Hulk Compiler Report (15-411 F21)

Jeff Tan (jefftan), Rachel Yuan (rachely)

December 15, 2021

1 Introduction

In this report, we discuss our findings from implementing the `-O1` and `--unsafe` flags for our compiler targeting C0, including alias analysis for our advanced optimization.

1.1 Compiler structure

Our compiler follows the same overall structure as Lab 4, with minimal changes to the parser and front-end. We begin by parsing the C0 source into a typed parsetree using an `LR(1)` parser generated by Menhir. Then, we perform typechecking on this tree and calculate datatype information such as size and struct offsets. Next, we translate the parsetree to an untyped AST, add size information as annotations, nd perform some simple elaboration steps such as converting `for` loops to `while` loops and elaborating short-circuit boolean operators to ternaries. Finally, we perform code generating using a convenient munch algorithm to flatten the reduced AST into basic instruction sequences, which are manipulated throughout the optimization process.

To facilitate various optimizations, we implemented several analysis passes to transform the code between representations:

- Control flow graph (`cfg.ml`)
- Dataflow framework (`dataflow.ml`)
- Dominator tree (`dominance.ml`)
- Static single assignment (`ssa.ml`)

We also implemented many optimization passes. Most of these act on the control flow graph (CFG), except for some global optimization passes such as function inlining and tail call optimization:

- Local copy propagation (`copyprop_local_ssa.ml`)
- Local constant propagation (`constprop_local_ssa.ml`)
- Constant folding (`constfold_ssa.ml`)
- (Agressive) Dead code elimination (`adce.ml` / `dce.ml`)
- Register allocation and coalescing (`regalloc.ml`)

- Peephole optimizations (`peephole_ssa.ml`)
- Integer divide strength reduction (`reducediv_ssa.ml`)
- Partial redundancy elimination (`pre.ml`)
- Function inlining (`function_inlining.ml`)
- Tail call optimization (`tail_call.ml`)
- Miscellaneous improvements
  - Branch fallthrough
  - Calculating memory addresses using `lea`

We also chose to implement alias analysis, of which we have two types. The flow-insensitive Steensgaard's algorithm uses a union-find data structure to find points-to relationships between temps and variables. We also use flow-sensitive dataflow analysis to find what variables and temps possibly alias each other at each line in the program:

- Steensgaard's algorithm (`point_to.ml`)
- Dataflow-based alias analysis (`pointer_analysis.ml`)

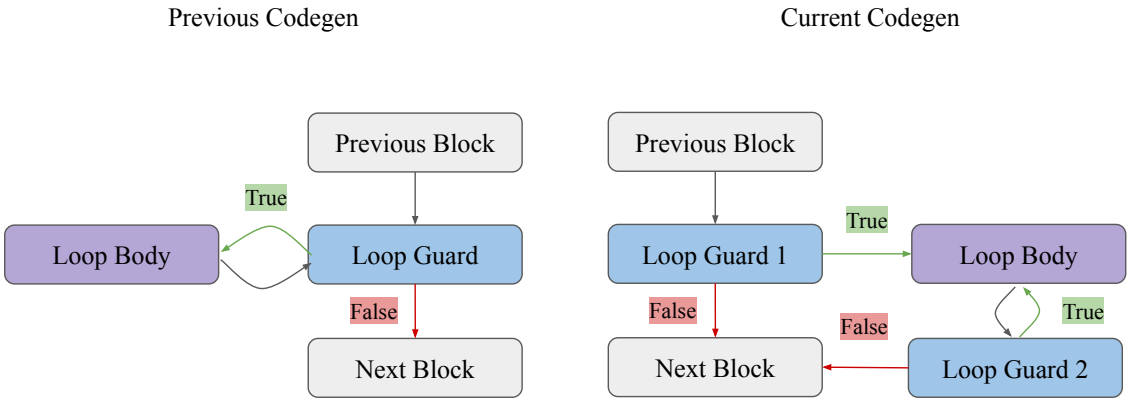
2 Frontend

In general, we made very few changes to our compiler frontend. The biggest change was implementing the `--unsafe` flag which removes all error checks from the codegen, including memory null checks and array bounds check, as well as bounds checks for left and right shift. Under unsafe mode, we also no longer need to store the length of an array after allocation, we can simply return the result of `calloc` directly.

During initial code generation, we focused much more heavily on simplicity over performance, leaving optimizations to be performed in future passes. Our codegen almost exactly follows the dynamic semantics of C0: for example, we generate a new temporary variable for every argument of a binop or function call. While this makes our frontend easier to implement and verify, it is also wasteful and increases our compiler runtime by creating extra work for the optimization passes.

2.1 Supporting partial redundancy elimination

To better support partial redundancy elimination (PRE) and loop-invariant code motion, we also duplicated the loop guard in our `while` loop codegen. Shown below is a comparison of our previous and current codegen:



Under our previous codegen, PRE was not very effective because the algorithm only moves statements to a point where they are guaranteed to be executed regardless of what path is taken to the exit. We cannot hoist loop-invariant statements outside the loop body and before the loop guard, as these statements might not be computed if the loop guard immediately returns false. Our current codegen fixes this issue by duplicating the loop guard, allowing loopinvariant statements to be placed between Loop Guard 1 and Loop Body. The loop is guaranteed to run once upon reaching this point, so loop-invariant statements can be safely hoisted out. Although this slightly increases code size by generating the loop guard twice, it is well worth it to enable PRE.

3 Optimization Passes

Our optimization passes are broadly defined in the `optimization.ml` file. We first perform tail call optimization and function call inlining on the abstract assembly outputted by the parser. Then, we convert each function's assembly to a control-flow graph (CFG) and convert the CFGs to chordal static-single assignment (SSA). We run our optimizations in the following order:

1) Tail call optimization (global)
2) Function call inlining (global)
3) Local constant propagation
4) Constant folding
5) Local copy propagation
6) Division reduction
7) Peephole optimizations
8) Aggressive dead code elimination
9) Coalescing CFG blocks
10) Partial redundancy elimination

These optimizations are repeated in a loop until the number of instructions converges and further optimizations have no additional effect. We also have a timeout to stop the compiler if compilation takes too long. After optimization, we pass the optimized CFG to liveness analysis and register allocation, then unravel the CFG back into a list of abstract assembly instructions to be converted to x86 assembly.

3.1 Constant/Copy Propagation and Constant Folding

We implement local constant and copy propagation by iterating through each basic block and keeping a local context that maps destination variables and temps to the source variables or constants that they refer to. First, we use information from the context to populate the right-hand side of each instruction (moves, binops, comparisons, function calls). For example, the line `x1<-x2` with `x2:=6` in the context becomes `x1<-6`. If the instruction assigns a variable, temp, or constant to another variable or temp, we add this assignment to the context and update the instruction. In this case, we add `x1:=6` to the context and change the instruction to `x1<-6`. Note that we do not remove any instructions from the code during constant and copy propagation: these are cleaned up later by (aggressive) dead code elimination (ADCE).

After constant propagation, we do constant folding. If both sides of a binop or comparison are constants, we evaluate the expression and replace the binop or comparison with a move. We replace any conditional jumps with deterministic jumps if we know the result of the condition, and if we know that there will be a division error, we remove the invalid expression and explicitly raise a floating point exception. Finally, we clean the CFG by removing unreasonable instructions and updating predecessor and successor blocks as necessary.

These optimizations significantly reduce the number of binops and comparisons, and provide a baseline so that we can remove unnecessary moves later on through dead code elimination. Given that constant propagation, copy propagation, and constant folding run in linear time, we also run these optimizations under the `-O0` optimization setting.

These optimizations also have the potential to reduce register pressure during register allocation, by eliminating the excess number of temps created during codegen. This significantly reduces the runtime of our register allocator by decreasing the size of our interference graph, as building the interference graph is $O(n^2)$.

