-j <jobs>` option, which instructs `ctest` to run multiple tests in parallel. For example `ctest -j4` will run four tests at a time.

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 `get-examples.sh` script 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 `ninja`.

Remember to use the `clang` binary that matches the build type (i.e. `/auto/groups/acs-software/L25/llvm/bin/clang` for the debug build, `/auto/groups/acs-software/L25/llvm-release/bin/clang` with the release build) or you may see some odd behaviour, including crashes) from ABI mismatches.

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: November 1st

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

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.
- Compilation (and optimisation) happens at a method granularity.

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! If you're feeling particularly adventurous, you can use the LLVM patchpoint intrinsic for true inline caching, though this is not required for the marks.

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. In particular, for sequences of integer arithmetic, you should be able to evaluate the entire sequence without branches and then branch to the handler at the end.

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 15th

Evaluation criteria

- You must make one of the proposed changes to the MysoreScript example.
- Your modified code must not cause regressions in the test suite.
- 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.

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 29th

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.