15-411 Compiler Design, Lab 5 (Spring 2024)

Jan and co.

Compiler Due: 11:59pm, Wednesday, April 10th, 2024
Report Due: 11:59pm, Tuesday, April 16th, 2024

1 Introduction

In Lab 5, you will be implementing optimizations for the language L4, which remains unchanged from Lab 4. Your goal is to minimize the running time of the executable generated by your compiler on a set of benchmarks.

1.1 Preview of Deliverables

You are expected to submit a working compiler for the L4 language, as with Lab 4. Your compiler will still be tested for correctness against the L1-L4 test suites. However, in addition to testing for correctness, we will be scoring your compiler's output assembly on a suite of benchmark tests created by the course staff.

You are also required to submit a write-up which describes the optimizations you implemented, and evaluates the performance improvement each one provided.

1.2 Unsafe Compilation

Your compiler is now expected to accept an `--unsafe` flag. When compiling in `--unsafe` mode, your compiler can ignore any exceptions that might be raised during the execution of the program, **except** ones due to `assert`. This means you can eliminate most safety checks from the code you generate. For tests which do not encounter runtime errors, your generated assembly should still return the correct value (we will be checking this).

You are not *required* to eliminate all (or, indeed, any) checks, but your compiled code will be significantly slower if you do not take advantage of this flag. Eliminating runtime checks also puts your compiler on a fair playing field when comparing your performance to `gcc` (see "Testing" section).

In addition to the `--unsafe` flag, your compiler must take a new option, `-O`n, where `-O0` means no optimizations, and `-O1` performs the most aggressive optimizations. `-O0` should minimize the compilation time, while `-O1` should prioritize the emitted code's running time. We will pass the `-O1` flag when timing your compiler on the benchmarks. The `-O0` flag is mainly for your own debugging purposes.

2 Optimizations

In the following sections, we provide you with a list of suggested analysis and optimization passes you can add to your compiler. This is a long list and we do NOT expect you to complete all of the optimizations. We suggest that you pick the optimizations that you are most interested in, and do enough optimizations so that your compiler is competitive with `gcc -O1`. To help you decide which passes to implement first, we also list the course staff's impression of how difficult each optimization is, and how useful you can expect it to be (though your mileage may vary).

If you have already implemented any of these optimizations, you can still include them in your written report, but you will need to empirically evaluate their impact. You may also want to revisit your optimizations to improve them. In this case, your report can include a description of the improvements you made (and their empirical impact).

Feel free to add other optimizations and analyses outside of this list as you see fit, although we strongly recommend first completing basic ones before you go for more advanced ones. It is a good idea to consult the course staff first to ensure that your planned optimizations are feasible to complete before the deadline.

There is abundant literature on all the following optimizations, and we have listed some good resources that might be helpful for this lab. We specifically recommend:

- The Dragon Book (Compilers: Principles, Techniques, and Tools, 2nd Edition)
- The Cooper Book (Engineering a Compiler, 2nd Edition)
- The SSA Book (SSA-Based Compiler Design)

all of which have sections on compiler optimizations. Additionally, the recitations, lecture slides, and lecture notes are great resources. We also encourage you to read relevant papers and adapt their algorithm for your compiler, as long as you cite your sources in your written report.

2.1 Analysis Passes

Analysis passes provide the infrastructure upon which you can do optimizations. For example purity/loop/alias analysis computes information that optimization passes can use. The quality of your analysis passes can affect the effectiveness of your optimizations.

1. **Control flow graph (CFG)**
   Difficulty: ★☆☆☆☆ Usefulness: ★★★★★

   Almost all global optimizations (intraprocedural optimizations) will use the CFG and basic blocks. We recommend implementing CFG as a standalone module/class with helper functions such as reverse postorder traversal and splitting critical edges.

2. **Dataflow Framework**
   Difficulty: ★★☆☆☆ Usefulness: ★★★★☆

   A dataflow framework is not only useful for liveness analysis, but also for passes such as partial redundancy elimination (which uses 4 separate Dataflow passes, see section below), among others. You probably want your Dataflow framework to work with general facts (a fact could be a temp/expression/instruction, etc.).