Without dead code elimination however, constant and copy propagation increase register pressure, as propagation does not remove the unnecessary moves and increases the live range of variables and temps. We get a much worse speedup overall with just propagation and folding. This is demonstrated further below.

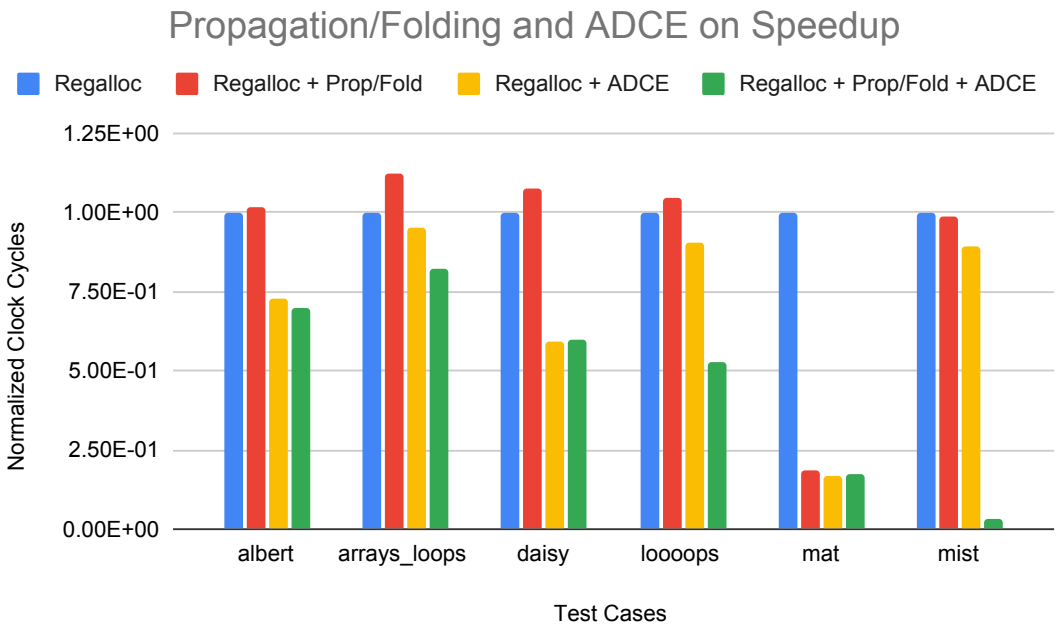
3.2 (Aggressive) Dead Code Elimination

We implement both aggressive and standard dead code elimination based on the Mark and Sweep algorithm in Section 10.2.1 of the Cooper book. Dead code elimination uses the same algorithm as ADCE, but it marks all conditional jumps as critical, and since no jumps are removed, there is no need to find the reverse dominance frontier.

Dead code elimination is used in place of aggressive dead code elimination if a function has an infinite loop. In this case, there are no ways to reach the exit, and thus, there is no way to form the reverse CFG. In this event, aggressive dead code elimination throws an exception and we default to dead code elimination.

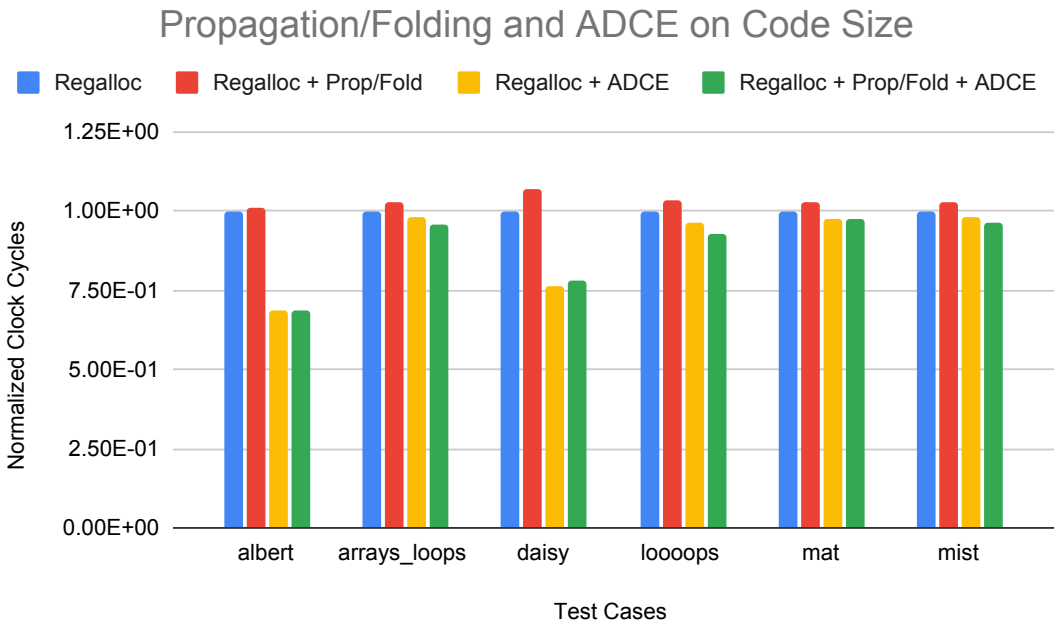
Aggressive dead code elimination significantly improves the quality of our compiler by removing dead code, as well as cleaning up after our other local optimizations. While ADCE may remove some loop blocks that have no effect on the program output, there are many moves of the form (`y<-x; z<-y`) that could be coalesced together into (`z<-x`). Propagation and folding help to merge these moves and make them redundant, and then ADCE actually removes them from the program.

These effects are demonstrated below on a subset of tests, run on the lab 5 benchmarking submission. All results were run with register allocation and some basic peepholes that are built into the compiler (e.g. fallthrough, lea, etc.). These optimizations were only applied once, rather than in multiple iteration. The results are for safe mode:



Aggressive dead code elimination shows tremendous improvement in the `mist.l4` benchmark, but only when combined with propagation and folding. This test case contains a loop on lines 23 to 27 that has no effect on the result of the mist function. However, we only know after constant propagation that this loop has no effect, as the calculation of `i/2` becomes (`t1<-i; t2<-2; t3<-t1/t2`) as a result of codegen. Aggressive dead code elimination then removes this loop.

In addition to speedup, we get significant reductions in code size due to a combination of propagation/folding and ADCE. Propagation and folding alone actually increases the code size, since these local optimizations do not actually eliminate any instructions. ADCE alone has significant impacts on `albert.l4` and `daisy.l4`, and smaller impacts on the other tests. However, we see the best overall impact with propagation/folding and ADCE in combination:



3.3 Peephole Optimizations in SSA

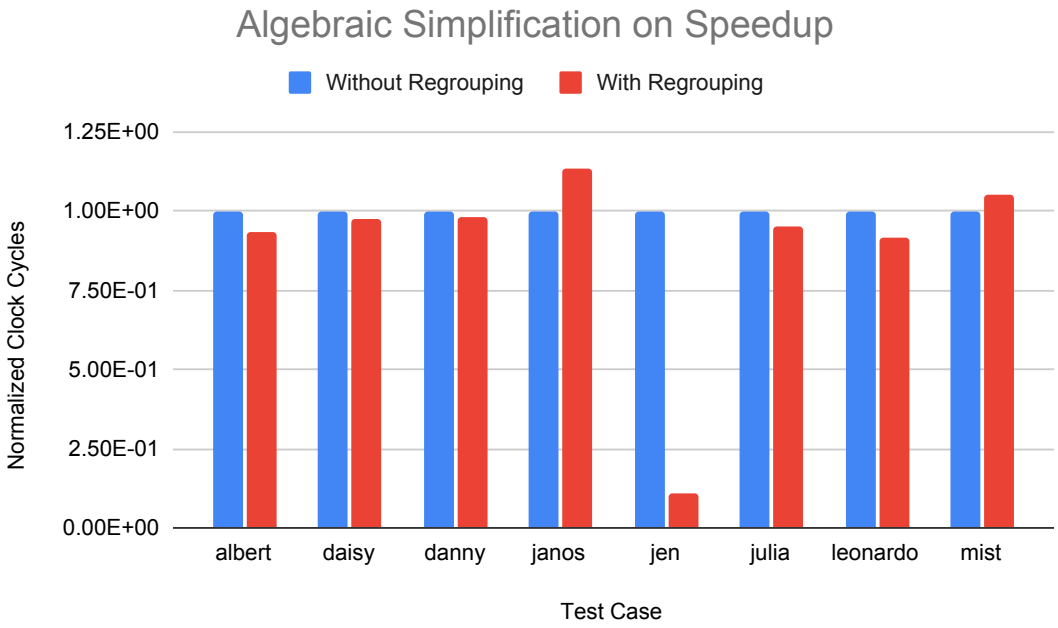
Our compiler implements a number of local peephole optimizations within basic blocks. Each peephole optimization acts on a single instruction at a time, and we maintain a local context that maps each variable to the expression that it currently holds.

3.3.1 Constant Regrouping and Algebraic Simplification

For all associative binary operations, we use the associative property to regroup constant expressions and achieve better constant folding. For example, if (`x1<-c1+x2`) and (`x2<-c2+y2`), we can rewrite `x1` as (`c1+(c2+y2)=(c1+c2)+y2`) and perform constant folding to eliminate an addition operation. The same can be done for multiplication, and, or, and xor. We also simplify algebraic expressions when possible, for example by eliminating addition or subtraction by 0, subtraction by yourself (e.g. `x-x`), multiplication by -1, 0, or 1, as well as shifts by 0. Many of these peephole optimizations are uncovered after applying other optimizations such as copy propagation and constant folding.

Similar to the constant folding pass, this optimization does not actually remove any instructions from the abstract assembly, which we leave to ADCE. We simply perform a lookup in the local context to identify the underlying expression of the left and right arguments, and replace the operands of the current binop if eligible.

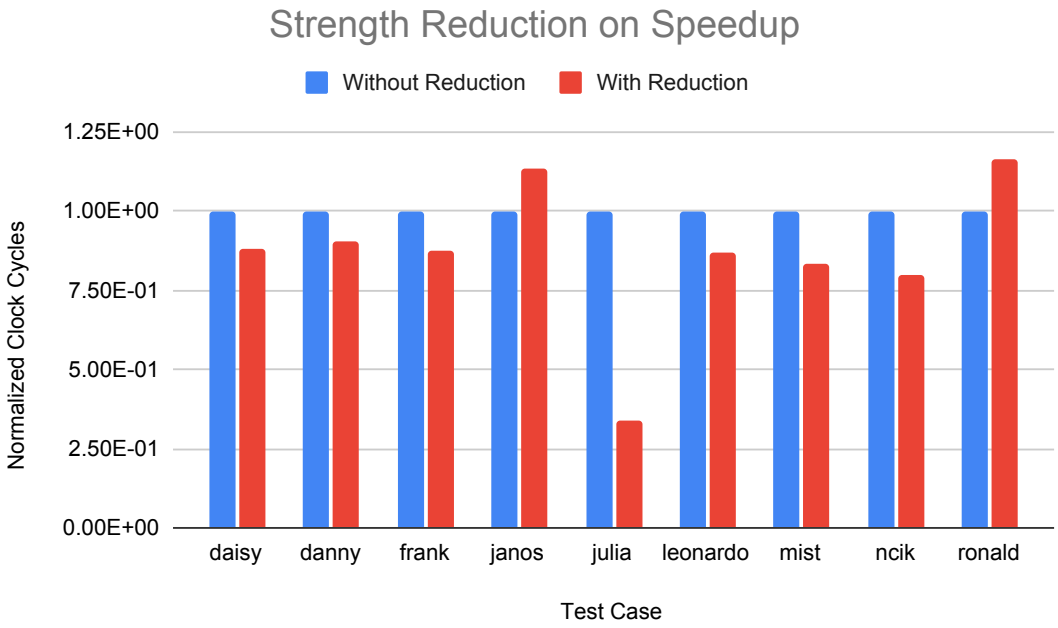
Below, we show the impact of running our fully iterated optimization passes in unsafe mode with and without algebraic regrouping. We perform slightly better on some tests such as `julia.l4` and `leonardo.l4`, and significantly better on `jen.l4` which has a significant amount of arithmetic computation. Unfortunately, constant grouping and simplification also slightly increases the runtime on `janos.l4` and `mist.l4`, possibly due to poorer register allocation from increased live ranges of variables. Despite this, we believe that the positive impacts are worth it.



3.3.2 Strength Reduction

We also use strength reduction to replace expensive operations such as multiplication and division with cheaper operations when possible. We use LEA to multiply by certain supported constants (1, 2, 3, 4, 5, 8, 9), we use left shift to multiply by powers of two, and we also implement the algorithm described in Granlund 91 to replace division by constants with multiplication and other cheap operations. This optimization brings significant performance improvement, especially when the expensive instructions are in an inner loop or on some kind of critical path.

Shown below is the result of locally running our fully iterated optimization passes in unsafe mode, with and without strength reduction, where we omit any benchmarks where there was no difference. We perform slightly better on some benchmarks like `leonardo.l4` and `ncik.l4`, and significantly better on `julia.l4` which performs division in an inner loop. As Granlund 91 creates many more temporary variables than the division it is replacing, we may get worse performance if strength reudction increases register pressure and causes spilling.



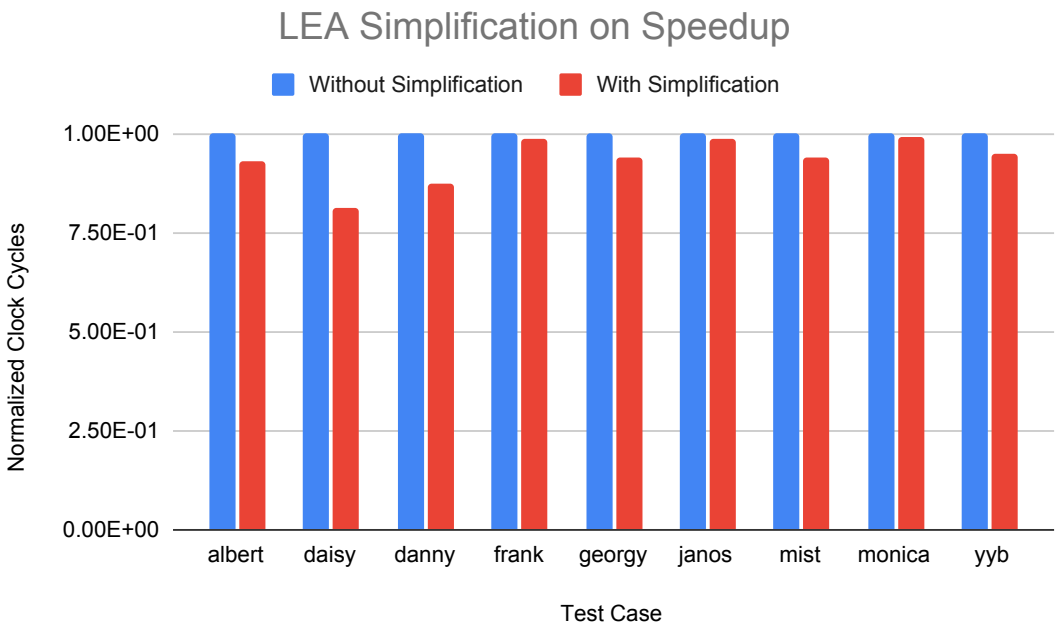
3.3.3 LEA Simplification

Finally, we perform a number of simpliications on LEA instructions to reduce the cost and register requirements of address computations. As our codegen exactly follows the dynamic semantics of C0, performing a constant index array access such as (`a[5]`) without optimization requires us to store 4 into a temp, use a LEA instruction to store the address (`a + 5 * 4`) into a temp, and dereference this temp. These three steps can be combined into a single step that directly uses (`20(a)`) as a memory operand.

