Abstract

In this report, we present our findings from implementing the -O1 and --unsafe flags for our L4 C0 compiler.

With or without optimizations, our compiler uses the same LL(1) backtracking parser and typed-AST front end. However, unlike the base L4 compiler, our L5 compiler uses a new codegen strategy that reduces the number temps created and creates more specific instructions for conditional jumps and tail calls. Our compiler's x86_64 backend now does many peephole optimizations, as well as tail call optimization, in all cases. In unsafe mode, we omit safety checks for out-of-bound shifts, null dereferences, and out-of-bound array access,

With the -O1 flag enabled, our compiler converts our codegen output (linear abstract assembly) into a CFG IR before further conversion into CSSA. In this form, we implement the SCCP algorithm, which encompasses constant/copy propagation and eliminates dead branches. With optimizations enabled, our register allocator will also use dominance information computed from the CFG in order to prioritize coloring temps used in nested loops, and will attempt to coalesce temps that only interfere due to move edges.

See the results section for quantitative benchmarks of each category of optimizations.

Methods

Codegen

Our abstract assembly codegen algorithm is implemented in **codegen.rs**. Unlike in previous labs, our recursive muncher now returns an operand (i.e. a temp or a constant) that an expression was munched into, rather than taking a destination Temp as a parameter. This allows for many expressions, for example constant assignments, to generate fewer Temps and move instructions. The codegen algorithm also recognizes new special cases, including conditional jumps based on the result of a comparison operator and tail-call returns.

Peephole Optimizations

These optimizations are found in **emit.rs**. We have implemented the following:

- Multiplication - emit_mul
    - Use shifts when one operand is a constant of the form $\pm\ 2^n$
- Division - emit_div
    - Use shifts when one operand is a constant of the form $\pm\ 2^n$
    - Use bitwise ands for modulo when one operand is a constant of the form $\pm\ 2^n$
- Jumps - emit_cjump, emit_jump, emit_cjump_op
    - Ignore jumps to the subsequent line
    - Use conditional jump instructions where possible (instead of munching the condition to its own temp)
    - Flip and ignore jumps we can re-order, e.g. (cjmp 1, jmp 2, [1] -> !cjmp 2, [1])
    - Reduce const conditional jumps to unconditional jumps
    - Reduce tail call returns to parameter setup and a jump to the function header
    - Automatically flip conditional jumps with constant-folded second arguments
- Moves - emit_mov, write_mov
    - Ignore moves to self
- Functions - emit_func, emit_call
    - 16-byte align the stack at the beginning instead of at each inner call
- Registers
    - Reorder register allocation priority to discourage use of callee saved registers when few registers are required
- Indexing - emit_index
    - When the index stride is 1,2,4, or 8, use the x86 memory access syntax to do the multiplication automatically
- Unsafe mode - emit_index, emit_shift, emit_deref, emit_lea
    - Omit array index bounds checking, lea null checking, dereference null checking, invalid shift checking, and allocation null checking.

SSA

Our algorithm to construct CSSA is heavily based on the notes found in the SSA lectures and SSA recitation. “A Simple, Fast Dominance Algorithm” was used to calculate the dominance tree and frontier. The placement of removal of phi functions was based on the SSA lecture notes. To solve the missing copy problem, critical edges were split. For the parallel copy problem, a DFS was performed on the Spartan Transfer graph to find the correct ordering/temps involved in the moves. Details can be found in **ssa.rs**, in the implementations of the SSAContext and SSACFG types. We list the high level steps below:

- The main function creates an SSAContext with the linear abstract assembly program, and converts it to SSA with SSAContext::init()
    - Linear abstract assembly is converted to a CFG in SSAContext::ssa_func()
    - A graph (**graph.rs**), is constructed with the cfg information and the dominance tree/frontier is computed on it
    - All phi functions are renamed in SSAContext::rename()
    - Deconstruction begins with SSAContext::split_all_crit()