3. **Dominator Tree**
   Difficulty: ★★☆☆☆ Usefulness: ★★★☆☆
   Resources: SSA Recitation Notes
   You can build a Dominator Tree on top of your CFG. The Dominator Tree could be useful for constructing SSA (depending on your implementation algorithm), loop analysis, and many other optimizations.

4. **Single Static Assignment (SSA)**
   Difficulty: ★★★★☆ Usefulness: ★★★★★
   Resources: SSA Recitation Notes
   A program in SSA form has the nice guarantee that each variable/temp is only defined once. This means we no longer need to worry about a temp being redefined, which makes a lot of optimizations straightforward to implement on SSA form, such as SCCP, ADCE, Global Copy Propagation, Safety Check Eliminations, and various Loop Optimizations, among others. In fact, modern compilers such as LLVM uses SSA form for all scalar values and optimizations before register allocation. Your SSA representation will need to track which predecessor block is associated with each phi argument.

   Warning: SSA is immensely helpful, but implementing SSA alone might bloat up your assembly with moves and extra splitted basic blocks, while not giving you much performance benefits. You must implement optimizations on SSA to reap its benefits. Additionally, it will help to compress your CFG and doing coalescing after deconstructing SSA.

5. **Purity Analysis**
   Difficulty: ★☆☆☆☆ Usefulness: ★★★☆☆
   Purity analysis identifies functions that are *pure* (*pure* can mean side-effect free, store-free, etc.), and can enhance the quality of numerous optimization passes. This is one of the simplest interprocedural analysis you can perform, probably using a call graph.

6. **Loop Analysis**
   Difficulty: ★★☆☆☆ Usefulness: ★★★★☆
   Resources: LLVM Loop Terminology
   A Loop Analysis Framework is the foundation of loop optimizations, and is also useful for other heuristics-based optimizations such as inlining and register allocation. Generally, you will do loop analysis based on the CFG, and identify for each loop its header block, exit blocks, subloops, parent loop, nested depth, among other loop features. You might also consider adding preheader blocks during this pass.

7. **Value Range Analysis**
   Difficulty: ★★★☆☆ Usefulness: ★★☆☆☆
   Resources: Compiler Analysis of the Value Ranges for Variables (Harrison 77)
   Value Range Analysis identifies the range of values a temp can take on at each point of your program. We recommend an SSA-based approach. Value Range Analysis can make other optimizations more effective, such as SCCP (eliminating dead branches), Strength Reduction (knowing that a temp is non-negative), and Safety Check Elimination (removing array bounds checks). Identification of the range of loop indices will make this pass more effective.

2.2 Optimization Passes

Below is a list of suggested optimization passes. All these optimizations are doable - they have been successfully performed by students in past iterations of this course.

1. **Cleaning Up Lab3 & Lab4**
   Difficulty: ★★☆☆☆ Usefulness: ★★★★★★

   When implementing Lab3 and Lab4, you likely focused on getting your compiler working rather than outputting perfect assembly. This means that there are likely many things that can be cleaned up. We highly recommend inspecting the x86 assembly from your compiler, and identify any places where you can see obvious improvment. For example, you would want to optimize for calling conventions in lab3 (try not to push/pop every caller/callee register), and you would want to make use of the x86 *disp(base, index, scale)* memory addressing scheme to reduce the number of instructions needed for each memory operation in lab4. Another common mistake is a poor choice of instructions in instruction selection, especially for branch/jump statements (try comparing your assembly to gcc/clang output), or fixing too many registers in codegen and not making full use of your register allocator.

2. **Strength Reductions**
   Difficulty: ★★☆☆☆ Usefulness: ★★★★☆
   Resources: Division by Invariant Integers using Multiplication (Granlund 91)
   Hacker's Delight 2nd Edition Chapter 10
   Strength reductions modifies expressions to equivalent, cheaper ones. This includes unnecessary divisions, modulus, multiplications, other algebraic simplifications, and memory loads. Though simple to implement, this optimization can bring a huge performance improvement (a division/modulus takes dozens of cycles on a modern CPU). Deriving the magic number formulas for division and modulus is tricky, and we recommend you read the above resources, or look at how GCC/LLVM implements strength reductions.

