L25: Modern Compiler Design Exercises

David Chisnall

Due: December 6

These simple exercises account for 20% of the course marks. They are intended to provide practice with the techniques covered in the course. Each exercise involves modifying one of the two sample systems.

1 ToyRuntime

This system is a (very!) simplified version of a dynamic language runtime with a class-based model, with no support for inheritance. It is intended to be used from C and comes with a simple set of macros to allow this use.

The runtime is intentionally highly suboptimal and not thread-safe, allowing a lot of potential for improvement. The test program shows two trivial implementations of Fibonacci sequence generators, which use dynamic dispatch for recursive calls. These provide a very simple benchmark.

You can compile the test program with `clang -emit-llvm` to generate LLVM IR (add `-S` to get the human-readable form) and then transform it in a variety of ways.

2 CellularAutomata

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 prallelism into this system.

3 Exercises

You must complete any three of the following tasks:

3.1 Parallel Automata

Extend the CellularAutomata language to execute in parallel, in different threads. This may make use of an existing library such as libdispatch or your own man-

ual pthread creation. Think about how you will partition the execution and distribute the data.

3.2 Vector Automata

Extend the CellularAutomata language to make use of vectors

3.3 GPU Automata

Extend CellularAutomata language to run on the GPU using the PTX back end to LLVM (or SPIR, if you can find some drivers that support it). Note that the GPUs of the machines in the project lab do not support PTX.

3.4 Inverted Dispatch Tables

Modify the example dynamic language runtime to implement inverted dispatch tables and write an LLVM pass that creates a per-selector function that walks the inverted dispatch table for each used selector and modifies calls to it to use it.

The `dlsym()` function will be useful for setting up the inverted dtables from the runtime. This is a standard library function that allows running code to dynamically look up symbols. Note that you may need to compile with `-Wl,--export-dynamic` for this to work, depending on the platform.

3.5 Inline Caching

Write an LLVM pass that adds inline caching to method lookup calls for the dynamic language runtime.

4 Submission

Each exercise should be submitted as source code and a single-page PDF file outlining the changes that you made, your rationale, and how you evaluated performance differences.