Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2016 | Optimizer Project Assignment | Thursday, Nov 10

Important Dates:

Checkpoint: Wednesday, Dec 7, 11:59 pm
Final Compiler and Documentation: Tuesday, Dec 13 @ 11am
Derby: Wednesday, Dec 14

In this final phase of the project your goal is to reduce the execution time of your generated code while preserving the semantics of the given application. This phase is completely opened-ended; you are not required to implement any specific transformations. Your grade will be based on your overall design (with heavy emphasis on the writeup) and how well your compiler performs as compared to the submissions of the other groups. During the final class, we will hold the Compiler Derby where your compiler implementation will compete against the implementations of your fellow classmates on a hidden program (the derby program).

In the final month of class, we will cover various code improving optimizations. The x86-64 architecture is very complex and aggressive. A substantial portion of this project phase is to determine which optimizations will provide a benefit given the programs of the provided benchmark suite and the target architecture. Some optimizations that we will cover in class include:

- **Register Allocation** – Your compiler can implement a graph coloring based register allocator. See Chapter 16 of the Whale book and §9.7 of the Dragon book. Your register allocator should take advantage of the full set of general-purpose registers provided by the x86-64 ISA. It should also respect the Linux calling convention including caller-save and callee-save properties of each register.
- **List Scheduling** – Instruction scheduling minimizes pipeline stalls for long latency operations such as loads, multiplies, and divides. See Chapter 17 of Whale book.
- **Instruction Selection** – So far, we have been using a restricted subset of the x86-64 ISA. As a peephole optimization, you might replace a sequence of simple instructions with a single, complex instruction that performs the same operation in fewer cycles.
- **Data Parallelization** – Computation executed in different iteration of a loop may be independent of each other. We are providing you with a library to help you find independent computations, and perform automatic loop parallelization.

In order to identify and prioritize optimizations, we have provided you with a benchmark suite of image-processing programs. These programs are more complex than the code that has been provided during the previous phases, so your first priority is to produce correct unoptimized assembly code for each benchmark.

After your compiler produces correct unoptimized code, you should begin to study the applications of the benchmark suite. You will use these programs (and any other programs provided during previous phases) to determine which optimizations you will implement and in what order they will be applied. You are expected to analyze the assembly code produced by your compiler to classify the effectiveness of each proposed optimization, perhaps first applying the optimization manually and empirically measuring the benefit. Your writeup should include evidence for the effectiveness of each optimization you considered.

You are not limited to the optimizations covered in class. You are free to implement any optimization that you come across in your reading or any optimization that you devise. However, your writeup must demonstrate that each optimization is effective and semantics preserving. Furthermore, you must argue that each optimization is general, meaning it is not a special-case optimization that works only for the derby program or a specific application in the benchmark suite.

You should consult Intel's documentation for details regarding our target architecture and the ISA. The documentation is linked from the class website. To benchmark your generated assembly code, provide the `-pg` option to `gcc` or `cc` while assembling in order to generate profiling information. After the code is executed, you can use `gprof` to examine the generated profile statistics. Also, you can use the Unix command `time` focusing on the "user" time. A more accurate timing mechanism is included in the provided thread library. It will be discussed during the project recitation.

Writeup

The writeup for this project is extremely important. Although it explicitly accounts for only 20% of the grade, it will also be used to determine your score for the Implementation aspect of the project (40%).

Your written documentation must discuss each optimization that you considered. The thoroughness of your exploration of the optimization space will be an important aspect of your grade. For each optimization that you implement, your writeup must convince the reader that the optimization was beneficial, general, and correct. For each optimization that you decide not to implement, you must convince the reader that the optimization would not be beneficial given your generated code for the given benchmarks and our target architecture.

You should include assembly or IR code examples for each optimization. Show how your generated code is transformed by the optimization (might be hand-applied for optimizations you did not implement). Highlight the benefit of the optimization on the assembly or IR code. Discuss exactly how the benefit was achieved given the characteristics of our target architecture. Include empirical evidence that proves your conclusion for the optimization.

Your compiler should include a "full optimizations" command-line option (see below). Your written documentation should present a detailed discussion of this option including how you determined the order in which your optimizations are performed and how many times you apply the optimizations. Finally, describe any hacks or solutions to tricky problems that you encountered.

Derby

On the final day of class we will hold the compiler derby. Your compiler will be pitted against the submissions from the other groups in the class. The derby benchmark will be revealed one day

prior to the derby. 40% of the grade will be determined by your ranking in the derby. At this time, we will not reveal the exact formula for determining this portion of the project grade.

Register Allocation / Other Papers

Beyond the normal course materials, we have provided links on the course website to useful and well-known papers on register allocation. Reading these before designing and writing your register allocator may be very useful. We will also post links to papers about other useful optimizations to the course website as we discover them. You can find these links under the "Reference Materials" section.

Parallelization

We are providing you with a simple parallelization analysis library to help you find parallelism in programs being compiled. The library will help you compute the *distance vector*. For more information you are encouraged to read §9.3 in the Whale book. There will also be more information on specific implementation details provided during the project information session. You can find the library in the Java and Scala skeleton repositories. At this time a Haskell library does not yet exist; however, it is not difficult to port the library to Haskell. You may find this library useful, or you may do your own analysis.

We will also provide you with a library that will help you spawn and join threads. Again, you will be welcome to use it, or you can use your own techniques for spawning and joining threads.

Compiler Submission

Your compiler must provide a command-line option for each optimization. Your project writeup should include documentation for each command-line option. For example:

- `--opt=regalloc`: turns on register allocation
- `--opt=instsel`: turns on instructions selection peephole optimizations
- `--opt=listsched`: turns on list scheduling
- `--opt=parallel`: turns on data parallelization of loops

You must provide a `--opt=all` flag to turn on all optimizations and apply them in the order you have determined ("full optimizations"). This option should consider how many times each optimization is applied and the application order of the optimizations.

As before, your generated code must perform the runtime checks listed in the language specification. These may be optimized (or removed in some cases) as long as they report a runtime error whenever the unoptimized program reports an error.

What to Hand In

Follow the directions given in project overview handout when writing up your project. For the written portion, make sure to include a description of every optimization you implement. Show clear *examples* of IR excerpts (and x86-64 excerpts, if relevant) detailing what your optimizations do. You should also include a discussion of how you determined the order in which your optimizations are performed.

Hand in your code using the online handin system on the website.

You should be able to run your compiler from the command line with:

```
./run.sh --target=assembly <filename> -o <outputname>        # no optimizations
./run.sh --target=assembly --opt=all <filename> -o <outputname> # all optimizations
```

Your compiler should then write a x86-64 assembly listing to: `<outputname>`. Note that gcc uses the `.S` extension to determine the filetype, so be sure to use it.

Test Cases

We have provided sample programs that perform image processing and filtering. The programs are located in test repository, along with test images. These programs must be linked against the library provided in lib directory. You should make sure that any valid program provided during previous phases continues to run correctly. We will also test your compiler (including optimizations) on a suite of hidden programs.