3. **Peephole & Local Optimizations**
   Difficulty: ★★☆☆☆ Usefulness: ★★★★★
   Peephole and local optimizations are performed in a small window of several instructions or within a basic block. Similar to strength reductions, these are easy to implement but can bring a large performance improvement. We recommend comparing your assembly code to gcc/clang assembly code to find various peephole opportunities and efficient x86 instructions.

4. **Improved Register Allocation**
   Difficulty: ★★☆☆☆ – ★★★★☆ Usefulness: ★★★★★★
   Resources: Pre-spilling - Register Allocation via Coloring of Chordal Graphs (Pereira 05)
   Live Range Splitting - Lecture notes on Register Allocation
   SSA-based Register Allocation - SSA Book Chapter 17
   Accessing memory over a memory can often lead to a 4x slow down in execution. So, a good register allocator is essential for optimised code. Since a simple register allocator is likely not sufficient, we provide a list of possible extensions (ordered roughly in increasing difficulty) to your current register allocator[1], which is likely far from perfect. We **highly recommend** at

[1] hopefully you have a register allocator

least implementing coalescing. You may find that pre-spilling is also important for improving your allocator's performance. The rest is up to you:

(a) **Coalescing** We recommend greedy coalescing, which integrates seamlessly into the graph coloring approach taught in lecture. However, there is abundant literature in this area so feel free to explore other coalescing approaches, such as optimistic coalescing.

(b) **Pre-spilling** Pre-spilling identifies maximum cliques in the graph and attempts to pick the best temps to spill before coloring the interference graph. You can also integrate this with your current post-spilling approach.

(c) **Heuristics for MCS and Coalescing** Using some heuristics to break ties in Maximum Cardinality Search and decide the order of coalescing might enhance the quality of your register allocator.

(d) **Live Range Splitting** The naive graph coloring approach assigns each temp to the same register or memory location throughout its whole lifetime, but if the temp has "lifetime holes" between its uses, one can split its live range to reduce register pressure, especially at the beginning and ending of loops. This optimization is more naturally integrated with a linear scan register allocator, but is still possible with a graph coloring allocator. See the lecture notes on register allocation for details.

(e) **Register Allocation on CSSA** Doing register allocation and spilling on SSA might be faster and *might be* more effective. However, it turns out that getting out of SSA is very difficult after doing register allocation on SSA. You should probably spend your time doing some of the other (more interesting and fun) optimizations if you haven't already decided to do this. If you're still interested, the paper *SSA Elimination after Register Allocation* might be useful. **Warning:** this could be challenging to get right, and might not provide significant benefits.

5. **Code Layout**
Difficulty: ★★☆☆☆ Usefulness: ★★★☆☆
Optimizations for code layout include deciding the order of basic blocks in your code, minimizing jump instructions and utilizing fall throughs, and techniques such as loop inversion.

   (a) **Code alignment** Most modern processors can benefit from alignment of loops and functions in the executable, because these instructions are executed multiple times and alignment helps keep instructions within the L1 instruction cache and ITLB. Aligning blocks adds no-ops, which bloats the code size and can impact instruction caching, so you may want to have heuristics for which blocks to align.

   The assembler provides a directive to do this: `.align n` will align the next instruction to n bytes (where n is a power of 2).

6. **Sparse Conditional Constant Propagation (SCCP)**
Difficulty: ★★★☆☆ Usefulness: ★★★☆☆
Resources: Constant Propagation with conditional branches (Wegman and Zadeck)
It is possible to do local constant propagation within basic blocks, but we recommend this SSA-based global constant propagation approach. Additionally, SCCP can also trim dead conditional branches that will never be visited. SCCP might not bring a large improvement

in code performance, but would significantly reduce code size and improve readability. This is one of the first optimizations to consider after you implement SSA.

(a) **Copy Propagation** One related optimization is copy propagation, which is straightforward to implement on SSA, and can also serve to eliminate redundant phi functions. Note that much of copy propagation's functionality is covered by register coalescing, and aggressively doing copy propagation might increase register pressure. However, copy propagation can serve to reduce the number of instructions which speeds up subsequent passes and makes your code easier to debug.

