15-411: Compiler Design — Fall 2021

Recitation 8: Optimizations — 12 November

1 Optimizations

In the following sections, we will provide you with list of suggested analysis and optimization passes you can add to your compiler. This is a long list and we obviously 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 the `cc0` reference compiler and `gcc -O1`. We list the course staff recommended difficulty and usefulness rating (your experience may vary) of optimizations to help you decide which passes to implement first.

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), and the SSA Book (SSA-Based Compiler Design), all of which have great 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 the source.

1.1 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.

(a) **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.

(b) **Dataflow Framework**
Difficulty: ★★☆☆☆ Usefulness: ★★★★☆

The Dataflow Framework we made you do in the l2 checkpoint is not only useful for liveness analysis, but also for passes such as alias analysis, 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.).

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

(d) **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, Redundant Safety

Check Elimination, 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.

(e) **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 inter-procedural analysis you can perform.

(f) **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, nested depth, and other loop features. You might also consider adding preheader blocks during this pass.

(g) **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. You can use an SSA-based or dataflow-based approach. Value Range Analysis can make other optimizations more effective, such as SCCP, Strength Reduction, and Redundant Safety Check Elimination.

1.2 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.

(a) **Cleaning Up Lab3 & Lab4**
Difficulty: ★★☆☆☆ Usefulness: ★★★★★

Before performing global optimization passes, we suggest inspecting the x86 assembly code output of your compiler as you did for l5 checkpoint, and finding opportunities for improvement. 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 (try comparing your assembly to gcc/clang output), or fixing too many registers in codegen and not making full use of your register allocator.

(b) **Strength Reduction**
Difficulty: ★★☆☆☆ Usefulness: ★★★★☆
Resources: Division by Invariant Integers using Multiplication (Granlund 91)
Hacker's Delight 2nd Edition Chapter 10
Strength reduction 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). Getting the magic number forumlas for division and modulus is tricky, and we recommend you read the above resources, or look at how GCC/LLVM implements strength reductions.

(c) **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.

(d) **Improved Register Allocation**
Difficulty: *varies* 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
A good register allocator is essentially for code optimization, and below we provide a list of possible extensions (ordered roughly in increasing difficulty) to the register allocator we had you build for the L1 checkpoint, which is far from perfect. We highly recommend at least implementing coalescing, and the rest is up to you.

(a) **Coalescing** We recommend optimistic 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.

(b) **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.

(c) **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.

(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 more effective, and with SSA you get a chordal interference graph. However, it turns out that getting out of SSA is a bit 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.

(e) **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.

(f) **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.

   (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.

(g) **Aggressive Deadcode Elimination (ADCE)**
Difficulty: ★★☆☆☆ Usefulness: ★★★☆☆
Similar to SCCP, 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.

(h) **Partial Redundancy Elimination (PRE)**
Difficulty: ★★★★☆ Usefulness: ★★★★★
Resources: Dragon Book Section 9.5, or Cooper Book Section 10.3, or SSA Book Chapter 9 (Redundancy Elimination) for SSAPRE
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. Your dataflow framework from L2 checkpoint will come in handy here. An alternative to implementing PRE is to implement CSE and LICM as 2 separate passes.

   (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.

(i) **Function Inlining**
Difficulty: ★★☆☆☆ Usefulness: ★★★☆☆
Inlining a function can reduce the overhead of a call and potentially open up opportunities for more optimizations. However, it 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.

(j) **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 also perform accumulation transformations on tail-call expressions when required. You might also find other benefits of turning recursive calls into jumps.

(k) **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`.

(l) **Loop Optimizations**
Difficulty: *varies, hard in general* Usefulness: *useful for programs abundant with loops*

  (a) **Loop unrolling/tiling/fusion/interchange/...** You need good heuristics to perform these optimizations, and their effectiveness is target specific (dependent on the hardware). Your loop analysis framework will come in handy here.

  (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.

1.3 Advanced Analysis and Optimization Passes

Below we list some optimizations that might be beyond the scope of this project - they are either too hard, or might not affect your score enough to justify the time investment. We only recommend you to implement these once you have most of the optimizations in the above section working.

(a) **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, though you could also refer to Lattner's paper and LLVM's cheaper alias analysis approach. Note that you are allowed to assume the *strict aliasing rule* of C, that the function pointer arguments in a function are assumed not to alias if they point to fundamentally different types. Alias analysis will enhance the quality of many of your optimizations, including ADCE, PRE, redundant store elimination, instruction scheduling, among others.

(b) **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.

(c) **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. This can also be an interesting project for lab6.

(d) **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 to form larger blocks. These optimizations are very tricky to get right and are target-specific.

2 Tips and Hints

(a) Implementing optimizations and SSA is a lot of work, so start early!

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

(c) As in the Lab 5 checkpoint, 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.

(d) That being said, optimizing for a specific line in a specific function in a specific benchmark might not be a good strategy, as it is unlikely to affect your score by much. Global optimizations that eliminates efficiencies across the board are likely a better investment of your time.

(e) Since you will likely perform most optimizations on SSA form or some other IR form, compiler utilities and flags to print out code in that IR can be really helpful for debugging, as is using dot to generate visual representations of control flow graphs (you can download graphviz here: https://graphviz.org/)

(f) 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).

(g) If testing locally on a Mac outside of a docker container, you may get a slightly worse code size score than you would on autolab or in a docker container. This is because native MacOS uses the Mach-O executable format which rounds the size of the code segment up to a multiple of 4096 whereas the Linux ELF format does not round up at all. We strongly recommend testing for code size inside of a docker container on MacOS – otherwise, you will not be able to measure code size changes less than 4096 bytes.

(h) Don't name any local folders `bench`. Our autograder defaults to searching for benchmarks