- The main function may run other optimizations

- The main function deconstructs SSA with SSAContext::deconstruct() and receives a new linear abstract assembly program
    - Phi function moves are generated with SSAContext::rem_phi()
    - Spartan Transfer Graph is built with SSAContext::move_order()

SCCP

We implemented Sparse Conditional Constant Propagation based on the paper by [Wegman, Zadeck]. Our implementation can be found in **ssa.rs**, within the implementations of the SCCPLattice and SCCPContext types. SCCP is structured as an optional pass that may be run between building and deconstructing SSA, and simply updates the SSA CFGs in place. It is called from SSAContext::run_sccp(), which in turn constructs an SCCPContext and runs SCCPContext::run(). Our implementation follows the SCCP algorithm almost exactly, with the following steps. Note that we do not attempt to constant-propagate pointer values.

- Compute def-use mapping via a BFS on the CFG and iterating each basic block in SCCPContext::compute_def_use().
- Initialize the lattice and set function arguments to Bottom.
- Initialize edge flow work list with the entry block & an empty SSA def-use work list. Iterate SCCP work rules until convergence.
    - Phi functions are updated using SCCPContext::visit_phi(), and will compute the meet of all arguments coming from executable blocks.
    - Other instructions are updated using SCCPContext::visit_expression(). Binary/unary operations and moves will attempt to constant-fold their arguments

using the lattice rules in SCCPContext::eval_unop() and SCCPContext::eval_op(). Conditional jumps will also attempt to fold their conditions. If successful, they only add one branch to the edge flow list, and both otherwise. Dereferences and calls will always make their destinations non-constant.

- Once the previous pass has computed the lattice value for every temp in the CFG, the CFG is re-built in SCCPContext::cull_cfg(), omitting any blocks not marked as executable, instructions that result in constant results, and folding constant temps into individual instructions where they are used.

Register Allocation

Previously, our register allocation was performed on an interference graph including X86 registers (i.e. pre-colored nodes) using the chordal SEO coloring algorithm discussed in lecture. To save callee-saved registers, we move them into/from unique temps at the start/end of the function, respectively.

When optimizations are enabled, we improve the chordal coloring algorithm using dominance frontier information by computing the loop depth of each temp in our CFG. To do so, we use the loop-finding algorithm described in lecture. Subsequently, we initialize the weights of the temps in SEO by their loop depth. This causes the SEO to start with temps in the deepest loop depth, hence attempt to assign them a color first. We spill all temps with a color assigned to them larger than the number of available registers, so by coloring temps with a higher loop depth first, they are more likely to not be spilled.

With optimizations, our register allocator will also perform a move coalescing step. After creating an initial coloring of the graph, it will detect all edges between temps that only interfere due to moves between the two temps. If possible, we contract vertices connected by move edges, assigning them the same register color. No further changes are made to the assembly IR, since our peephole optimizations of not emitting moves from a register to itself will automatically ignore coalesced moves.

Code for register coloring may be found in **coloring.rs**, specifically color_fn(). To compute the liveness analysis we use **cfg.rs** and the flow algorihtm from the L2 Checkpoint. We do the graph coloring in **graph.rs**, specifically coloring() and simplicial_ordering(). Move coalescing may be found at the end of the color_fn().