7. **Aggressive Deadcode Elimination (ADCE)**
Difficulty: ★★☆☆☆ Usefulness: ★★★☆☆
Resources: Cooper Book Section 10.2.1
Similar to SCCP, aggressive deadcode elimination is made easier by SSA, and can bring improvement to both code size and performance. ADCE can be made more effective by purity analysis. This is also one of the first optimizations to consider after you implement SSA. It might also help to get rid of the many unnecessary ϕs in your SSA.

(a) **DCE** While ADCE deals with dead blocks and treats conditional jumps as non-critical, you can also implement a much easier yet still quite effective DCE pass that treats conditional jumps as critical instructions.

8. **Partial Redundancy Elimination (PRE)**
Difficulty: ★★★★☆ Usefulness: ★★★★★
Resources: Dragon Book Section 9.5, or Cooper Book Section 10.3
PRE eliminates partially redundant computations, and provides the additional benefits of Common Subexpression Elimination (CSE) and Loop Invariant Code Motion (LICM). The latter is especially important to reduce loop execution overhead. You can implement the SSAPRE algorithm, but we recommend the simpler alternative using 4 dataflow passes which you read about in the Dragon Book or Cooper Book. A dataflow framework will come in handy here. An alternative to implementing PRE is to implement CSE and LICM as 2 separate passes. Some benefits of LICM over PRE, is that you can hoist expressions to top level loops, and you can hoist effectful operations in order.

(a) **Global value numbering (GVN)** GVN can identify equivalent computations in the code, and can either be a standalone pass or be incorporated into PRE to eliminate more redundant computation. Note that GVN might eliminate some expressions that CSE cannot.

(b) **PRE implementation tips**

i. In some circumstancecs, memory loads and function calls can be PRE candidates too! Think carefully about when these are allowed.

ii. If you did not implement SSAPRE or GVN, your PRE might have trouble finding common subexpressions, so try doing local copy propagation.

iii. To make the dragon book algorithm work you might want to split your basic blocks such that all PRE candidate expressions are locally transparent.

9. **Function Inlining**
   Difficulty: ★★☆☆☆ Usefulness: ★★★★☆
   Inlining a function can reduce the overhead of a call and potentially open up opportunities for more optimizations. The real strength of inlining comes with its interaction with other optimizations. However, inlining can bring problems such as increased code size and register pressure. Choosing which functions calls to inline is often a tradeoff between code size and performance, and you will need some good heuristics. For example, common heuristics include size of the functions, and loop depth of the function call.

10. **Tail Call Optimization (TCO)**
    Difficulty: ★★☆☆☆ Usefulness: ★★★★☆
    Resources: LLVM TailRecursionElimination pass
    TCO turns recursive calls at the end of functions into jumps to reduce the overhead of function calls. You can do TCO on both self recursive tall calls and calls to other functions. You might also find other benefits of turning recursive calls into jumps, such as enabling more inlining opportunities.

    (a) **Basic accumulation** You can also perform accumulation transformations on tail-call expressions when required, thus turning functions such as factorial into a tail-recursive form. This is harder but much more effective on certain benchmarks than basic TCO.

11. **Redundant Safety Check Elimination**
    Difficulty: ★★☆☆☆ Usefulness: ★★★☆☆
    You can implement an SSA-based or dataflow-based approach to eliminate redundant null-checks and array bounds-checks when dereferencing pointers or accessing arrays at runtime. This optimization is specifically tailored towards speedup in safe mode, as these checks can be removed entirely when running with `--unsafe`.

12. **Loop Optimizations**
    Difficulty: ★★★★★ Usefulness: *crucial for programs abundant with loops*
    Again, we order these in order of difficulty.

    (a) **Loop unrolling** Loop unrolling can reduce the overhead of loops and increase the portion of time running useful computation within a loop. While important on its own, unrolling also makes other optimizations such as vectorization and instruction scheduling more effective. Your loop analysis framework will come in handy here, as well as a framework to detect loop indices and basic induction variables. If the unroll factor cannot divide the number of loop iterations, you would need to insert a prologue or epilogue loop (which perhaps could be completely unrolled). You should pick a good unrolling factor and design an unrolling cost model because blindly unrolling loops will bloat up the code size.