When the base of a LEA expression is another LEA expression with no index component, we can merge the two expressions together: for example, (`x1<-lea o1(x2,i1,s1)`) and (`x1<-lea o2(b2)`) becomes (`x1<-lea o1+o2(b2,i1,s1)`). When there are any constant terms inside a LEA expression, we can eliminate those and combine them into the offset term: for example, (`lea 4(b,3,8)`) is equivalent to (`lea 4+3*8(b)`) or (`lea 28(b)`). We also merge LEA instructions into moves when possible: for example, (`x1<-mem(x2)`) and (`x2<-lea o2(b2)`) is equivalent to directly accessing memory using a memory operand, (`x1<-mem o2(b2)`).

Note that LEA simplification provides much less benefit to our compiler in unsafe mode, as we did not implement redundant safety check elimination and must evaluate the relevant LEA expressions anyways to check them against NULL.

Shown below is the result of our running our full iterated optimization passes with and without LEA simplification, where we omit any benchmarks where there was no difference. We perform slightly better on some benchmarks like `albert.l4` and `daisy.l4`, where there is a high intensity of memory accesses.



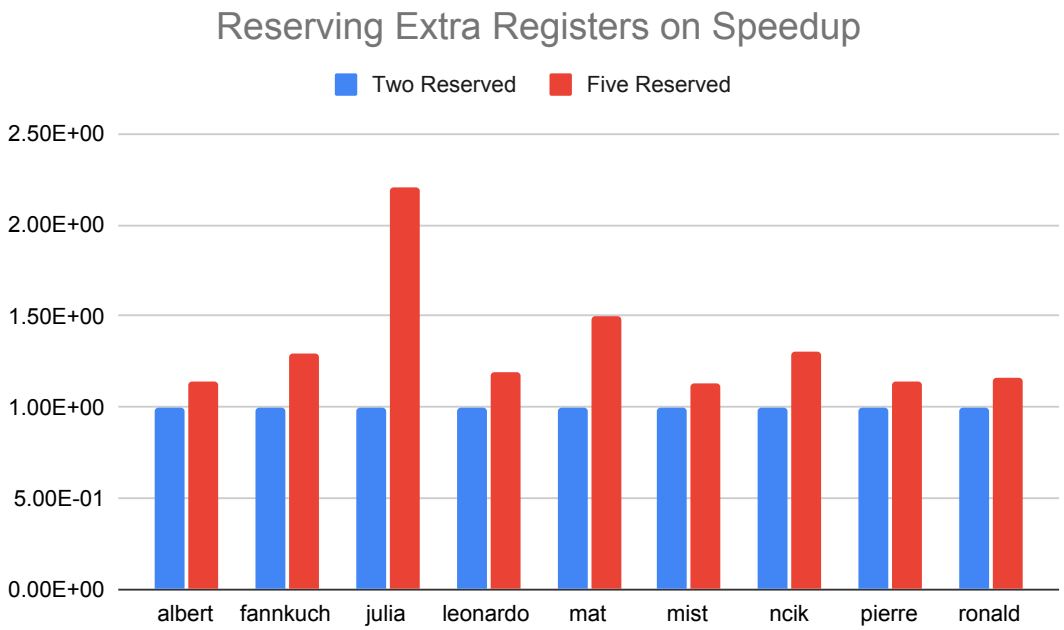
3.4 Register Allocation

We use the SSA-based register allocator based on chordal graphs that was presented in lecture. In this algorithm, we use dataflow to compute liveness, build an interference graph, find the simplicial elimination ordering (SEO) using the maximum cardinality search (MCS) algorithm, perform prespilling, and finally determine a greedy coloring and perform coalescing. After mapping each variable and temp to an integer color, we assign each color to a register or a slot on the stack, where lower-valued colors are more frequently used and thus assigned to registers. Finally, we filter out any redundant moves that may have resulted from two variables being mapped to the same register.

3.4.1 Reserved Registers

We reserve the two registers `RBP` and `R13` for use during instruction selection in order to handle spilled registers. Originally, in our Lab 4 compiler, we had reserved many more registers including the use of RBP as a base pointer for the stack, but for this lab we tried hard to minimize the number of registers we reserved as much as possible to maximize the effectiveness of our register allocator.

Below, we show the effect of reserving two registers vs. reserving five registers on speedup. Each additional register we reserved gave a significant performance penalty, especially on benchmarks with an already-high register pressure such as `arrays_and_loops.l4`, `jack.l4`, `julia.l4`, due to the cost of spilling and reloading values from memory. In fact, when just 3 additional registers are reserved, `julia.l4` takes twice as long. However, on tests where there are enough registers anyways to accommodate all the temps in the program, reserving additional registers has no effect:



3.4.2 Spill Cost Estimation

In order to break ties during the maximum cardinality search (MCS) algorithm, we use heuristics to estimate the spill cost of each vertex. As we did not implement a loop analysis framework, we were not able to consider the loop depth as part of our heuristic. Rather, we approximated the spill cost of each temp by the number of uses it has, measured by counting the number of times that each variable is defined or used at each line in liveness. This way, variables that are more commonly used are more likely to be placed early in the MCS algorithm, meaning that they are more likely to be assigned to registers. Considering loop depth or real-world usage statistics would likely improve the quality of our spill cost estimation.

3.4.3 Prespilling

We also use prespilling to improve the quality of our spills with the maximal clique algorithm. We iterate through the simplicial elimination order (SEO) and for each vertex, we form a clique conisting of that vertex and any neighbors of that vertex which came beforehand in the SEO ordering. After computing the result of the maximal clique algorithm, we count the number of cliques that each vertex is in and repeatedly spill the vertex that is in the most cliques until all cliques have fewer than K nodes, where K is the number of available registers for coloring. At this point, the resulting graph is guaranteed to be colorable with registers only. Once we perform a greedy coloring for these vertices, the prespilled vertices are assigned to slots on the stack.

3.4.4 Coalescing

Finally, we implement register coalescing to merge together move-related edges that do not otherwise interfere with each other in the interference graph. Coalescing helps ensure that as many

variables and temps are allocated to registers as possible, and helps to reduce spilling and reduce the number of redundant moves. To perform coalescing, we pick two vertices that are lined by a move-related edge and search for a register that neither vertex is adjacent to in the interference graph. To avoid creating a spill due to coalescing, we will only coalesce two vertices into a register, not a stack slot. This means that coalescing is less effective in programs with a lot of register pressure, as it is unlikely that there is an available register that neither vertex shares.