Results

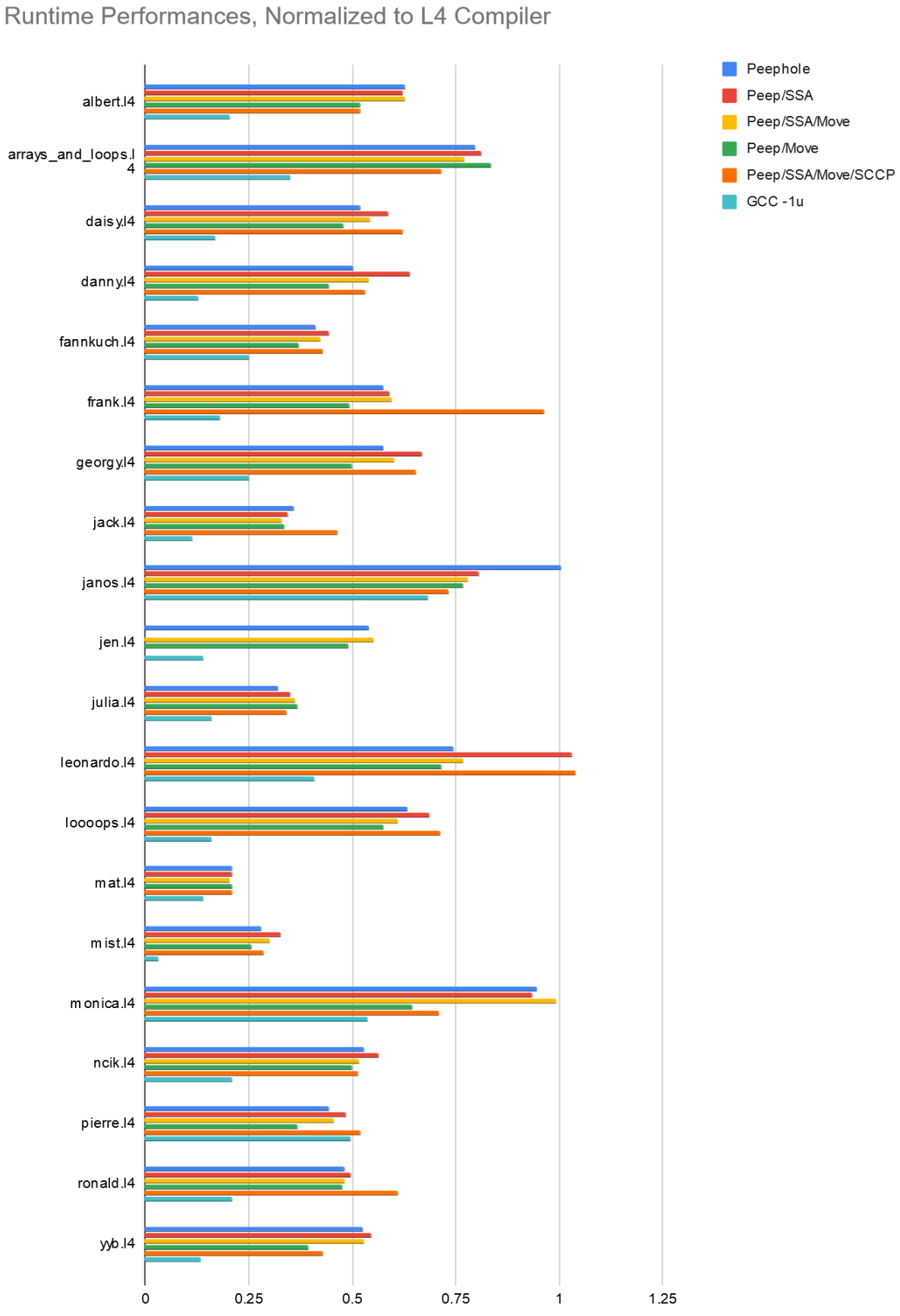
Our compiler was benchmarked on the given tests in the tests/bench directory of the dist repo. To evaluate metrics of speed and code size, we used the Lab 5 submission system on Notolab to evaluate the cycles taken and size of our executable.