    (b) **Induction Variable Elimination (IVE)** You need to first perform Induction Variable Detection which detects induction variables in a loop, and dependence analysis. Then you could perform strength reduction, scalar replacement, and deadcode elimination based on the induction variables. We recommend doing SSA-based IVE. See the lecture slides for more details.

(c) **Loop tiling/fusion/interchange/...** You need good heuristics to perform these optimizations, and their effectiveness is target specific (dependent on the hardware).

2.3 Advanced Analysis and Optimization Passes

Below we list some optimizations that might be beyond the scope of this lab – they are either too hard, or might not affect your score enough to justify the time investment. However, they are all fascinating topics to explore. We recommend you implement these only if you have most of the optimizations in the above section working.

1. **Alias Analysis**
   Difficulty: ★★★★★ Usefulness: ★★★★★
   Resources: Andersen's or Steensgaard's Points-To Analysis
   Making context-sensitive points-to analysis practical for the real world (Lattner 07)
   We recommend Andersen's or Steensgaard's approach as it is simpler to implement, especially Steengaard's approach as it is cheapest. C0 has the additional benefit of being a typed language, where pointer arithmetic is not allowed, and structs and arrays are fundamentally different types. Thus the type information might help you form a better alias analysis algorithm than Steengaard's. Alias analysis will enhance the quality of many of your optimizations, including PRE, LICM, instruction scheduling, among many others.

2. **Interprocedural Optimizations**
   Difficulty: *varies, hard in general* Usefulness: *depends*
   Most optimizations in the above section are intraprocedural, but some passes such as register allocation and alias analysis, can be made more effective when applied across functions. Interprocedural Optimizations generally involve traversing the call graph.

3. **Vectorization using Streaming SIMD Extensions (SSE)**
   Difficulty: ★★★★★ Usefulness: *useful for programs with a lot of parallelism*
   You can take advantage of the X86 SSE, SSE2, or AVX-512 extensions to vectorize loops, like *gcc -O3* does. For L4 grammar, vectorization is much more effective when combined with loop unrolling. This can also be an interesting project for lab6.

4. **Instruction Scheduling (Software Pipelining / Hyperblock or Trace Scheduling)**
   Difficulty: ★★★★★★ Usefulness: *depends on the program and the processor*
   Code scheduling involves moving instructions around to increase instruction level parallelism and reduce pipeline stalls. You might also perform if-conversions (using x86 `cmovs` to form larger blocks. These optimizations are very tricky to get right and are target-specific. Our grading instances use an out-of-order processor which limits the benefit of local scheduling within small basic blocks.
   The scheduling algorithm we recommend is *list scheduling*, which occurs within a basic block. This optimization is made more effective by alias analysis, and loop unrolling.

3 Testing

As you are implementing optimizations, it is extremely important to carry out **regression testing to make sure your compiler remains correct**. We heavily recommend that your optimizations be modular, and that correctness does *not* depend on a particular previous optimization. We will call your compiler with and without the `--unsafe` flags at various levels of optimization to ascertain its continued correctness, though your performance will be evaluated primarily through our benchmarks.

To enable you to perform more compiler optimizations, we increased COMPILER_TIMEOUT of the autograding harness to 20 seconds. We also increased the RUN_TIMEOUT of the executable produced by your compiler to 120 seconds.

To help you test your performance, you'll see some new files in the `dist` repository:

- `tests/bench/`, which contain the benchmark programs.
- `timecompiler`, a script which counts the cycles of your compiler on these benchmarks.
- `score_table.py`, a script which you can use to generate a table of your score. You can also see the times and code sizes of the executables produced by `cc0` and `gcc`, and the formula we use to compute your multiplier on Gradescope. Read the comments within `score_table.py` on how to use this script.

To use the `timecompiler` script, you should follow these steps:

1. Ensure your compiler supports `--unsafe` and `-O1`.
2. If you wish, add additional benchmarks to the benchmark folder.
3. Run `../timecompiler` from your compiler's directory. `timecompiler` accepts the same flags that gradecompiler does – however, we don't recommend running the benchmarks in parallel.