3.4.5 Register Allocation Results

Shown below is a summary of the output of our register allocator:

Test Case	Temps / Vars	Coalesced	Spilled
`albert`	1463	43	92
`arrays_loops`	296	28	15
`daisy`	2292	139	153
`danny`	1477	136	0
`fannkuch`	264	32	2
`frank`	409	23	0
`georgy`	1438	118	0
`jack`	795	48	0
`janos`	298	33	0
`jen`	191	52	0
`julia`	451	46	61
`leonardo`	90	8	0
`loooops`	6	21	0
`mat`	253	25	0
`mist`	185	17	0
`monica`	83	8	0
`ncik`	620	55	58
`pierre`	116	25	0
`ronald`	411	33	89
`yyb`	343	36	0

3.5 Function Inlining and Tail Call Optimization

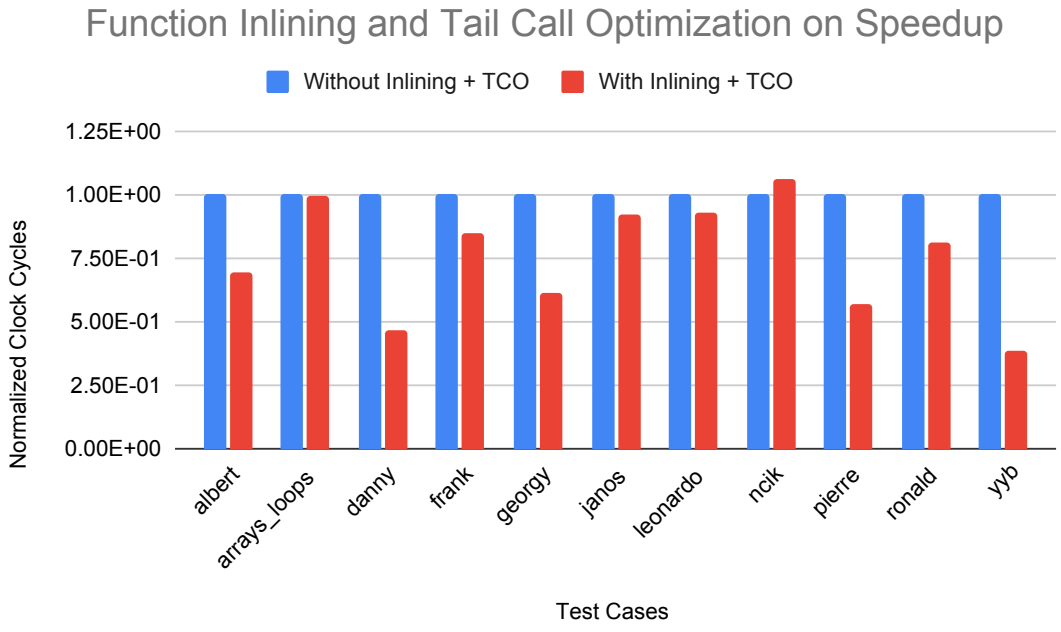
Function inlining is one of two global optimizations we implement, and for simplicity we directly perform it on the abstract assembly before converting it to a control-flow graph (CFG). This allows us to simply replace the Call instruction with the inlined function's abstract assembly, without the hassle of renaming labels, dealing with branches in a CFG, and maintaining SSA. The only thing we have to do is to rename all the variables by prepending the name of the function to each inlined variable. At the beginning and end of the function call, we also generate instructions that move everything into the function arguments, and move the end result into the function's destination instead of the `%rax` register.

We developed several function inlining heuristics to decide which functions to inline. We always choose to inline functions that are only called once, as this will almost always reduce the code size. We also choose to inline functions that are sufficiently small but used quite often (abstract assembly is less than 30 lines), as doing so reduces the overhead of making a function call.

Similar to function inlining, we do tail call optimization directly on the list of abstract assembly instructions before generating a CFG and converting to SSA. This allows tail call optimized functions to also be inlined in the next step. We only use simple tail call optimization, where the function must call itself at exactly the last line in the assembly. To perform tail call optimization, we move all function arguments into the appropriate variables and replace the call with a jump to the beginning of the function.

Below, we demonstrate the effects of function inlining, which significantly improves speedups across a wide range of test cases. Aggressive function inlining allows local optimizations to act across functions, which uncovers many further optimizations. Although aggressive inlining may cause an increase in register pressure, this seems to only be a concern in specific benchmarks with an already-high register pressure, for which inlining can cause a critical variable to spill and greatly decrease the code's performance. All runs below include copy and constant propagation, aggressive dead code elimination, and register allocation, which are applied after function inlining.

Function inlining helps with test cases like `albert.l4` and `frank.l4` where there are a large number of function calls. Tail call optimization is particularly beneficial in `pierre.l4` which contains the tail call optimizable function `pow`. As `pow` is a small function and only called three times, we see additional benefits when tail call optimization is combined with function inlining. Tail call optimization also sees a benefit in `julia.l4` as the recursive Collatz function is tail call optimizable. As we do not have basic accumulation, we do not have the means to transform functions that are very close to tail call optimizable, which would have benefited test cases such as `monica.l4`.



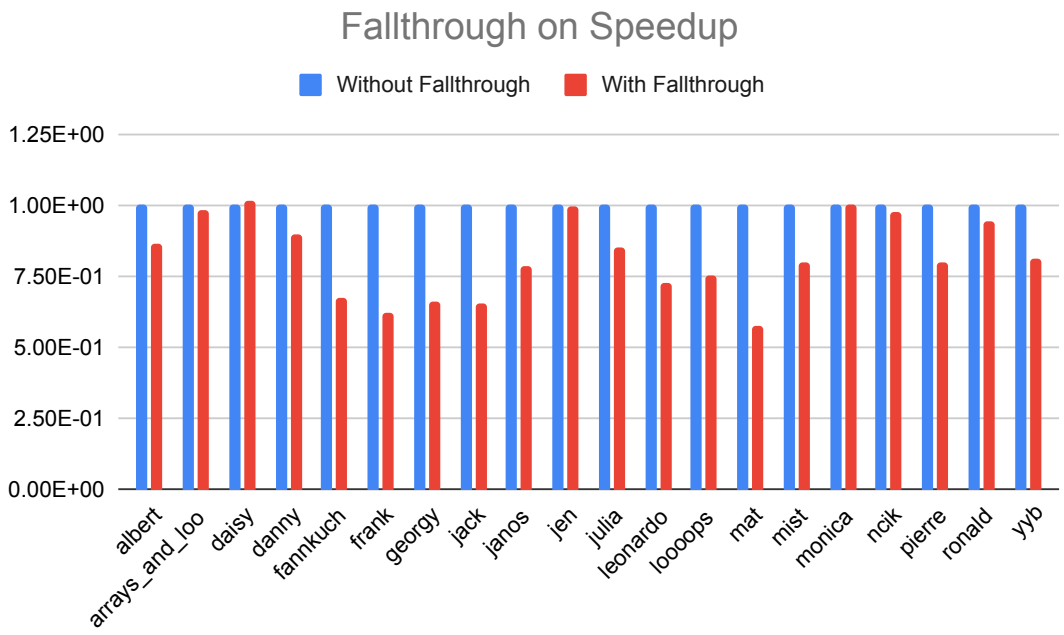
3.6 Code Layout and Improved x86 Codegen

Our compiler uses a variety of strategies to improve the generation of x86 code from abstract assembly. It is difficult to judge the effectiveness of these optimizations as they were built into the compiler. For example, we use LEA to calculate array and struct offsets instead of Mul and Add, which we were using in Lab 4. We also use liveness analysis information from register allocation to

calculate which registers must be pushed onto the stack for each function call, instead of pushing all caller-saved registers. Both of these optimizations had performance benefits on a large portion of the test programs.

Improving our code layout and implementing fallthrough was critical to achieving good performance from our compiler. By default, our CFG inserts jumps at the end of every basic block, even if the jump target immediately follows in the code. To improve this, we ordered our basic blocks such that "branch not taken" is the default path at every branch, and made this the fallthrough case at every branch. Doing so significantly improves the CPU's ability to prefetch instructions and improves locality of the instruction cache.

Shown below are the speedups we achieved in safe mode from implementing fallthrough. Adding fallthrough to our compiler resulted in nearly 40% performance increase for many benchmarks. The results are especially noticeable in safe mode, as memory and null checks add a lot of basic blocks and jumps to the program.



We also experimented with improving code alignment for branch targets and function calls using assembly directives such as `.align`. Our results varied for different programs. On some programs, improving code alignment gave better performance, possibly by allowing the instructions of an inner loop to fit in a single cache line. On other programs, padding the code with `nop`'s increased the code size and gave worse performance, possibly by reducing locality or causing thrashing in the instruction cache. As we did not implement loop analysis, a better approach may have been to only align branch targets that are frequently visited, such as a loop guard or the start of a frequently called function.