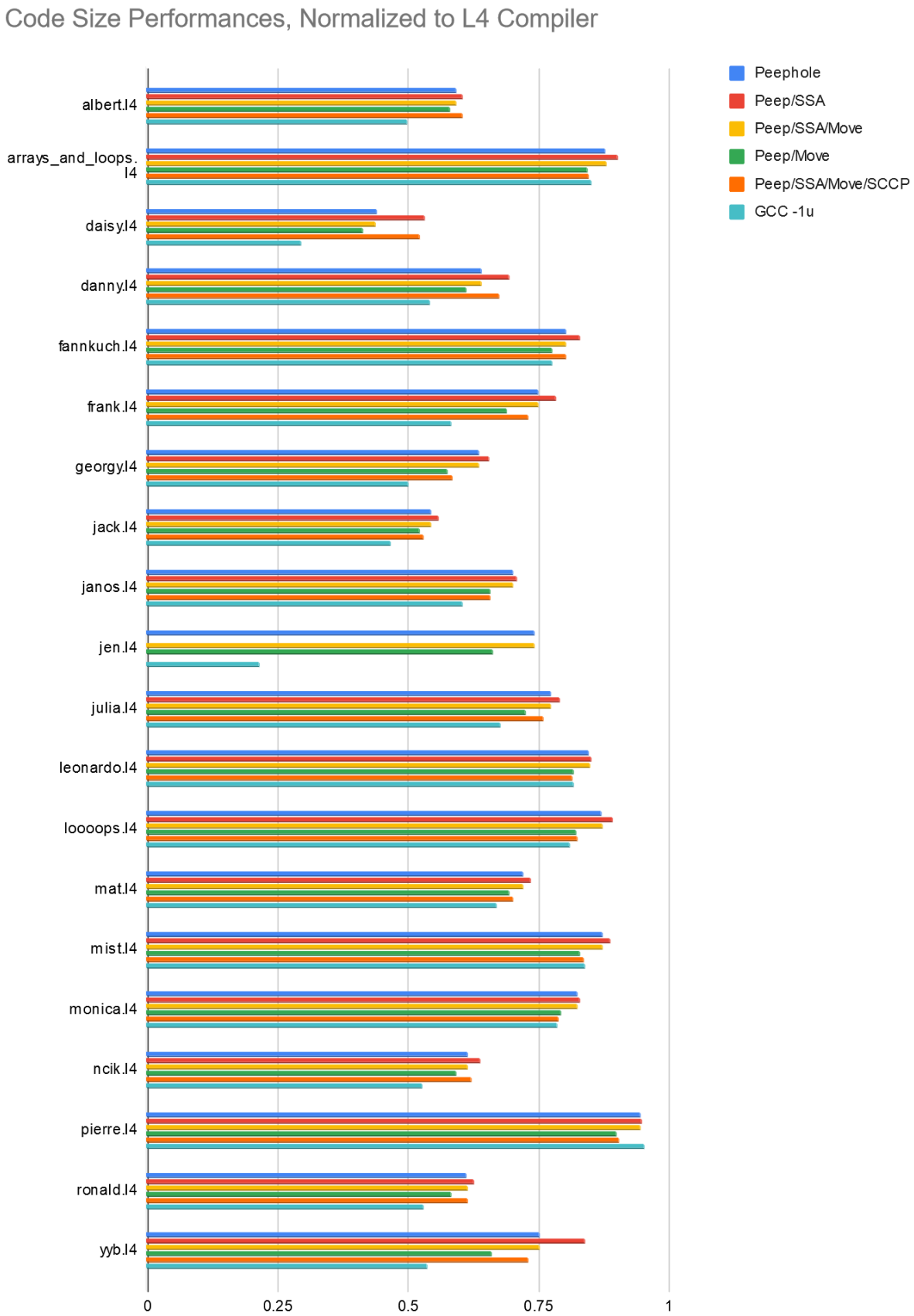
For each test, we've normalized the runtime/codesize with respect to our L4 compiler. The table below presents the average of these values over all 20 tests in the benchmark.

All tests of the L5 compiler were run in unsafe mode. Peephole optimizations include everything listed in the peephole section, as well as upgraded codegen and the loop-depth spilling heuristic. Move represents enabling move coalescing in the register allocator. SSA and SCCP represent enabling SSA and SCCP respectively.