Here is a cheatsheet of useful commands:
`timecompiler bench`:
this will time your compiler on the benchmarks in the `../tests/bench` folder, and print out the cycles and code sizes for each benchmark
`timecompiler -q --autograde`:
the `-q` flag suppresses unhelpful output, and `--autograde` will print a json array of the cycles and code sizes at the end, which you can pass to `score_table.py`
`bin/c0c -ex86-64 -O1 --unsafe ../tests/bench/daisy.l4`:
this generates a `.s` file from your compiler
`gcc -m64 -no-pie ../runtime/run411.o ../tests/bench/daisy.l4.s`:
using the `.s` file, you can link to our runtime file to generate an executable
`gcc -m64 -no-pie ../runtime/bench.o ../tests/bench/daisy.l4.s`:
alternatively, you could link to `bench.o` to generate an executable that will run the benchmark numerous times, and print out the average of the k best times
`gcc -O1 -fno-asynchronous-unwind-tables -S ../tests/bench/unsafe/daisy.c`:
this uses `gcc` to generate a `.s` file in the current directory, which you can use to compare your own assembly against
`gcc -O1 ../runtime/run411.o ../tests/bench/unsafe/daisy.c`:
alternatively, you can let `gcc` directly generate an executable

4 Deliverables and Deadlines

For this project, you are required to hand in a complete working compiler for L4 that produces correct target programs written in Intel x86-64 assembly language, and a description and assessment of your optimizations. The compiler must accept the flags `--unsafe` and `-O`n with $n = 0, 1$. When we grade your work, we will use the `gcc` compiler to assemble and link the code you generate into executables using the provided runtime environment on the lab machines.

Note that for the benchmarks, we will call not just the `_c0_main` function in the assembly file you generate, but also four other functions in order to obtain cycle counts that are as precise as possible. These are `_c0_init`, `_c0_prepare`, `_c0_run`, and `_c0_checksum`, each corresponding to their un-prefixed counterparts in the benchmark source. Given this, it is critical that your code follow the standard calling conventions and function naming conventions from Labs 1–4 for these functions.

Compiler

The sources for your compiler should be handed in via Gradescope as usual, and *must* contain documentation that is up to date. Particularly, your compiler should document each of your performed optimizations in both a `README` file and the source itself. The course staff *will* be reading your code as part of the submission for this lab. You may use up to five late days for the compiler.

Project Report

The project report should be a PDF file of approximately 4–5 pages (possibly more, particularly with figures), and should be handed in on Gradescope. Your report should describe the effect of `--unsafe` as well as your optimizations and other improvements,and assess how well they worked in improving the code, over individual tests and the benchmark suite.

At the *absolute minimum*, your project should present a description and quantitative evaluation of the optimizations you performed at the `-O0`, `-O1`, and default levels. A good report must also discuss the way your individual optimizations *interact*, backed up by quantitative evidence. (Tables are a good idea. Graphs are an even better idea.) Make sure to carefully document how your got your numbers; someone with access to your code should, if they're willing to buy whatever hardware and operating system you were using, be able to replicate your results. A good report should also spend some time describing the effect of individual optimizations on the code you produce.

Your report should contain a specific commit hash for the course staff to review[2], and a comprehensive descriptions of where in your source files each optimization you have described is implemented. If you use algorithms that have not been covered in class, cite any relevant sources, and briefly describe how they work. If the algorithms have been covered in class, cite the appropriate lecture notes or paper, and focus on any implementation choices you made that are not described in those resources.

Other (optional) discussions that might be included in a high-quality report include:

- Effects of the ordering of different optimization passes.

[2]It is not necessary that this be the same commit hash that you submit to Gradescope. In fact, we encourage you to perform any code-cleanup that may be required.

- Time versus space tradeoffs in emitted code.
- Effects of various optimizations on the running time of your compiler.
- Examples of programs that your optimizations would interact particularly well with.
- Examples of programs that your optimizations would interact particularly poorly with.

Grading

This assignment is worth 150 points. The written report is worth 50 points. The remaining 100 points will be based on the correctness of your compiler and on the performance of your emitted code relative to our benchmarks, as reflected by your Gradescope score.