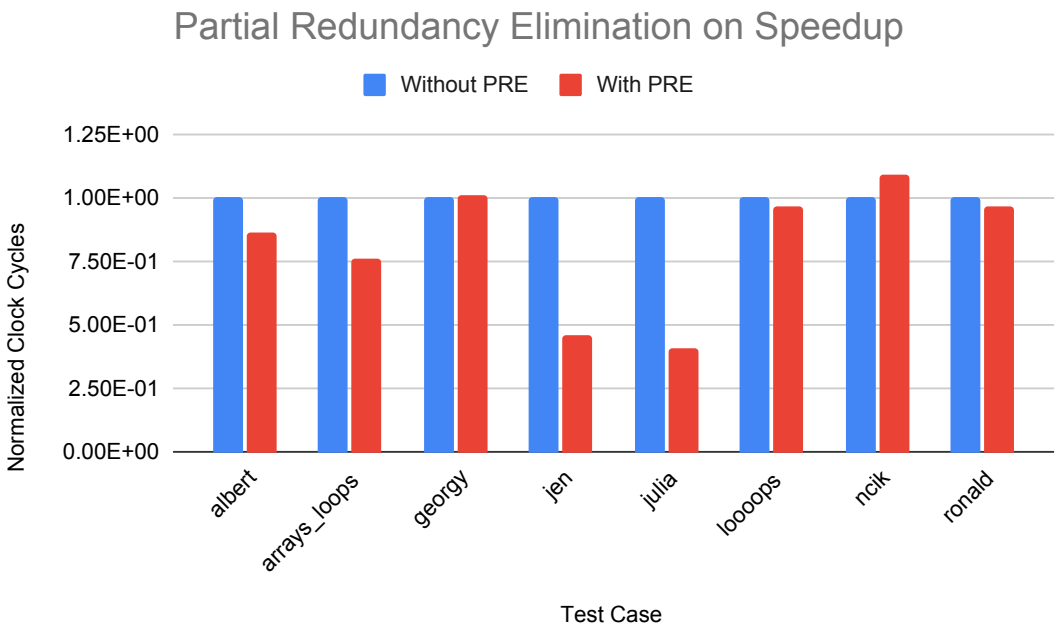
3.7 Partial Redundancy Elimination

We implemented the partial redundancy elimination (PRE) algorithm described in Section 9.5 of the Dragon Book. PRE performs the dual tasks of common subexpression elimination and loop-

invariant code motion, and aims to simplify the code by moving expressions around to a single optimal point of computation. As discussed in the Frontend section, we split critical edges and changed our loop generation code to create a location where redundant expressions could be placed.

PRE would sometimes interact with our local optimizations by moving expressions out of a basic block before they had the chance to participate in local optimizations that would have improved code quality. For this reason, we first perform a round of iterated local optimizations without PRE, then we perform another round of iterated optimization with PRE.

We show the impacts of PRE below. PRE significantly improves runtime for some tests such as `arrays_and_loops.l4` and `julia.l4`, which both contain redundant expressions in loops that can be hoisted out. However, while PRE can reduce the number of overall computations, splitting critical edges can also increase the number of jumps, which may cancel out the optimization benefit from reducing computations. For `nick.l4`, there are not many redundant expressions, but many loops, so applying PRE actually increases the runtime.



3.8 Iterated Optimization Passes

We run all the previously described optimization passes in a loop until the number of instructions converges. This allows optimizations to uncover each other until no more progress is possible. Below, we show the number of instructions of each type in the `jen.l4` benchmark after each round of simplification. Each round of optimization continues to improve the quality of the code, with diminishing returns as the number of rounds increases:

Round	Add	And	Call	Cmp	Jump	JumpC	JumpZ	Mov	Mul	Or	Phi	Sub	Xor
SSA	280	217	2	82	1312	393	175	5038	198	177	2484	1094	202
Round 1	135	55	2	16	392	158	47	463	46	50	497	68	38
Round 2	41	19	2	5	114	26	14	78	8	8	128	10	8
Round 3	40	18	2	5	111	26	14	75	8	8	128	9	8
Round 4	19	3	2	2	49	10	7	42	3	2	51	3	3

4 Advanced Optimizations

In addition to the optimizations above, we also implemented pointer / alias analysis. Alias analysis aims to determine when two pointer expressions may refer to the same memory location. Steensgaard's algorithm uses a union find algorithm to find points-to relationships between variables. We also implemented a dataflow-based algorithm based on the Appel book.

4.1 Steensgaard's Algorithm

Steensgaard's algorithm is a points-to-analysis algorithm that maintains a union-find graph to track relationships between pairs of variables `p` and `q`, where a directed edge `(p,q)` on the graph means that `p` possibly points to the location of `q`. Although Steensgaard's algorithm is not as precise as other existing pointer analysis algorithms such as Anderson's, it is much cheaper and runs in linear time as opposed to $O(n^3)$ for Anderson's. Given that many programs and benchmark tests involve large amounts of variables, it is more cost-efficient to implement a faster pointer analysis algorithm that can run within the time bounds of our compiler.

Steensgaard's algorithm is both flow-insensitive and field-insensitive. Flow insensitivity means that the points-to relatoins are computed over the entire program, no matter what order the statements are executed in, and field insensitivity means that the algorithm cannot distinguish between different fields of a struct (e.g. `p.x` and `p.y`) even though they may never alias.

Both of these limitations reduce the quality of the outputted pointer analysis, but make it possible to run the algorithm in linear time. For example, consider the following test program:

```
struct IntPtr { int f1; int* f2; }
struct IntPtr* s = alloc(struct IntPtr);
f(s->f1, s->f2);
```

Since `s->f1` and `s->f2` are different types, they can never alias, but according to Steensgaard's algorithm they are both memory locations related to `s`, and so our implementation of Steensgaard's algorithm would indicate that they may alias. Supplementing Steensgaard's algorithm with type information for each variable and struct field would be one way to mitigate this limitation, although it would increase the runtime of the algorithm and require significant re-engineering of our compiler's frontend to pass around type information past typechecking.

4.1.1 Implementation Details

Our implementation of Steensgaard's algorithm uses a hash table to represent a union find graph, and iterates linearly through the code to compute a set of alias classes from the variables. A graph

is drawn between the alias classes to indicate points-to relationships between those variables. Any time a pointer dereference, pointer copy, or pointer assignment is encountered, the relevant nodes are joined in the points-to graph to indicate that they possibly refer to the same memory location.

Each node in the hash table is either a root node or a leaf node, with leaf nodes pointing further along the tree to other root nodes and each root node pointing to another node further down the tree. Conceptually, a root node represents a class of variables that may alias each other, for example the set of variables that many point to a specific location that was `calloc`'ed in the heap. In order to look up the alias class of a variable, we must simply traverse the tree until encountering a root node, taking care to perform path compression to reduce the lookup complexity of the union find graph.

Steensgaard's algorithm could be further improved with type information, as C0 is a type-safe language with no aliasing possible between values of different types.

4.1.2 Results

Our alias analysis impleentation based on Steensgaard's algorithm produces some limited speedups on certain benchmarks, including `danny.l4` on safe mode where it decreases the runtime from $3.61 \cdot 10^8$ cycles to $3.44 \cdot 10^8$ cycles. In this test case, there are a large number of moves between memory locations and variables, and our constant and copy propagation benefit from not having to kill all memory values at each memory operation.

5 Dataflow Based Alias Analysis

Our dataflow-based alias analysis algorithm is based on Section 17.5 of the Appel book. This algorithm maps each variable and temp to possible points of origin, which refers to the allocated memory location that the variable points to. These points of origin are generated by either a call to `calloc`, a function return value, or from the stack frame. We then use a forward-may analysis to find possible points of origins for all variables.