Average of bench/L4 (lower is better)

Optimizations Enabled	Runtime	Code Size
Peephole	0.55	0.73
Peep/SSA	0.59	0.75
Peep/SSA/Move	0.55	0.73
Peep/SSA/Move/SCCP	0.58	0.71
Peep/Move	0.49	0.69
GCC -1u	0.25	0.62


Runtime Performances, Normalized to L4 Compiler
albert.l4
arrays_and_loops.l4
daisy.l4
danny.l4
fannkuch.l4
frank.l4
georgy.l4
jack.l4
janos.l4
jen.l4
julia.l4
leonardo.l4
loooops.l4
mat.l4
mist.l4
monica.l4
ncik.l4
pierre.l4
ronald.l4
yyb.l4
0
0.25
0.5
0.75
1
1.25
Peephole
Peep/SSA
Peep/SSA/Move
Peep/Move
Peep/SSA/Move/SCCP
GCC -1u


Code Size Performances, Normalized to L4 Compiler
Peephole
Peep/SSA
Peep/SSA/Move
Peep/Move
Peep/SSA/Move/SCCP
GCC -1u
albert.l4
arrays_and_loops.l4
daisy.l4
danny.l4
fannkuch.l4
frank.l4
georgy.l4
jack.l4
janos.l4
jen.l4
julia.l4
leonardo.l4
loooops.l4
mat.l4
mist.l4
monica.l4
ncik.l4
pierre.l4
ronald.l4
yyb.l4
0
0.25
0.5
0.75
1

Discussion

General Results

Compared to our L4 compiler, most reductions in both code size and runtime result from unsafe mode, peephole optimizations, and improved register allocation with loop detection + move coalescing. Our best result in terms of overall performance and code size came from enabling these three types of optimizations. The best configuration was comparable to gcc regarding code size across the board, and comparable regarding runtime in a handful of benchmarks.

When enabling SSA conversion, we note a significant increase in runtime/code size across most of the test suite. This makes sense, since by constructing/deconstructing SSA, we introduce many new temps and move instructions to emulate phi functions. A smarter register allocator (such as Chatin's allocator, as discussed in lecture) might do better at removing these excess moves, since our allocator only attempts to coalesce temps post-coloring.

The inclusion of SCCP with SSA results in improved code size (compared to SSA only) across the benchmark suite, as the algorithm will prune dead basic blocks, remove instructions that have constant results, and remove unnecessary temps/moves by folding constants. Runtime results with SCCP are improved in some tests due to the omission of extraneous instructions, but others suffer in run-time. Slower run-times were likely due to increased pressure on the register allocator, as short-lifetime temps are often folded away by SCCP.

Specific Tests

There are a few outliers in our testing data. For example, it seems that SCCP greatly increases the runtime for frank.l4. Unfortunately, we are unable to come up with a reason for this, as frank.l4 exhibits few places where constant folding would have an effect.

We also note our largest runtime improvement in pierre.l4 when compared to both GCC and L4. This is primarily due to our implementation of tail-call optimizations, which greatly speed up the pow and fermat functions that take up the entire runtime of the test.

Many tests we lag behind GCC on, such as arrays_and_loops.l4, jen.l4, and loooops.l4, exhibit many redundant or loop-invariant computations. Our optimizations on these tests could be greatly improved with the addition of PRE, which would eliminate redundancies and move invariant code out of loops. Strength reductions and smarter codegen, such as detecting hidden constants (e.g. x - x, i + j == j + i) and eliminating duplicate conditions, would also help here.

Lastly, in daisy.l4, all versions of our compiler seem to significantly trail GCC. Upon inspection, the implementation accesses many similar locations in memory, which could be improved upon by alias analysis.

Compiler Runtime

Almost all optimizations we implemented did not severely affect the runtime of the compiler, with the exception of SCCP in cases that include a huge number of temps. Specifically, our compiler is not able to run SCCP to completion on jen.l4 within the time limit, as it spends too much time iterating the SCCP SSA work lists to convergence. This makes sense, as the workload is highly dependent on the number of temps, which is gigantic for jen.l4 in SSA form.

SSA construction and deconstruction, on the other hand, is never a bottleneck, as its runtime primarily depends upon the number of basic blocks in a function. The work required to deconstruct phi functions can depend on the number of temps as well, but this was not found to be an issue in practice.

References

[1] https://www.cs.utexas.edu/users/lin/cs380c/wegman.pdf
[2] https://www.cs.rice.edu/~keith/EMBED/dom.pdf