Your performance will be measured as a factor of how far between `gcc`'s `-O0` and `-O1` your generated assembly is. We do not expect you to achieve true parity with `-O1` in a single semester, which would be a tremendous feat. But you may be surprised by how close your compilers already are. We hope the comparison offers you a sense of how much you have done this semester! :)

We will run your compiler numerous times in all of the optimization modes, and take the k-best times of all of these. We will use the benchmark score as a multiplier for your correctness score, and we derive your multiplier by averaging your score over all of the benchmark tests. The scheme is designed so that you do not have to exceed `-O0` on all tests, and may benefit from a score greater than 1 if your optimizations provide excellent speedup in certain cases.

The exact formula for each test is as follows: t_c is the average of the k-best times from your compiler[3], t_0 and t_1 denote the times of the `cc0` reference compiler using `-O0` and `-O1` respectively. The u variables denote the same times, but with `--unsafe` and using `gcc` as reference. The variable $f = 0.10$ is a difficulty reduction factor selected by the course staff. **Both P_s and P_u is clamped between 0 and 2.0.** T is our benchmark suite, and M is your multiplier for the lab.
For running time of a benchmark:

$$P_s = 1 + f - \frac{t_c - t_1}{t_0 - t_1} \qquad P_u = 1 + f - \frac{u_c - u_1}{u_0 - u_1} \qquad P_{time} = \frac{P_s + P_u}{2}$$

Overall score multiplier:

$$M = \frac{\Sigma_{bench \in T}\ P_{time}}{|T|}$$

C is a correctness score based on your performance in the general Labs 1–4 testsuite. Your compiler will be evaluated with your `O1` optimisations, and will be given a 20 second compilation timeout (but no change to the runtime timeout).

We will also be running your compiler both with and without the `--unsafe` flag. We will run `--unsafe` on tests with directives `//test return i`, `//test abort`, and `//test compile`. We will run `--safe` on tests with directives `//test div-by-zero` and `//test memerror`. Tests with `//test typecheck` and `//test error` will be skipped and not be considered.

Your correctness score C will be calculated using a similar formula to prior labs:

$$\text{subtotal} = 30 * \frac{\text{passed basic tests}}{\text{total basic tests}} + 70 * \frac{\text{passed large \& only tests}}{\text{total large \& only tests}}$$

$$\text{total} = C = \text{subtotal} - (1 \text{ point per compiler failure}) - (0.1 \text{ points per executable timeout})$$

[3] where $k = 1$

Your final score is then simply MC.

You will be able to get **extra credit** on this assignment! If your final score is above 100 points, you will receive $min(20, \frac{MC-100}{2})$ extra credit points for building an excellent optimizing compiler.

5 Tips and Hints

1. You have less than three weeks to work on this lab, and implementing optimizations is a lot of work, so start early!

2. The Godbolt Compiler Explorer (godbolt.org) is a really helpful tool for comparing your compiler against gcc/clang.

3. You should make a habit of closely inspecting the assembly outputted by your compiler, comparing it to gcc/clang's output, identifying inefficiencies in your code, and thinking about the possible optimizations to address those inefficiencies.

4. Early in your implementation process, you will likely find most of the benefit to come from removing local inefficiencies, accumulated from previous labs. However, the more interesting and rewarding experiences lie in implementing the global optimizations.

5. Some of you might find certain global optimizations to be not useful in improving your score. This is almost always because your implementation is either flawed, buggy, fails to cover important cases, or needs to interact properly with another optimization. We have carefully designed our benchmarks so that all global optimizations mentioned in this handout, when designed and implemented correctly, can provide a boost to your score.

6. In the final stages, when you are done with most of your optimizations, you can try optimizing for the *hot path* of benchmarks. Often, a few functions or loops on the *hot path* of programs dominates the program's run time, and the largest benefit will come from optimizing them.

7. Since you will likely perform most optimizations on some IR form, compiler utilities and flags to print out code in that IR can be really helpful for debugging, as is using graphviz and dot to generate visual representations of control flow graphs, which will also help you when writing your report (you can use graphviz online here).

8. The ordering of optimizations and analysis passes can be extremely important, and you should explore how different optimizations interact. It is often worth it to perform a certain pass multiple times (before and after related passes).