Unfortunately, this analysis is less precise than we would like it to be. Consider a function call that we have no additional compiler-level information about. If the destination is a pointer and the function call takes in pointers as arguments, it could be that this function call manipulates the input pointer and returns it, or returns a newly allocated pointer. Conservatively, we must assign the function's return value to all input points of origin in addition to a newly alloated point of origin, which reduces the accuracy of the analysis.

The accuracy further decreases when we consider pointers from the stack frame. Without inter-procedural analysis, we do not know how these pointers are aliased to each other, so these pointers must all have the same point of origin: the stack frame, and any pointer that was a function argument or dereferences a pointer from the stack must have the stack frame as a possible point of origin. Therefore, in a function call with no allocations where all manipulated pointers are from the stack, there is no additional benefit to alias analysis.

This algorithm can also be improved with type information. If we keep track of the types for each variable and temp, we only need to union all points of origin of the same type for function calls, and the stack frame pointers can be further differentiated by type. However, our current alias analysis algorithm does not take into account types. Since we do not keep track of types, however, one small optimization we do is to maintain a map of allocations to allocations. For example, in the

following program:

```
int** x = alloc(int*);
*x = alloc(int);
y = *x;
```

In line 3, rather than giving `y` all possible points of origin as the algorithm in the book suggests, we know that its point of origin comes from line 2. We map the first allocation to the second allocation, and this signals that the dereference of the first allocation has a point of origin in the second allocation. This allows us to have some benefits of a type-based alias analysis without actually having type information, especially in settings with nested allocations (such as within structs).

This algorithm is costly on large programs. For each line, we potentially have to iterate through $O(n)$ different points of origins to find all possible points of origin for that variable, where n is the number of lines in the program. We run through each line at most once per dataflow pass, so each dataflow pass is $O(n^2)$, implying that the overall algorithm is $O(n^3)$.

The results of the alias analysis were integrated with constant and copy propagation (`constprop_pa.ml` and `copyprop_pa.ml`), as well as aggressive dead code elimination (`adce_pa.ml`). This was tested on the lab4-large test suite for correctness. For constant and copy propagation, the results of alias analysis greatly helps on test cases like the following (named `alias.l4`):

```
int[] A = alloc_array(int, 5);
int[] B = alloc_array(int, 5);

int sum = 0;
for (int i = 0; i < 5; i++) {
    A[i] = i;
    B[i] = i;
    sum += A[i] + B[i];
}
return sum;
```

Without alias analysis, we would not know if `A` and `B` are alias to each other, so `A[i]` must be killed since we write to memory. However, with copy propagation, we can optimize this to calculate `sum += i + i` instead of `sum += A[i] + B[i]`, which reduces the number of instructions needed for dereferencing and removes the need for reading from memory to calculate `sum`.

Alias analysis also helps ADCE. In ADCE, all writes to memory are marked as critical instructions, since they may modify the pointers passed in as arguments, or they might be needed in the final result. With alias analysis, we can identify writes to memory that have no effect on the output: in the test case above, we no longer need `A` and `B`, which can be removed in dead code elimination.

Looking at the list of instructions generated by the compiler, we can see the effect of alias analysis together with copy propagation, constant propagation, and ADCE:

	`Add`	`Call`	`Jump`	`JumpC`	`Lea`	`Mov`
Without Alias Analysis	3	2	4	1	4	16
With Alias Analysis	3	0	4	1	0	8

Consider the test case `albert.l4` from the benchmark suite. If we remove the use of `w[j]` on line 164, `w` is not needed at all to calculate the final result. With alias analylsis, we are able to identify that `w` is not aliased to anything else, and we can remove all writes to `w`, which removes a number of loops from the output result. The impacts of this optimization are shown below:

In safe mode, there is no improvement and in fact the runtime becomes worse, as ADCE cannot remove writes to memory in safe mode due to the possibility of error. As before, adding constant and copy propagation without the benefit of ADCE can increase the register pressure as the live range of temps is extended. However, in unsafe mode, there is a significant 5x speedup when we add alias analysis, by removing the loop that modified `w`.

5.1 Steensgaard vs. Flow-Based Alias Analysis

Steensgaard's algorithm has a much faster runtime than flow-based alias analysis, as its runtime is $O(n)$ compared to $O(n^3)$. As a result, the flow-based algorithm is too slow to run on large test cases. For example, in the `l4-basic` suite, Steensgaard's finishes while the flow-based algorithm while the flow-based algorithm times out on `bigstruct.l4`. For this reason, we submitted Steensgaard's pointer analysis. In addition, the flow-based algorithm has very little effect when few calls to `calloc` are made, as explained above, whereas Steensgaard's algorithm is able to track points-to relationships more specifically. In programs where pointers are commonly passed in through function calls rather than `calloc`'ed, we observed that Steensgaard's algorithm was better.

6 Results

We compare our compiler denoted `c0c` against both `gcc -O0` and `gcc -O1`, and report the runtime of each test case in clock cycles. Our compiler is run in both safe and unsafe mode: safe mode includes runtime checks for memory safety and undefined floating-point behavior, while the `--unsafe` flag allows the compiler to ignore any exceptions that might be raised during the execution of the program, except ones due to `assert`.

We also compute a benchmark score to measure how far our compiler is between `gcc -O0` and `gcc -O1`. Let s_c and u_c denote the average of the k-best times from our compiler in safe and unsafe mode respectively. Let s_0 and s_1 denote the times of `gcc -O0` in safe mode, adn let u_0 and u_1 denote the times of `gcc -O1` in unsafe mode. Then, our benchmark score P_{time} is computed as follows, where both P_s and P_u are clamped between 0 and 2.5:

$$P_s = 1 - \frac{s_c - s_1}{s_0 - s_1} \qquad P_u = 1 - \frac{u_c - u_1}{u_0 - u_1} \qquad P_{\text{time}} = \frac{P_s + P_u}{2}$$

We report benchmark scores for each test in the table below:

Name	`gcc -0s`	`gcc -1s`	`c0c -s`	P_s	`gcc -0u`	`gcc -1u`	`c0c -u`	P_u	P_{time}
`albert`	$12.4 \cdot 10^9$	$8.56 \cdot 10^9$	$2.34 \cdot 10^9$	2.500	$5.08 \cdot 10^9$	$1.61 \cdot 10^9$	$2.16 \cdot 10^9$	0.841	1.671
`arrays_loops`	$13.9 \cdot 10^9$	$8.31 \cdot 10^9$	$4.58 \cdot 10^9$	1.667	$7.81 \cdot 10^9$	$3.45 \cdot 10^9$	$4.38 \cdot 10^9$	0.787	1.227
`daisy`	$8.23 \cdot 10^9$	$7.04 \cdot 10^9$	$3.28 \cdot 10^9$	2.500	$2.35 \cdot 10^9$	$6.33 \cdot 10^8$	$1.48 \cdot 10^9$	0.507	1.503
`danny`	$3.34 \cdot 10^9$	$2.70 \cdot 10^9$	$1.12 \cdot 10^9$	2.500	$9.19 \cdot 10^8$	$2.71 \cdot 10^8$	$3.44 \cdot 10^8$	0.887	1.694
`fannkuch`	$57.0 \cdot 10^9$	$50.8 \cdot 10^9$	$19.1 \cdot 10^9$	2.500	$16.9 \cdot 10^9$	$7.30 \cdot 10^9$	$6.24 \cdot 10^9$	1.110	1.805
`frank`	$6.38 \cdot 10^8$	$4.54 \cdot 10^8$	$1.65 \cdot 10^8$	2.500	$2.68 \cdot 10^8$	$7.48 \cdot 10^7$	$9.12 \cdot 10^7$	0.915	1.708
`georgy`	$1.65 \cdot 10^9$	$1.26 \cdot 10^9$	$6.06 \cdot 10^8$	2.500	$6.88 \cdot 10^8$	$3.34 \cdot 10^8$	$4.74 \cdot 10^8$	0.605	1.552
`jack`	$1.53 \cdot 10^9$	$1.26 \cdot 10^9$	$5.40 \cdot 10^8$	2.500	$2.48 \cdot 10^8$	$8.55 \cdot 10^7$	$2.57 \cdot 10^8$	0.000	1.250
`janos`	$4.01 \cdot 10^8$	$3.30 \cdot 10^8$	$1.80 \cdot 10^8$	2.500	$2.31 \cdot 10^8$	$1.90 \cdot 10^8$	$1.72 \cdot 10^8$	1.439	1.970
`jen`	$11.9 \cdot 10^9$	$1.78 \cdot 10^9$	$9.33 \cdot 10^7$	1.167	$11.9 \cdot 10^9$	$1.78 \cdot 10^9$	$9.33 \cdot 10^7$	1.167	1.167
`julia`	$7.92 \cdot 10^9$	$4.39 \cdot 10^9$	$4.11 \cdot 10^9$	1.079	$7.02 \cdot 10^9$	$2.96 \cdot 10^9$	$4.47 \cdot 10^9$	0.628	0.854
`leonardo`	$5.05 \cdot 10^9$	$3.32 \cdot 10^9$	$4.61 \cdot 10^9$	0.254	$4.93 \cdot 10^9$	$3.30 \cdot 10^9$	$4.61 \cdot 10^9$	0.196	0.225
`loooops`	$10.6 \cdot 10^9$	$7.92 \cdot 10^9$	$5.48 \cdot 10^9$	1.910	$8.04 \cdot 10^9$	$2.53 \cdot 10^9$	$5.47 \cdot 10^9$	0.466	1.188
`mat`	$7.17 \cdot 10^9$	$6.64 \cdot 10^9$	$2.10 \cdot 10^9$	2.500	$1.87 \cdot 10^9$	$5.49 \cdot 10^8$	$7.31 \cdot 10^8$	0.862	1.681
`mist`	$9.44 \cdot 10^9$	$8.71 \cdot 10^8$	$2.54 \cdot 10^8$	1.072	$9.31 \cdot 10^9$	$7.49 \cdot 10^8$	$2.27 \cdot 10^8$	1.061	1.066
`monica`	$2.37 \cdot 10^9$	$1.80 \cdot 10^9$	$1.61 \cdot 10^9$	1.333	$2.34 \cdot 10^9$	$1.78 \cdot 10^9$	$1.61 \cdot 10^9$	1.304	1.318
`ncik`	$6.83 \cdot 10^9$	$5.52 \cdot 10^9$	$3.68 \cdot 10^9$	2.405	$3.17 \cdot 10^9$	$1.15 \cdot 10^9$	$2.51 \cdot 10^9$	0.327	1.366
`pierre`	$2.58 \cdot 10^7$	$1.46 \cdot 10^7$	$7.76 \cdot 10^6$	1.611	$2.45 \cdot 10^7$	$1.43 \cdot 10^7$	$7.79 \cdot 10^6$	1.638	1.624
`ronald`	$2.29 \cdot 10^9$	$1.78 \cdot 10^9$	$1.42 \cdot 10^9$	1.706	$9.22 \cdot 10^8$	$3.96 \cdot 10^8$	$1.00 \cdot 10^9$	0.000	0.853
`yyb`	$10.6 \cdot 10^9$	$8.51 \cdot 10^9$	$2.55 \cdot 10^9$	2.500	$2.38 \cdot 10^9$	$7.82 \cdot 10^8$	$1.09 \cdot 10^9$	0.807	1.654
Average				1.960				0.777	1.369

Overall, our compiler outperforms the reference compiler in safe mode with runtime checks are enabled, and is competitive with `gcc -O1` on some (but not all) test cases in unsafe mode. Our average benchmark score is 1.960 across safe tests, 0.777 across unsafe tests, and 1.369 overall.

7 Future Improvements

Based on the benchmarks, our optimizations help improve performance on most test cases, particularly improved register allocation, constant and copy propagation, constant folding, peephole optimizations, and aggressive dead code elimination. Other optimizations such as function inlining, tail call optimization, and PRE do little by themselves, but allow other optimizations to be far more effective. Our compiler achieves the worst speedup on `julia.l4`, `leonardo.l4`, `monica.l4`, and `ronald.l4`.

7.1 Function Inlining Heuristics

We observed that adding function inlining made `julia.l4` perform significantly worse, because we currently choose to inline functions that are either small or called only once, with no other metrics or heuristics to inform function inlining. Implementing loop analysis would allow us to use loop depth as an additional heuristic to determine if a function should be inlinable or not. We could also use the register pressure of the caller and callee fnctions to inform whether to inline, as inlining under an already-high register pressure could potentially cause spills and negate the performance benefit of eliminating function call overhead.

7.2 Global Constant Propagation

Using global constant propagation as opposed to local constant propagation could also help speed up `julia.l4`. The `julia` function contains a loop that has no effect on the return value, but in order to remove it with ADCE, we would need to know that `DIM/2` is nonzero. As we can only propagate within blocks, we do not know that `DIM` is deterministically 144, so ADCE cannot remove the instruction as it may cause side effects.

One wya to implement global constant propagation under our local optimization framework is to add an additional move at the top of any basic block guarded by an equality condition. For example, in while generating the if-statement `if (x == 1) { ... }`, we could insert an additional assignment statement `x<-1` at the top of the loop contents. In the best case, this gives our local optimizations additional information to perform constant and copy propagation as they do now. In the worst case, this assignment statement has no effect and is removed later by ADCE.

7.3 Register Allocation and Loop Analysis

Loop analysis would give us better heuristics for what variables to spill during register allocation. Our current heuristic only takes into account the number of appearances in the code and not loops, which may cause inner loop variables to be spilled and reloaded many times. Especially on `ronald.l4` which spills many variables, our register allocator does not have the best performance.

7.4 Tail Call Optimization and Basic Accumulation

As mentioned previously, `monica.l4` has functions that are close to tail call optimizable, but require basic accumulation in order to perform this optimization. To improve `leonardo.l4`, we can also either optimize our function calls, or unroll the function and turn it into a loop.

7.5 Alias Analysis Type and Field Sensitivity

Further improvements are possible for our alias analysis algorithms. For dataflow-based alias analysis, we could make it type-sensitive and field-sensitive. As discussed previously, passing around type information could reduce the number of possible points of origins that would need to be assigned to a pointer in ambiguous situations, including function calls and the stack frame. By adding field-sensitive information, we can further distinguish between pointers. For example, consider the following code:

```
struct x { int* a; int *b; };
struct x* var = alloc(struct x);
var->a = alloc(int);
var->b = alloc(int);
```

Our algorithm currently does not distinguish between the points of origin on lines 2 and 3, so `var->a` could refer to both of these points of origin as we do not know which allocation relates to each field. Adding field information would further increase the accuracy of alias analysis.

8 References

1) Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. "Compilers: Principles, Techniques, and Tools." *Pearson Education Inc.*, 2006.

2) Adrew A. Appel. "Modern Compiler Implementation in ML." *Cambridge University Press*, 1998.

3) Keith D. Cooper and Linda Torczon. "Engineering a Compiler." *Elsevier Science*, 2004.

4) Fernando Magno Quintao Pereira and Jens Palsberg. "Register Allocation via Coloring of Chordal Graphs." *APLAS*, 2005.

5) Torbjorn Granlund and Peter L. Montgomery. "Division by Invariant Integers Using Multiplication." *PLDI*, 1994.